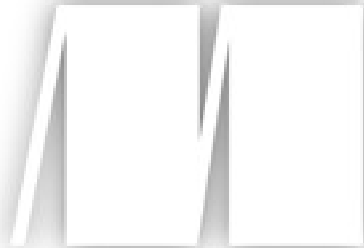
Creating Modular Applications in Java

Richard S. Hall
Karl Pauls
Stuart McCulloch
David Savage

MEAP Unedited Draft



MEAP Edition
Manning Early Access Program

www.wowebook.com

Copyright 2010 Manning Publications
For more information on this and other Manning titles go to www.manning.com

Table of Contents

1. OSGi Revealed
2. Mastering Modularity
3. Learning Lifecycle
4. Studying Services
5. Delving Deeper into Modularity
6. Moving Toward Bundles
7. Managing Bundles and Applications
8. Testing Applications
9. Debugging Applications
10. Component Models
11. Launching and Embedding an OSGi Framework
12. Security
13. Web Applications and Services

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

1

OSGi Revealed

The Java™ platform is an unqualified success story. It is used to develop applications for small mobile devices to massive enterprise endeavors. This is a testament to its well thought out design and continued evolution. However, this success has come in spite of the fact that Java does not have explicit support for building modular systems beyond ordinary object-oriented data encapsulation.

So, what does this mean to you? If Java is a success despite its lack of advanced modularization support, then you might wonder if its absence is a problem. Most well managed projects have to build up a repertoire of comparable, but project specific, techniques to compensate for the lack of modularization in Java. These include:

- Programming practices,
- Tricks with multiple class loaders, and
- Serialization between in-process components.

However these techniques are inherently brittle and error prone since they are not enforceable via any specific compile-time or run-time checks. The end result has detrimental impacts on multiple stages of an application's lifecycle:

- Development – you are unable to clearly and explicitly partition development into independent pieces.
- Deployment – you are unable to easily analyze, understand, and resolve requirements imposed by the collection of independently developed pieces that make up the system.
- Execution – you are unable to manage and evolve the constituent pieces of a running system, nor minimize the impact of doing so.

It is definitely possible to manage these issues in Java, and lots of projects do so using the custom techniques mentioned above, but it is much more difficult than it should be. We're tying ourselves in knots to work around the lack of a fundamental feature. If Java had explicit support for modularity, then you would be freed from such issues and could

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

concentrate on what you really want to do, which is developing the functionality of your application.

Welcome to the OSGi™ Service Platform. The OSGi Service Platform is an industry standard defined by the OSGi Alliance to specifically address the lack of support for modularity in the Java platform. Additionally, it also introduces a new service-oriented programming model, referred to by some as "SOA in a VM." This chapter will give you an overview of the OSGi Service Platform and the issues it is intended to address. Once we have finished this chapter we will have enough background knowledge to start digging into the details in chapter 2.

1.1 The what and why of OSGi

The sixty-four-thousand dollar question is, "What is OSGi?" The simplest answer to this question is it is a modularity layer for the Java platform. Of course, the next question that might spring to mind is, "What do you mean by modularity?" Here we use modularity more or less in the traditional computer science sense, where the code of your software application is divided into logical parts representing separate concerns as in Figure 1.1.1. If your software is modular, then you can simplify development and improve maintainability by enforcing the logical module boundaries; we will discuss more modularity details in Chapter 2.

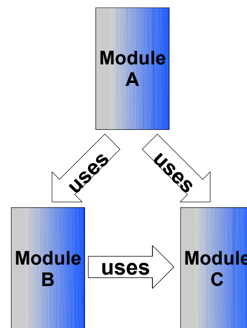


Figure 1.1.1 Modularity refers to the logical decomposition of a large system into smaller collaborating pieces

The notion of modularity is not new. The concept actually became fashionable back in the 1970s. So, why is OSGi all the rage right now? To better understand what OSGi can do for you, it is worthwhile to understand what Java is not doing for you with respect to modularity. Once you understand that, then you can see how OSGi can help.

1.1.1 Java's modularity limitations

Java was never intended to support modular programming, so we admit that criticizing its inability to do so is a little unfair. Java has been promoted as a platform for building all sorts

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

of applications for all sorts of domains ranging from mobile phone to enterprise applications. Most of these endeavors require, or could at least benefit from, modularity, so Java's lack of explicit support does cause some amount of pain for developers. From this point of view, we do feel the following criticisms are valid.

Low-level code visibility control

While Java provides a fair complement of access modifiers to control visibility (e.g., `public`, `protected`, `private`, and package private), these tend to address low-level object-oriented encapsulation and do not really address logical system partitioning. Java has the notion of a package, which is typically used for partitioning code. For code to be visible from one Java package to another, the code must be declared `public` (or `protected` if using inheritance). Sometimes the logical structure of your application calls for specific code to belong in different packages, but then this means any dependencies among the packages must be exposed as `public`, which makes it accessible to everyone else too. Often this can expose implementation details, which makes future evolution more difficult since users may end up with dependencies on your non-public API.

To illustrate this, let's consider a trivial hello world application that provides a public interface in one package, a private implementation in another and a main class in yet another.

Listing 1.1.1 Trivial example of the limitations of Java's object-orientated encapsulation

```
package org.foo.hello;

public interface Greeting {                                            #1
  void sayHello();
}
```

```
package org.foo.hello.impl;

import org.foo.hello.Greeting;

public class GreetingImpl implements Greeting {
  final String m_name;

  public GreetingImpl(String name) {                                   #2
    m_name = name;
  }

  public void sayHello() {
    System.out.println("Hello, " + m_name + "!");
  }
}
```

```
package org.foo.hello.main;

import org.foo.hello.Greeting;
import org.foo.hello.impl.GreetingImpl;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
public class Main {
  public static void main(String[] args) {
    Greeting greet = new GreetingImpl("Hello World");                    #3
    greet.sayHello();
  }
}
```

In Listing 1.1.1, the author may have intended a third party to only interact with the application via the `Greeting` interface at (#1). He or she may mention this in Javadoc, tutorials, blogs, or even email rants, but there is nothing actually stopping a third party from constructing a new `GreetingImpl` using its public constructor at (#2) as is done at (#3).

You might argue that the constructor should not be public and there is no need to split the application into multiple packages, which could well be true in this trivial example. But in real-world applications class-level visibility when combined with packaging turns out to be a very crude tool for ensuring API coherency. Seeing how a supposedly private implementation can be accessed by third-parties developers, now you need to worry about changes to private implementation signatures as well as that of public interfaces when making updates.

This problem stems from the fact that although Java packages appear to have a logical relationship via nested packages, they actually do not. A common misconception for people first learning Java is to assume that the parent-child package relationship bestows special visibility privileges on the involved packages. Two packages involved in a nested relationship are equivalent to two packages that are not. Nested packages are largely useful for avoiding name clashes and provide only partial support for the logical code partitioning.

What this all means is in Java you are regularly forced to decide between:

1. Impairing your application's logical structure by lumping unrelated classes into the same package to avoid exposing non-public API or
2. Keeping your application's logical structure by using multiple packages at the expense of exposing non-public API so it can be accessed by classes in different packages.

Neither choice is particularly palatable.

Error-prone class path concept

The Java platform also inhibits good modularity practices. The main culprit is the Java class path. Why does the class path pose problems for modularity? Largely due to all of the issues that it hides, such as code versions, dependencies, and consistency. Applications are generally composed of various versions of libraries and components. The class path pays no attention to code versions, it simply returns the first version that it finds. Even if it did pay attention, there is no way to explicitly specify dependencies. The process of setting up your class path is largely trial and error; you just keep adding libraries until the VM stops complaining about missing classes.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

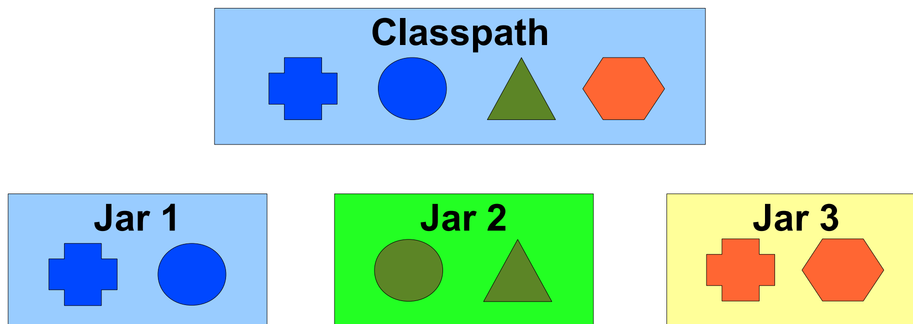


Figure 1.1.2 Multiple JARs containing the "same" class are merged based on their order of appearance in the class path paying no respect to logical coherency between archives

Figure 1.1.2 shows the sort of "class path Hell" that can often be found when more than one JAR file provides a given set of classes. Even though each JAR file may have been compiled to work as a unit, when merged at run time the Java class path pays no attention to the logical partitioning of the components. This tends to lead to such hard to predict errors, such as `NoSuchMethodError`, when a class from one JAR file interacts with an incompatible class from another one.

In large applications, created from independently developed components, it is not uncommon to have dependencies on different versions of the same component, such as logging or XML parsing mechanisms. The class path forces you to choose just one version in such situations, which may not always work for all parts of the application if there are incompatibilities between versions. Worse, if you happen to have multiple versions of the same package on the class path, either on purpose or accidentally, they are treated as split packages by Java and are implicitly merged based on order of appearance. Overall, the class path approach lacks any form of consistency checking. You just get whatever classes have been made available by the system administrator, which is very likely only an approximation of what the developer actually expected. This hardly inspires confidence.

LIMITED DEPLOYMENT AND MANAGEMENT SUPPORT

Java also lacks support when it comes to deploying and managing your application. There is no easy way in Java to deploy the proper transitive set of versioned code dependencies and execute your application. Likewise for evolving your application and its components after deployment. Consider the common requirement of wanting to support a dynamic plugin mechanism. The only way to achieve such a benign request is to use class loaders, which are low level and error prone. Class loaders were never intended to be a common tool for application developers, but so many of today's systems require their use. A properly defined modularity layer for Java can deal with these issues by making the module concept explicit and raising the level of abstraction for code partitioning.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

With this better understanding of the limitations of Java when it comes to modularity, we can ponder whether OSGi is the right solution for your projects.

1.1.2 Can OSGi help you?

Nearly all but the simplest of applications can benefit from the modularity features OSGi provides, so if you are wondering if OSGi is something you should be interested in, the answer is most likely, "Yes!" Still not convinced? Here are some common scenarios that you may have encountered where OSGi can be helpful:

- If you ever received `ClassNotFoundException`s when starting your application because the class path was not correct. OSGi can help you here by ensuring that code dependencies are satisfied before allowing the code to execute.
- If you ever encountered run-time errors when executing your application due to the wrong version of a dependent library on the class path. OSGi verifies that the set of dependencies are consistent with respect to required versions and other constraints.
- If you ever wanted to share classes between modules without worrying about constraints implied by hierarchical class loading schemes; put in a more concrete way, the dreaded appearance of "`foo instanceof Foo == false`" when sharing objects between two servlet contexts.
- If you ever wanted to package your application as logically independent JAR files and be able to deploy only those pieces you actually need for a given installation. This pretty much describes the purpose of OSGi.
- If you ever wanted to package your application as logically independent JAR files and also wanted to declare which code is accessible from each JAR file and have this visibility enforced. OSGi enables a new level of code visibility for JAR files that allows you to specify what is and what is not visible externally.
- If you ever wanted to define an extensibility mechanism for your application, like a plugin mechanism. OSGi modularity is particularly suited to providing a powerful extensibility mechanism, including support for run-time dynamism.

As you can see, these scenarios cover a lot of use cases, but are by no means exhaustive. The simple and non-intrusive nature of OSGi tends to make you discover more ways to apply it the more you use it. Having explored some of the limitations of the standard Java class path we'll now properly introduce you to OSGi.

1.2 A quick OSGi overview

The OSGi Service Platform is composed of two parts: the OSGi framework and OSGi standard services (depicted in Figure 1.1.3). The framework is the runtime that implements and provides OSGi functionality. The standard services define reusable APIs for common tasks, such as Logging and Preferences.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

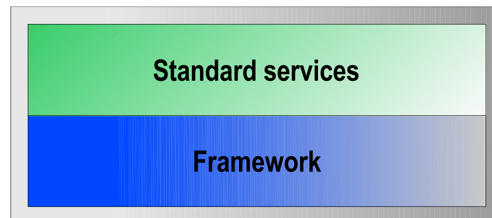


Figure 1.1.3 The OSGi Service Platform specification is divided into two halves, one for the OSGi framework and one for standard services

The OSGi specifications for the framework and standard services are managed by the OSGi Alliance (http://www.osgi.org). The alliance is an industry backed non-profit corporation founded in March 1999. The framework specification is now on it's fourth major revision and is stable. Technology based on this specification is in use in a range of large scale industry applications including (but not limited to) automotive, mobile devices, desktop applications, and more recently enterprise application servers.

NOTE

Once upon a time, the letters "OSGi" were an acronym that stood for the Open Services Gateway Initiative. This acronym highlights the lineage of the technology, but has fallen out of favor. After the third specification release, the OSGi Alliance officially dropped the acronym and "OSGi" is now simply a trademark for the technology.

In the bulk of this book we will discuss the OSGi framework, its capabilities, and how to leverage these capabilities. Since there are so many standard services, we will only discuss the most relevant and useful services where appropriate. For any service we miss, you can get more information from the OSGi specifications themselves. Let's continue our overview of OSGi by introducing the broad features of the OSGi framework.

1.2.1 The OSGi Framework

The OSGi framework plays a central role when creating OSGi-based applications, since it is the application's execution environment. The OSGi Alliance's framework specification defines the proper behavior of the framework, which gives you a well-defined API to program against. The specification also enables the creation of multiple implementations of the core framework to give you some freedom of choice; there are a handful of well-known open source projects, such as Apache Felix, Eclipse Equinox, and Knopflerfish. This ultimately benefits you, since you are not tied to a particular vendor and are able to program against the behavior defined in the specification. It's sort of like the reassuring feeling you get by knowing you can go into any McDonald's anywhere in the world and get the same meal.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The OSGi specification conceptually divides the framework into three layers (see Figure 1.1.4):

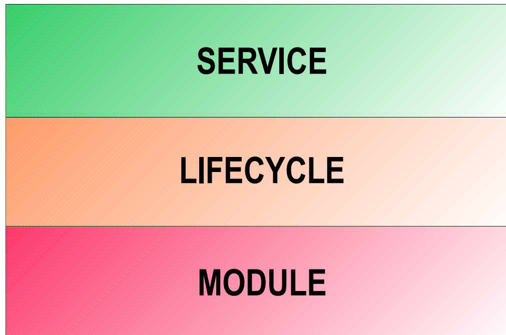


Figure 1.1.4 OSGi layered architecture

- Module layer – this layer is concerned with packaging and sharing code.
- Lifecycle layer – this layer is concerned with providing run-time module management and access to the underlying OSGi framework.
- Service layer – this layer is concerned with interaction and communication among modules, specifically the components contained in them.

Like typical layered architectures, each layer is dependent upon the layers beneath it. Therefore, it is possible for you to use lower OSGi layers without using upper ones, but not vice versa. The next three chapters discuss these layers in detail, but we will give an overview of each here.

1.2.2 Module layer

The module layer defines the OSGi module concept, called a bundle, which is simply a JAR file with extra metadata (i.e., data about data) as depicted in Figure 1.1.5. A bundle contains your class files and their related resources. Bundles are typically not an entire application packaged into a single JAR file; rather, they are the logical modules that combine to form a given application. Bundles are more powerful than standard JAR files, since you are able to explicitly declare which contained packages are externally visible (i.e., exported packages). In this sense, bundles extend the normal access modifiers (i.e., `public`, `private`, and `protected`) associated with the Java language.

Another important advantage of bundles over standard JAR files is that you are also able to explicitly declare on which external packages your bundle depends (i.e., imported packages). The main benefit of explicitly declaring your bundles' exported and imported packages is that the OSGi framework can manage and verify the consistency of your bundles automatically; this process is called bundle resolution and involves matching exported packages to imported packages. Bundle resolution ensures consistency among bundles with respect to versions and other constraints, which we will discuss in detail in Chapter 2.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

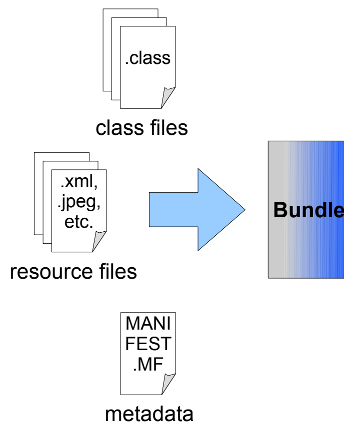


Figure 1.1.5 A bundle contains code, resources, and metadata

1.2.3 Lifecycle layer

The lifecycle layer defines how bundles are dynamically installed and managed in the OSGi framework. If we were building a house, the module layer provides the foundation and structure, while the lifecycle layer is the electrical wiring – it makes everything go.

The lifecycle layer serves two different purposes. External to your application, the lifecycle layer precisely defines the bundle lifecycle operations (e.g., install, update, start, stop, and uninstall). These lifecycle operations allow you to dynamically administer, manage, and evolve your application in a well-defined way. This means that bundles can be safely added and removed from the framework without restarting the application process. Internal to your application, the lifecycle layer defines how your bundles gain access to their execution context, which provides them with a way to interact with the OSGi framework and the facilities it provides during execution. This overall approach to the lifecycle layer is

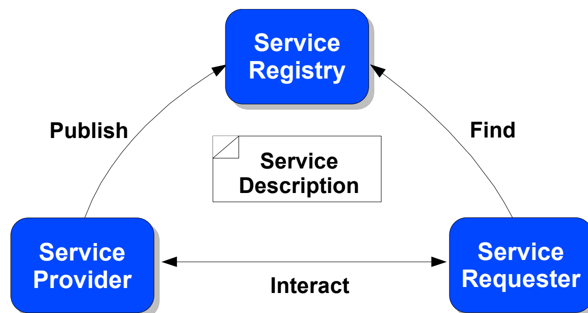


Figure 1.1.6 The service-oriented interaction pattern. Providers publish services into a registry where requesters can discover which services are available for use.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

powerful since it allows you to create externally (and remotely) managed applications or completely self-managed applications (or any combination of the two).

1.2.4 Service layer

Finally, the service layer supports and promotes a flexible application programming model that incorporates concepts popularized by service-oriented computing (although, these concepts were part of the OSGi framework before SOA became popular). The main concepts revolve around the service-oriented publish, find, and bind interaction pattern: service providers publish their services into a service registry, while service clients search the registry to find available services to use (see Figure 1.1.6). Nowadays, SOA is largely associated with web services, but OSGi services are local to a single VM, which is why some people refer to it as "SOA in a VM".

The OSGi service layer is very intuitive, since it promotes an interface-based development approach, which is generally considered good practice. Specifically, it promotes the separation of interface and implementation. OSGi services are simply Java interfaces that represent a conceptual contract between service providers and service clients. This makes the service layer very lightweight, since service providers are simply Java objects accessed via direct method invocation. Additionally, the service layer expands the bundle-based dynamism of the lifecycle layer with service-based dynamism, i.e., services can appear or disappear at any time. The result is a programming model that eschews the monolithic and brittle approaches of the past, in favor of being modular and flexible.

Certainly, this sounds all well and good, but you might still be wondering how these three layers all fit together and how you go about using them to create an application on top of them. Fair enough, in the next couple of sections we'll explore how these layers fit together using some small example programs.

1.2.5 Putting it all together

The OSGi framework is made up of layers, but how do we use these layers in application development? Let's try to make it a little clearer by outlining the general approach you will use when creating an OSGi-based application:

3. Design your application by breaking it down into service interfaces (i.e., normal interface-based programming) and clients of those interfaces.
4. Implement your service provider and client components using your preferred tools and practices.
5. Package your service provider and client components into [usually] separate JAR files, augmenting each JAR file with the appropriate OSGi metadata.
6. Start the OSGi framework.
7. Install and start all of your component JAR files from step 3.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

If you are already following an interface-based approach, then the OSGi approach will feel very familiar to you. The main difference will be how you locate your interface implementations (i.e., your services). Normally, you might instantiate implementations and pass around references to initialize clients. In the OSGi world, your services will publish themselves in the service registry and your clients will look up available services in the registry. Once your bundles are installed and started, your application will start and execute like normal, but with several advantages. Underneath, the OSGi framework is providing more rigid modularity and consistency checking and its dynamic nature opens up a whole world of possibilities.

Don't fret if you don't or can't use an interfaced-based approach for your development. The first two layers of the OSGi framework still provide a lot of functionality for you; in truth, the bulk of OSGi framework functionality lies in these first two layers, so keep reading. Enough talk, let's look at some code.

1.3 "Hello, world!" examples

Since OSGi functionality is divided over the three layers mentioned previously (modularity, lifecycle, and service), we will show you three different "Hello, world!" examples that illustrate each of these layers.

1.3.1 Modularity layer

The modularity layer is actually not related to code creation as such; rather, it is related to the packaging of your code into bundles. There are certain code-related issues of which you need to be aware when developing, but by and large you prepare your code for the modularity layer by adding packaging metadata to your project's generated JAR files. For example, suppose you want to share the class in Listing 1.1.2.

Listing 1.1.2 Basic greeting implementation

```
package org.foo.hello;

public class Greeting {
  final String m_name;

  public Greeting(String name) {
    m_name = name;
  }

  public void sayHello() {
    System.out.println("Hello, " + m_name + "!");
  }
}
```

During the build process you would compile the source code and put the generated class file into a JAR file. To leverage the OSGi modularity layer you must add some metadata into your JAR file's `META-INF/MANIFEST.MF` file, such as the following snippet:

```
Bundle-ManifestVersion: 2                                              #A
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Bundle-Name: Greeting API                                              #B
Bundle-SymbolicName: org.foo.hello                                     #C
Bundle-Version: 1.0                                                    #C
Export-Package: org.foo.hello;version="1.0"                            #D
```

#A Indicates the OSGi metadata syntax version
#B Human-readable name, not strictly necessary
#C Symbolic name and version bundle identifier
#D Share packages with other bundles

In this small example, the bulk of the metadata is largely related to bundle identification. The important part is the `Export-Package` statement, since this extends the functionality of a typical JAR file with the ability for you to explicitly declare which packages contained in the JAR are visible to the users of it. In this particular example, only the contents of the `org.foo.hello` package are externally visible; if there were other packages in our example, they would not be externally visible. So what does this really mean? It means when you run your application, other modules will not be able to accidentally (or intentionally) depend on packages that your module doesn't explicit expose.

To use this shared code in another module you would again add metadata, this time using the `Import-Package` statement to explicitly declare which external packages are required by the code contained in the client JAR. The following snippet illustrates this:

```
Bundle-ManifestVersion: 2                                              #A
Bundle-Name: Greeting Client                                           #B
Bundle-SymbolicName: org.foo.hello.client                              #C
Bundle-Version: 1.0                                                    #C
Import-Package: org.foo.hello;version="[1.0,2.0)"                      #D
```

#A Indicates the OSGi metadata syntax version
#B Human-readable name, not strictly necessary
#C Symbolic name and version bundle identifier
#D Specify dependency on an external package

To see this example in action, go into the `greeting-example/modularity/` directory for Chapter 1 in the accompanying code and type "`ant`" to build it and "`java -jar main.jar`" to run it. Although this example is simple, it illustrates that creating OSGi bundles out of your existing JAR files is a reasonably non-intrusive process. In addition, there are tools that can help you create your bundle metadata, which we will discuss in [ref xx], but in reality no special tools are required to create a bundle other than what you normally use to create a JAR file. Chapter [ref ch2] will go into all of the juicy details of OSGi modularity and how to take advantage of it in your applications.

1.3.2 Lifecycle layer

In the last subsection we saw that it is possible to leverage OSGi functionality in a non-invasive way by simply adding metadata to your existing JAR files. Such a simple approach is sufficient for most reusable libraries, but sometimes we need or want to go further to meet specific requirements or to use additional OSGi features. The lifecycle layer moves us deeper into the OSGi world.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Perhaps you want to create a module that performs some initialization task, such as starting a background thread or initializing a driver; the lifecycle layer makes this possible. Bundles may declare a given class as an "activator," which is the bundle's hook into its own lifecycle management. We will discuss the full lifecycle of a bundle later in chapter [ref ch3], but first let's look at a simple example to give you an idea of what we are talking about. In Listing 1.1.3, we extend our previous `Greeting` class to provide a singleton instance.

Listing 1.1.3 Extended greeting implementation

```
package org.foo.hello;

public class Greeting {
  static Greeting instance;                                             #A
  final String m_name;

  Greeting(String name) {                                               #B
    m_name = name;
  }

  public static Greeting get() {                                        #C
    return instance;
  }

  public void sayHello() {
    System.out.println("Hello, " + m_name + "!");
  }
}
```

#A singleton instance to be managed
#B constructor is now package private
#C clients can only use the singleton

Listing 1.1.4 implements a bundle activator that initializes the `Greeting` class singleton when the bundle is started and clears it when it is stopped. The client can now use the pre-configured singleton instead of creating its own instance.

Listing 1.1.4 OSGi bundle activator

```
package org.foo.hello;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {                     #A

  public void start(BundleContext ctx) {                                #B
    Greeting.instance = new Greeting("lifecycle");
  }

  public void stop(BundleContext ctx) {                                 #C
    Greeting.instance = null;
  }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

You can see at #A that a bundle activator must implement a simple OSGi interface, which comprises the two methods depicted at #B and #C. At execution time, the framework will construct an instance of this class and invoke the `start()` method when the bundle is started and the `stop()` method when the bundle is stopped. What we precisely mean by "starting" or "stopping" a bundle will become clearer in chapter [ref ch3]. Because the framework uses the same activator instance while the bundle is active, you can share member variables between the `start()` and `stop()` methods.

The inquisitive among you might be wondering what the single parameter of type `BundleContext` in the `start()` and `stop()` methods is all about. This is how the bundle gets access to the OSGi framework in which it is executing. From this context object, the module has access to all of the OSGi functionality for modularity, lifecycle, and services. In short, it is a fairly important object for most bundles, but we will defer a detailed introduction of it until later when we discuss the lifecycle layer. The important point to take away from this example is that bundles have a simple way to hook into their overall lifecycle and gain access to the underlying OSGi framework.

Of course, it is not sufficient to just create this bundle activator implementation, you actually have to tell the framework about it. Luckily, this is quite simple. If you have an existing JAR file you are converting to be a module, then you must add the activator implementation to the existing project so the class is included in the resulting JAR file. If you are creating a bundle from scratch, then you just need to compile the class and put the result in a JAR file. You also need to tell the OSGi framework about the bundle activator by adding another piece of metadata to the JAR file manifest. For our example, we would add the following metadata to the JAR manifest:

```
Bundle-Activator: org.foo.hello.Activator
Import-Package: org.osgi.framework
```

Notice we also need to import the `org.osgi.framework` package, since our bundle activator has a dependency on it. Otherwise, it is pretty simple to make bundles lifecycle aware. To see this example in action, go into the `greeting-example/lifecycle/` directory for Chapter 1 in the accompanying code and type "`ant`" to build it and "`java -jar main.jar`" to run it.

We've now introduced how to create OSGi bundles out of your existing JAR files using the modularity layer and how to make your bundles lifecycle aware so that they can leverage framework functionality. The last example in this section demonstrates the service-oriented application programming approach promoted by OSGi.

1.3.3 Service layer

If you follow an interfaced-based approach in your development, the OSGi service approach will feel quite natural to you. To illustrate, consider the `Greeting` interface depicted below:

```
package org.foo.hello;

public interface Greeting {
  void sayHello();
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
}
```

For any given implementation of the `Greeting` interface, when the `sayHello()` method is invoked a greeting will be displayed. In general, a service represents a contract between a provider and prospective clients; the semantics of the contract are typically described in a separate, human readable document, like a specification. The service interface above represents the syntactic contract of all `Greeting` implementations. The notion of a contract is necessary so that clients can be assured of getting the functionality they expect when using a `Greeting` service. The precise details of how any given `Greeting` implementation performs its task is not known to the client. For example, one implementation may print its greeting textually, while another may display its greeting in a GUI dialog box. The code in Listing 1.1.5 depicts a simple text-based implementation.

Listing 1.1.5 Greeting implementation

```
package org.foo.hello.impl;

import org.foo.hello.Greeting;

public class GreetingImpl implements Greeting {
  final String m_name;

  GreetingImpl(String name) {
    m_name = name;
  }

  public void sayHello() {
    System.out.println("Hello, " + m_name + "!");
  }
}
```

Your might be thinking that nothing in the service interface or Listing 1.1.5 indicate we are defining an OSGi service. Well, you'd be correct. That's what makes the OSGi's service approach so natural if you are already following an interface-based approach, since your code will largely stay the same. There are two places where your development will be a little different. One is how you make a service instance available to the rest of your application and the other is how the rest of your application discovers the available service.

All service implementations will ultimately be packaged into a bundle and that bundle will need to be lifecycle aware in order to register the service; this means that we need to create a bundle activator for our example service as depicted in Listing 1.1.6.

Listing 1.1.6 OSGi bundle activator with service registration

```
package org.foo.hello.impl;

import org.foo.hello.Greeting;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
  public void start(BundleContext ctx) {                                  #A
    ctx.registerService(Greeting.class.getName(),
        new GreetingImpl("service"), null);
  }

  public void stop(BundleContext ctx) {}                                  #B
}
```

This time in the `start()` method at #A, instead of storing the `Greeting` implementation as a singleton, we use the provided bundle context to register it as a service with the service registry. The first parameter we need to provide is the interface name(s) that the service implements, followed by the actual service instance, and finally the service properties. In the `stop()` method at #B we could unregister the service implementation before stopping the bundle, but in practice you don't need to do this. The OSGi framework will automatically unregister any registered services when a bundle stops.

We've seen how to register a service, but what about discovering a service? Listing 1.1.7 shows a very simplistic client, which does not handle missing services and suffers from potential race conditions. A more robust way to access services will be discussed in chapter [ref ch4].

Listing 1.1.7 OSGi bundle activator with service discovery

```
package org.foo.hello.client;

import org.foo.hello.Greeting;
import org.osgi.framework.*;

public class Client implements BundleActivator {

  public void start(BundleContext ctx) {
    ServiceReference ref =
        ctx.getServiceReference(Greeting.class.getName());              #A

    ((Greeting) ctx.getService(ref)).sayHello();                        #B
  }

  public void stop(BundleContext ctx) {}
}
```

The first thing you'll notice is that accessing a service in OSGi is a two-step process. First at #A, a indirect reference is retrieved from the service registry, which points to the earliest active service that was registered under the given interface name. Second at #B, this indirect reference is used to access the actual service object instance. The service reference can safely be stored in a member variable, but in general you should never store service object instances since this will make your application less dynamic and stop bundles from being cleanly uninstalled. We say "in general" since there are certain advanced use cases discussed in later chapters [ref xx] where you may wish to store a reference to a service. But

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

in these situations you must be very careful to de-reference the service when the OSGi framework tells you it is no longer valid.

Also note that the client code only needs to declare an import for the `Greeting` interface, it has no direct or explicit dependency on the actual service implementation(s). This makes it very easy to swap services dynamically without restarting the client bundle. Both the service implementation and client should be packaged into separate bundle JAR files. Each bundle will name an activator in their respective bundle metadata, but only the service implementation will export the `org.foo.hello` package, whereas the client will import it. To see this example in action, go into the `greeting-example/service/` directory for Chapter 1 in the accompanying code and type "`ant`" to build it and "`java -jar main.jar`" to run it.

Now that we have seen some examples, it is possible for us to better understand how each layer of the OSGi framework builds on the previous one. Each layer gives you additional capabilities when building your application, but OSGi technology is flexible enough for you to adopt it according to your specific needs. If you only want better modularity in your project, then use the modularity layer. If you want a way to initialize your modules and interact with the modularity layer, then use both the modularity and lifecycle layer. If you want a dynamic, interface-based development approach, then use all three layers. The choice is yours.

1.3.4 Setting the stage

To help introduce the concepts of each layer in the OSGi framework in the next three chapters, we will use a simple paint program, whose user interface is depicted in Figure 1.1.7. The paint program is not intended to be independently useful; rather, it is used to demonstrate common issues and best practices. From a functionality perspective, the paint program only allows the user to paint various shapes, such as a circles, squares, and triangles. The shapes are painted in predefined colors. Available shapes are displayed as buttons in the main window's toolbar. To draw a shape, the user selects it in the toolbar and then clicks anywhere in the canvas to draw it. The same shape can be drawn repeatedly by clicking in the canvas numerous times. The user can drag drawn shapes to reposition them. This sounds simple enough. The value of using a visual program for demonstrating these concepts will become evident when we start introducing run-time dynamism.

We have finished our overview of the OSGi framework and are ready to delve into the details, but before we do let's try to put OSGi in context by discussing similar or related technologies. While no Java technology fills the exact same niche as OSGi, there are several treading similar ground, so it is worth understanding their relevance before moving forward.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

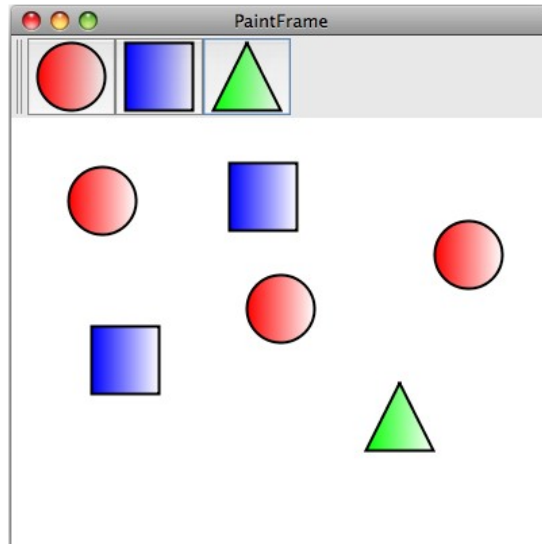


Figure 1.1.7 Simple paint program user interface

1.4 Putting OSGi in context

OSGi is often mentioned in the same breath with many other technologies, but it is actually in a fairly unique position in the Java world. Over the years, no single technology addressed OSGi's exact problem space, but there has been overlaps, complements, and offshoots. While it is not possible to cover how OSGi relates to every conceivable technology, we will try to address some of the most relevant in roughly chronological order. After reading this section you should have a good idea whether OSGi replaces your familiar technologies or is complimentary to it.

1.4.1 Java Enterprise Edition

Java Enterprise Edition (Java EE, formerly J2EE) has roots dating back to 1997. Java EE and OSGi started targeting opposite ends of the computing spectrum (i.e., enterprise vs. embedded markets, respectively). Only within the last couple years has OSGi technology really started to take root in the enterprise space. Taken in total, the Java EE API stack is not really related to OSGi. The Enterprise JavaBeans (EJB) specification is probably the closest comparable technology from the Java EE space, since it defines a component model and packaging format. However, its component model focuses on providing a standard way to implement enterprise applications that must regularly handle issues of persistence, transactions, and security. The EJB deployment descriptors and packaging formats are relatively simplistic and do not address the full component lifecycle, nor did they support clean modularity concepts. OSGi is now moving into the Java EE space to provide a more sophisticated modularity layer beneath these existing technologies. Since the two ignored

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

each other for so long, though, there are some challenges in moving existing Java EE concepts to OSGi, largely due to different assumptions about how class loading is performed. Still, progress is being made and today OSGi plays a role in several application servers, such as IBM's Websphere, Redhat's JBoss, BEA's Weblogic, Sun's GlassFish, and ObjectWeb's JOnAS.

1.4.2 Jini

An often overlooked Java technology is Jini, which is definitely a conceptual sibling of OSGi. Jini targets OSGi's original problem space of networked environments with a variety of connected devices. Sun began developing Jini in 1998. The goal of Jini is to make it possible to administer a networked environment as a flexible, dynamic group of services. Jini introduces the concepts of service providers, service consumers, and a service lookup registry. All of this sounds completely isomorphic to OSGi. Where Jini differs is its focus on distributed systems. In typical Jini fashion, consumers access clients through some form of proxy using a remote procedure call mechanism, like RMI. The service lookup registry is also a remotely accessible, federated service. The Jini model assumes remote access across multiple VM processes, while OSGi assumes everything occurs in a single VM process. However, in stark contrast to OSGi, Jini does not define any modularity mechanisms and relies on the run-time code loading features of RMI. The open source project Newton is an example of combining OSGi and Jini technologies in a single framework.

1.4.3 NetBeans

NetBeans, an IDE and run-time platform for Java, has a long history of having a very modular design. Sun purchased NetBeans back in 1999 and has continued to evolve it. The NetBeans platform actually has a lot in common with OSGi. It defines a fairly sophisticated module layer and also promotes interface-based programming using a “lookup” pattern which is quite similar to the OSGi service registry. While OSGi focused on embedded devices and dynamism, the NetBeans platform was originally just an implementation layer for the IDE. Eventually the platform was promoted as a separate tool in its own right, but focused on being a complete GUI application platform with abstractions for file systems, windowing systems, and much more. NetBeans was never really seen as being comparable to OSGi, even though it is; perhaps OSGi's more narrow focus was an asset in this case.

1.4.4 Java Management Extensions

Java Management Extensions (JMX), released in 2000 through the Java Community Process (JCP) as JSR 003, was compared to OSGi in the early days. JMX is a technology for remotely managing and monitoring applications, system objects, and devices; it defines a server and component model for this purpose. JMX is not really comparable to OSGi; it is actually complementary, since it can be used to manage and monitor an OSGi framework and its bundles and services. Why did the comparisons arise in the first place? There are probably

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

three reasons: the JMX component model was sufficiently generic so it was possible to use it for building applications, the specification defined a mechanism for dynamically loading code into its server, and certain early adopters pushed JMX in this direction. One major perpetrator was JBoss, who adopted and extended JMX for use as a modularity layer in its application server (since eliminated in JBoss 5). Nowadays, JMX is not (and shouldn't be) confused with a module system.

1.4.5 Lightweight containers

Around 2003 lightweight or inversion of control (IoC) containers started to appear, such as PicoContainer, Spring, and Apache Avalon. The main idea behind this crop of IoC containers was to simplify component configuration and assembly by eliminating the use of concrete types in favor of interfaces. This was combined with dependency injection techniques, where components depend on interface types and implementations of the interfaces are injected into the component instance. OSGi services promote a similar interface-based approach, but employs a service locator pattern to break a component's dependency on component implementations, similar to Apache Avalon. At the same time, the Service Binder project was creating a dependency injection framework for OSGi components. It is fairly easy to see why the comparisons arose. Regardless, OSGi's use of interface-based services and the service locator pattern long predated this trend and none of these technologies were offering a sophisticated dynamic module layer like OSGi. There is now significant movement from IoC vendors to port their infrastructure to the OSGi framework, such as Spring Dynamic Modules.

1.4.6 Java Business Integration

Java Business Integration (JBI) was developed in the JCP and released in 2005. Its goal was to create a standard SOA platform for creating enterprise application integration (EAI) and business-to-business integration (B2B) solutions. In JBI, “plugin” components provide and consume services once they are plugged into the JBI framework. Components do not directly interact with services, like in OSGi; instead, they communicate indirectly using normalized WSDL-based messages. JBI uses a JMX-based approach to manage component installation and lifecycle and defines a packaging format for its components. Due to the inherent similarities to OSGi's architecture, it was easy to think JBI was competing for a similar role. On the contrary, its fairly simplistic modularity mechanisms mainly addressed basic component integration into the framework. It actually made more sense for JBI to leverage OSGi's more sophisticated modularity, which is ultimately what happened in Project Fuji from Sun and ServiceMix from Apache.

1.4.7 JSR 277

In 2005, Sun announced a new JCP specification, called JSR 277, to define a module system for Java. JSR 277 was intended to define a module framework, packaging format, and repository system for the Java platform. From the perspective of the OSGi Alliance, this was

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

a major case of reinventing the wheel, since the effort was starting from scratch rather than building on the experience gained from OSGi. In 2006, many OSGi supporters pushed for the introduction of JSR 291 (titled "Dynamic Component Support for Java"), which was an effort to bring OSGi technology properly into JCP standardization. The goal was two-fold: to create a bridge between the two communities and to ensure OSGi technology integration was considered by JSR 277. The completion of JSR 291 was fairly quick since it started from the OSGi R4 specification and resulted in the R4.1 specification release. During this period OSGi technology continued to gain momentum, while JSR 277 continued to make slow progress through 2008 until it was put on hold indefinitely.

1.4.8 JSR 294

During this time in 2006, JSR 294 (titled "Improved Modularity Support in the Java Programming Language") was introduced as an offshoot of JSR 277. Its goal was to focus on necessary language changes for modularity. The original idea was to introduce the notion of a "superpackage" into the Java language, more simply described as a package of packages. The specification of superpackages got bogged down in various details until they were scrapped in favor of simply adding a "module" access modifier keyword to the language. This simplification ultimately led to JSR 294 being dropped and merged back into JSR 277 in 2007. However, when it became apparent in 2008 that JSR 277 would be put on hold, JSR 294 was pulled back out. Currently, its scope is still evolving. With JSR 277 on hold, Sun introduced an internal project, called Project Jigsaw, to modularize the JDK itself. The details of Jigsaw are also evolving, but at this point it is intended to be an implementation detail of Sun's JDK.

1.4.9 Service Component Architecture

Service Component Architecture (SCA), which started as an industry collaboration in 2004 and ultimately resulted in final specifications in 2007. SCA defines a service-oriented component model similar to OSGi's, where components provide and require services. It's component model is more advanced since it defines composite components (i.e., a component made of other components) for a fully recursive component model. SCA is intended to be a component model for declaratively composing components implemented using various technologies (e.g. Java, BPEL, EJB, C++) and integrated using various bindings (e.g. SOAP/HTTP, JMS, JCA, IIOP). SCA does define a standard packaging format, but it does not define a sophisticated modularity layer like OSGi provides. The SCA specification leaves open the possibility of other packaging formats, which makes it possible to use OSGi as a packaging and modularity layer for Java-based SCA implementations; Apache Tuscany and Newton are examples of an SCA implementation leveraging OSGi. Additionally, bundles could be used to implement SCA component types and SCA could be used as a mechanism to provide remote access to OSGi services.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

1.4.10 .NET

Although Microsoft's .NET is not a Java technology, it deserves mention since it was largely inspired by Java and did improve on it in ways which are similar to how OSGi improves Java. Released in 2002, Microsoft not only learned from Java's example, but they also learned from their own history of dealing with “DLL Hell”. As a result, .NET includes the notion of an assembly, which has modularity aspects similar to an OSGi bundle. All .NET code is packaged into an assembly, which takes the form of a DLL or EXE file. Assemblies provide an encapsulation mechanism for the code contained inside of them; an access modified, called “internal”, is used to indicate visibility within an assembly, but not external to it. Assemblies also contain metadata describing dependencies on other assemblies, but the overall model is not as flexible as OSGi's. Since dependencies are on specific assemblies, the OSGi notion of provider substitutability is not attainable.

Additionally, at run time assemblies are loaded into application domains and can only be unloaded by unloading the entire application domain. This makes the highly dynamic and lightweight nature of OSGi hard to achieve, since multiple assemblies loaded into the same application domain must be unloaded at the same time. It is possible to load assemblies into separate domains, but then communication across domains must use inter-process communication to collaborate and type sharing is greatly complicated. There have been research efforts to create OSGi-like environments for the .NET platform, but the innate differences between the .NET and Java platforms results in not a significant amount of details in common. Regardless, .NET deserves credit for improving on standard Java in this area.

1.5 Summary

The Java platform is great for developing all sorts of applications, but it does not do a good job of supporting modularity, which is critical for the long-term success of your projects. The OSGi Service Platform, through the OSGi framework, addresses the modularity shortcomings of Java to create a powerful and flexible solution.

The declarative, metadata-based approach employed by OSGi provides a non-invasive way to take advantage of its sophisticated modularity capabilities. New projects can define and enforce separation of concerns from the outset, while existing projects can leverage OSGi modularity by simply modifying how they are packaged with few, if any, changes to the code itself. With this high-level understanding of OSGi technology, we can dive into the details of the modularity layer in chapter 2, which is the foundation of everything else in the OSGi world.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

2
Mastering Modularity

In the previous chapter we took a whistle stop tour of the OSGi landscape. We made a number of observations about how standard Java is broken with respect to modularity and gave you examples where OSGi could help. We briefly covered a fair amount of computer science theory and history and potentially introduced you to a lot of new concepts, including the core layers of the OSGi framework; modules, lifecycles and services. You could be forgiven for feeling a little bit bewildered, but please, don't panic!

In this chapter we deal specifically with the modularity layer, who's features are typically the siren's song for attracting Java developers to OSGi. The modularity layer is the foundation on which everything else rests in the OSGi world. Here we will provide you with a full understanding of what OSGi modularity is, why modularity is so important in a general sense, and how it can help you in designing, building and maintaining Java applications in the future.

The goal of this chapter is to get you thinking in terms of bundles, rather than JAR files. We will teach you about OSGi bundle metadata and you will learn how to describe your application's modularity characteristics with it. To illustrate all of these concepts, we will introduce a simple paint program example, which we will convert from a monolithic application into a modular one. Let's get started with modularity.

2.1 What is modularity?

Modularity encompasses so many aspects of programming that we often take it for granted. The more experience you have with system design, the more you know good designs tend to be very modular, but what precisely does that mean? In short, it means designing a complete system from a set of logically independent pieces; these logically independent pieces are called modules. You might be thinking, “Is that it?” In the abstract, yes, that is it, but of course there are a lot of details underneath these simple concepts.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

A module defines an enforceable logical boundary, this means that code is either part of a module (i.e., on the inside) or it is not part of a module (i.e., on the outside). The internal (i.e., implementation) details of a module are only visible to code that is part of a module. For all other code, the only visible details of a module are those details that it explicitly exposes (i.e., public API), as depicted in Figure 2.1. This aspect of modules makes them an integral part of designing the logical structure of an application.

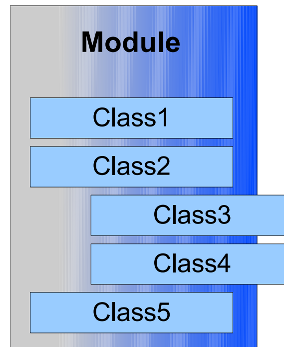


Figure 2.1 A module defines a logical boundary and the module itself is explicitly in control of which classes are completely encapsulated and which ones are exposed for external use.

2.1.1 Modularity versus object orientation

You might be wondering, "Hey, doesn't object orientation give us these things?" You would be correct, object orientation is intended to address these issues too. You will find that modularity provides many of the same benefits as object orientation. One reason these two programming concepts are similar is because both are a form of separation of concerns. The idea behind separation of concerns is you should break a system down into minimally overlapping functionality or concerns, so that each concern can be independently reasoned about, designed, implemented, and used. Modularity is actually one of the earliest forms of separation of concerns that gained popularity in the early 1970s, while object orientation gained popularity in the early 1980s.

Having said that, you might now be wondering, "If I already have object orientation in Java, why do I need modularity too?" Another good question. The need for both arises due to granularity.

Assume you need some functionality for your application. You sit down and start writing Java classes to implement the desired functionality. Do you typically implement all of your functionality in a single class? No. If the functionality is even remotely complicated, you implement it as a set of classes. You may also make use of existing classes from other parts of your project or from the JRE itself. When you are done, there is a logical relationship among the classes you created, but where is this relationship captured? Certainly it is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

captured in the low-level details of the code itself, because there will be compilation dependencies that will not be satisfied if all classes are not available at compilation time. Likewise at execution time, these dependencies will fail if all classes are not present on the class path when you try to execute your application.

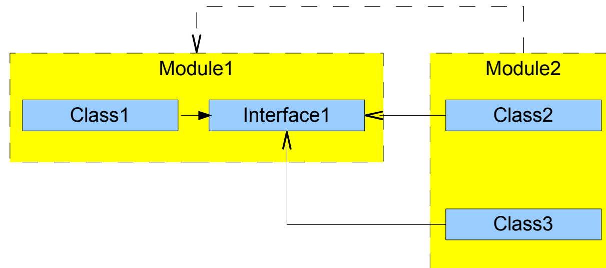


Figure 2.2 Classes have explicit dependencies due to the references contained in the code; modules have implicit dependencies due to the code they contain

Unfortunately, just looking at the external names of the classes is not sufficient to understand these dependencies and looking into the low-level code details to figure them out is quite a chore, especially if you are reusing code that someone else developed. The issue is that classes provide a mechanism to encapsulate data, but they do not provide one to encapsulate code. Modules do encapsulate code and this mechanism can be used to capture logical relationship among classes. Figure 2.2 illustrates how modules can encapsulate code and the resulting inter-module relationships. You might be thinking Java packages allow us to capture such logical code relationships. Well, you'd be right. Packages are not an object-oriented concept, but are a weak form of built-in modularity provided by Java as discussed in section 1.1.1. The key point to come away with from this discussion is that object orientation and modularity serve different, but complementary purposes (see Figure 2.3).

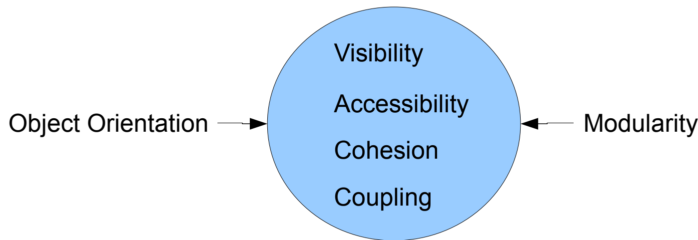


Figure 2.3 Even though object orientation and modularity provide similar capabilities, they actually address them at different levels of granularity

When developing in Java, object orientation can be viewed as the implementation approach for modules. As such, when you are developing classes you are "programming in the small," which means you are not thinking about the overall structure of your application,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

but instead are thinking in terms of specific functionality. Once you start to logically organize related classes into modules, then you start to concern yourself with "programming in the large," which means you are focusing on the larger logical pieces of your system and the relationships among these pieces.

In addition to capturing relationships among classes via module membership, modules also capture logical system structure by explicitly declaring dependencies on external code. With this in mind, we now have all the pieces in place to more concretely define what we mean by the term module in the context of this book:

MODULE

A set of logically encapsulated implementation classes, an optional public API based on a subset of the implementation classes, and a set of dependencies on external code.

Although this definition implies that modules contain classes, at this point this sense of containment is purely logical. Another aspect of modularity worth understanding is physical modularity, which refers to the actual container of module code.

Logical versus physical modularity

A module defines a logical boundary in your application, which impacts code visibility in a similar fashion as access modifiers in object-oriented programming. Logical modularity refers to code visibility. Physical modularity refers to how code is packaged and/or made available for deployment. In OSGi, these two concepts are largely conflated; a logical module is referred to as a bundle and so is the physical module (i.e., the JAR file). Even though these two concepts are nearly synonymous in OSGi, for modularity in general they are not, since it is possible to have logical modularity without physical modularity or to package multiple logical modules into a single physical module. Physical modules are sometimes also referred to as deployment modules or deployment units.

2.1.2 Driving home modularity

A traditional analogy used in discussions on separation of concerns is that of an automobile. Since we don't want to fly in the face of tradition, we'll employ it to help illustrate how these concepts fit together and why they are important.

A motor car is made up of many constituent parts, each part depends on some subset of the other parts. The motor and transmission must be designed to work well together, but the design of the motor does not generally effect the design of the doors. The chassis design is influenced by the weight of the motor and the number of doors, but does not effect what color the body is painted. Several engines may have common characteristics and be interchangeable, but this does not mean they are necessarily made out of the same parts – they may not even consume the same fuel. Let's move this analogy closer to Java

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

development by listing some basic car parts in terms of modules and classes, as shown in Table 2.1.

Table 2.1 Car component's expressed in terms of modules and classes

Modules	Classes
Chassis	Nut
Motor	Bolt
Transmission	Screw
Door	Cog
Gas Tank	Gas

Let's now consider the dependency diagram in Figure 2.4 for the modules making up the car. Each of the modules use one or more of classes for its internal functionality. The chassis, for example, uses nut, bolt, and screw classes, while the gas tank uses nuts, bolts, and gas classes.

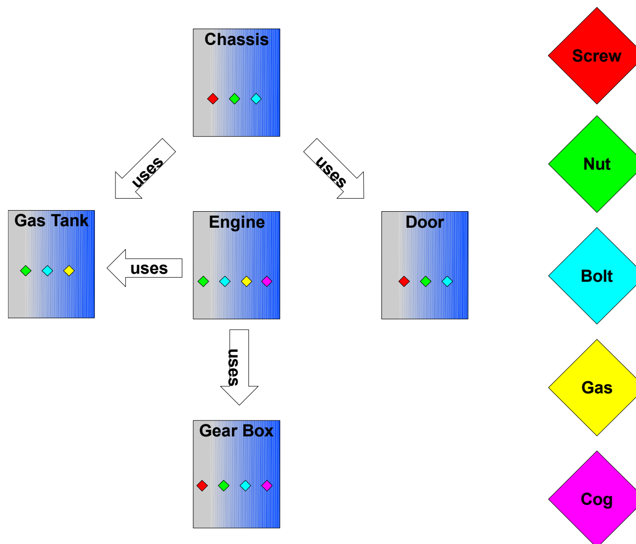


Figure 2.4 : Dependency diagram for modules and classes in an automobile

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Now consider that while the motor and the gas tank may conceptually use the same type of class, they don't necessarily use exactly the same kind of the class. Perhaps the gas tank is built using off-the-shelf nuts and bolts, while the motor uses heat-resistant nuts and bolts. In standard Java this would not be possible because whichever nut or bolt class reaches the assembly line (i.e., class path) first will be used to construct the entire car.

The Java EE developers among you will probably say, "Why don't we just use one `URLClassloader` per module? That way the motor can have it's version of it's cogs and the transmission it's version." Unfortunately, this only works if your modules are truly isolated from each other. A complex situation arises when the modules, such as the motor and the gas tank, need to share classes, in this case gas. In Java, the same byte code loaded by different class loaders it is considered a different class. This leads to extremely nasty situations where the gas in the gas tank is not the same as the gas in the motor. This tends to cause some pretty spectacular explosions (i.e., `ClassCastException`s) on our way to work.

Enterprise architects will typically tell you there are two possible solutions in such a situation. In order to get gas from the gas tank to the motor, they'll say you need to either:

- Attach a chemical processing factory (e.g., serialization via RMI, JMS, WS, etc.) to the input and outputs of these modules or
- Only ever fill the car up once with special gas provided by the manufacturer (i.e., put classes on the boot class path) and buy a whole new car when it runs out (i.e., restart the JVM for updates).

No wonder we're all driving around in odd looking applications! OSGi modularity allows us to properly express the modularity characteristics of our applications. But it is not a free pass, so let's look more in depth at why you should want to modularize your applications so you can make up your own mind.

2.2 Why modularize?

So we have talked about what modularity is, but we haven't really gone into great depth about why you might want to modularize your own applications. In fact, you might be thinking, "If modularity has been around for almost 40 years and it is so important, why isn't everyone already doing it?" Well, that's a great question and one that probably doesn't have any single answer. The computer industry is driven by "the next best thing", so we tend to throw out the old when the new comes along. And in fairness, as we discussed in the last section, our new technologies and approaches (e.g., object orientation and component orientation) do provide us with some of the same benefits that modularity was intended to address, so we might have been fooled into thinking that we no longer needed modularity itself.

Java also provides another important reason why modularity is once again an important concern. Traditionally, programming languages were the domain of logical modularity

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

mechanisms and operating systems and/or deployment packaging systems were the domain of physical modularity. Java blurs this distinction because it is a language and a platform. Thus, Java needs to provide both, but has only done a reasonably good job with the former and an inadequate job with the latter. If you compare this to the .NET platform, then you can see that Microsoft, given their history of operating system development and the pain of DLL Hell, recognized this connection early in .NET, which is why it has its own module concept, called an Assembly. Luckily for us Java programmers, OSGi technology provides a time-tested solution to the same modularity issues and much, much more.

This discussion provided some potential explanations as to why modularity is coming back in vogue, but it still doesn't answer the original question of this section, which is why should you modularize your own applications?

Modularity allows us to reason about the logical structure of our applications. Two key concepts that arose from modularity decades ago were cohesion and coupling:

- Cohesion measures how closely aligned a module's classes are with each other and with achieving the module's intended functionality. You should strive for high cohesion in your modules. For example, a module should not address many different concerns (e.g., network communication, persistence, XML parsing, etc.), it should focus on a single concern
- Coupling, on the other hand, refers to how tightly bound or dependent different modules are on each other. You want to strive for low coupling among your modules. For example, you don't want every module to depend on all other modules. As you start to use OSGi to modularize your applications, you cannot avoid these issues. Modularizing your application will help you see your application in a way that you weren't able to before

By following these principles of cohesion and coupling you will create code that is easier to reuse, since it is easier to reuse a module that performs a single function and does not have a lot of dependencies on other code.

More specifically, by using OSGi to modularize your applications, you will be able to address the Java limitations discussed in section 1.1.1. Additionally since the modules you create explicitly declare their external code dependencies, reuse is further simplified because you will no longer have to scrounge documentation or resort to trial and error to figure out what to put on the class path. You will now be developing code that more readily fits into a collaborative, independent development approach, such as in multi-team, multi-location projects or in large-scale open source projects.

Now we know what modularity is and why we want it, so let's begin to focus on how OSGi provides it and what you need to do to leverage it in your own applications. We will use an example paint program to help us understand the concepts.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

2.3 Modularizing a simple paint program

The functionality provided by OSGi's modularity layer is quite sophisticated and can seem overwhelming when taken in total. We will use a simple paint program, as discussed in chapter 1, to illustrate how to use OSGi's modularity layer. We will start from an existing paint program, rather than creating one from scratch. The existing implementation follows an interfaced-based approach with logical package structuring, so it is amenable to modularization, but it is currently packaged as a single JAR file. Listing 2.1 shows the contents of the paint program's JAR file.

Listing 2.1 Contents of existing paint program JAR file

```
META-INF/                                                      #A
META-INF/MANIFEST.MF                                           #A
org/
org/foo/
org/foo/paint/                                                 #B
org/foo/paint/PaintFrame$1$1.class                             #B
org/foo/paint/PaintFrame$1.class                               #B
org/foo/paint/PaintFrame$ShapeActionListener.class             #B
org/foo/paint/PaintFrame.class                                 #B
org/foo/paint/SimpleShape.class                                #B
org/foo/paint/ShapeComponent.class                             #B
org/foo/shape/                                                 #C
org/foo/shape/Circle.class                                     #C
org/foo/shape/circle.png                                       #C
org/foo/shape/Square.class                                     #C
org/foo/shape/square.png                                       #C
org/foo/shape/Triangle.class                                   #C
org/foo/shape/triangle.png                                     #C
```

#A Standard manifest file
#B Main application classes
#C Various shape implementations

The main classes composing the paint program and their purpose are described in table 2.2.

Table 2.2 Overview of the classes in paint program

Class	Description
`org.foo.paint.PaintFrame`	The main window of the paint program, which contains the toolbar and drawing canvas; it also has a static `main()` method to launch the program.
`org.foo.paint.SimpleShape`	An interface representing an abstract shape for painting.
`org.foo.paint.ShapeComponent`	A GUI component responsible for drawing shapes onto the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

	drawing canvas.
`org.foo.shape.Circle`	An implementation of `SimpleShape` for drawing circles.
`org.foo.shape.Square`	An implementation of `SimpleShape` for drawing squares.
`org.foo.shape.Triangle`	An implementation of `SimpleShape` for drawing triangles.

For those familiar with Swing, `PaintFrame` extends `JFrame` and contains a `JToolBar` and a `JPanel` canvas. `PaintFrame` maintains a list of available `SimpleShape` implementations, which it displays in the toolbar. When the user selects a shape in the toolbar and clicks in the canvas to draw the shape, a `ShapeComponent` (which extends `JComponent`) is added to the canvas at the location where the user clicked. A `ShapeComponent` is associated with a specific `SimpleShape` implementation by name, which it retrieves from a reference to its `PaintFrame`. The static `main()` method on `PaintFrame` launches the paint program, which creates an instance of the `PaintFrame` and each shape implementation, adding each shape instance to the created `PaintFrame` instance. Figure 2.5 captures the classes and relationships among them.

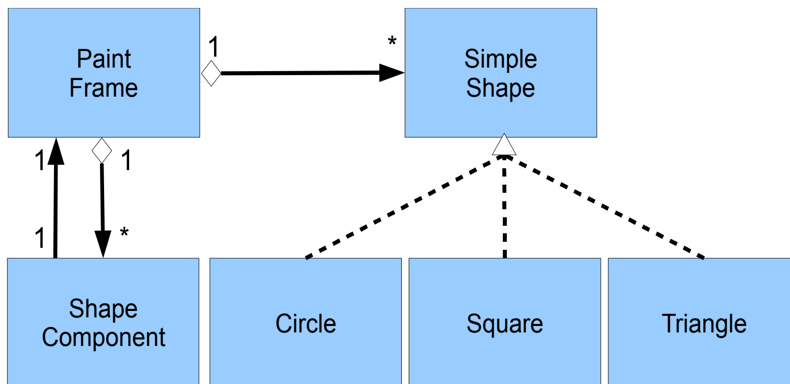


Figure 2.5 Paint program class relationships

To run this non-modular version of the paint program go into the `chapter01/paint-nonmodular/` directory of the companion code. Simply type `ant` to build the program, then type `java -jar main.jar` to run it. Feel free to click around and see how it works, but we won't go into any more details of the paint program implementation, since GUI programming is beyond the scope of this book. The important point is to understand the structure of the program and how it works. Using this understanding of the paint program, we will divide the program into bundles so we can enhance and enforce its modularity.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Currently, the paint program is packaged as a single JAR file, which we call version `1.0.0` of the program. Since everything is in a single JAR file, this implies the program is not already modularized. Of course, single JAR file applications can still be implemented in a modular way and just because an application is composed of multiple JAR files, doesn't imply it is modular. Our paint program example could have both its logical and physical modularity improved. First we will examine the logical structure of the paint program and define modules to enhance this structure. So where do we start?

One low hanging fruit is to look for public API. It is good practice in OSGi (we will see why later) to separate your public API into separate packages so they can be easily shared without worrying about exposing implementation details. Our paint program has a good example of public API, which is its `SimpleShape` interface. This interface makes it easy to implement new, possibly third-party shapes for use with our paint program. Unfortunately, `SimpleShape` is inside the same package as the implementation classes of our paint program. To remedy this situation, we will shuffle our the package structure slightly. We will move `SimpleShape` into the `org.foo.shape` package and move all shape implementations into a new package called `org.foo.shape.impl`. These changes divide our paint program into three logical pieces according to the package structure:

- `org.foo.shape` – The public API for creating shapes.
- `org.foo.shape.impl` – Various shape implementations.
- `org.foo.paint` – The application implementation.

Given this structure (logical modularity), we could package each of these packages as separate JAR files (physical modularity). To have OSGi verify and enforce our modularity, it is not sufficient to simply package our code as JAR files, we must package them as bundles. To do this, we need to understand OSGi's bundle concept, which is its logical and physical unit of modularity. Let's introduce ourselves to bundles.

2.4 Introducing bundles

If you are going to be using OSGi technology, you might as well start getting familiar with the term bundle, since you are going to be hearing and saying it a lot. The term bundle is what OSGi uses to refer its specific realization of the module concept. Throughout the remainder of this book, the terms module and bundle will be used interchangeably, but in most cases we are specifically referring to bundles and not modularity in general, unless otherwise noted. Enough fuss about how we will use the term bundle, let's define it:

BUNDLE

A physical unit of modularity in the form of a JAR file containing code, resources, and metadata, where the boundary of the JAR file also serves as the encapsulation boundary for logical modularity at execution time.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

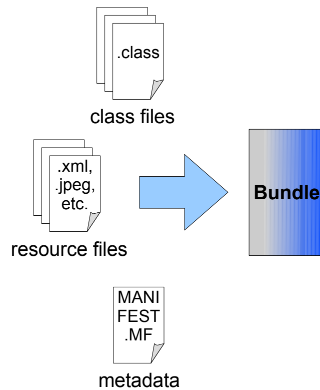


Figure 2.6 A bundle can contain all the usual artifacts you would expect in a standard JAR file, the only major difference is the manifest file contains information describing its modular characteristics

The contents of a bundle are graphically depicted in Figure 2.6. The main thing making a bundle JAR file different than a normal JAR file is it includes module metadata. All JAR files, even if they are not bundles, have a place for metadata, which is in their manifest file or, more specifically, in the `META-INF/MANIFEST.MF` entry of the JAR file. This is where OSGi places its module metadata. Whenever we refer to a bundle's manifest file, we are specifically referring to the module-related metadata in this standard JAR manifest file.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

We should also note that this definition of a bundle is very similar to our definition of a module, except that it combines both the physical and logical aspects of modularity into one concept. So, before getting into meat of this chapter, which is defining bundle metadata, let's

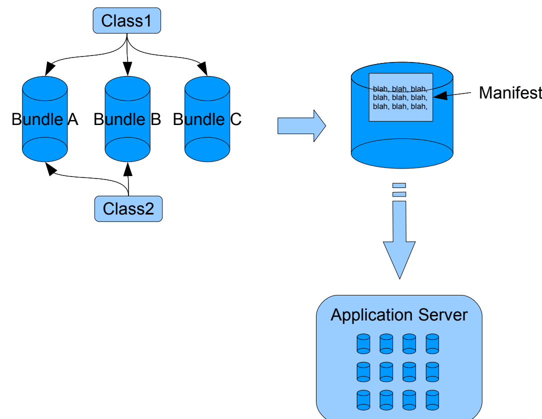


Figure 2.7 A class is a member of a bundle if it is packaged within it, the bundle carries it's module metadata inside it as part of it's manifest data and the bundle can be deployed as a unit into a runtime environment

discuss the bundle's role in physical and logical modularity in a little more detail.

2.4.1 The bundle's role in physical modularity

The main function of a bundle with respect to physical modularity is to determine module membership. There is no metadata associated with making a class a member of a bundle. A given class is a member of a bundle if it is contained in its bundle JAR file. The benefit for you is that you do not need to do anything special to make a class a member of a bundle, just put it inside the bundle JAR file.

This physical containment of classes leads to another important function of bundle JAR files as a deployment unit. The bundle JAR file is tangible and it is the artifact we share, deploy, and use when working with OSGi. The final important role of the bundle JAR file is as the container of bundle metadata, since, as we mentioned, the JAR manifest file is used to store it. These aspects of the bundle are shown in Figure 2.7. The issue of metadata placement is part of an ongoing debate, which we address in the sidebar for those interested in the issue.

Where should metadata go?

Is it a good thing to store the module metadata in the physical module and not in the classes themselves? Well, there are two schools of thought on this subject. One says it is better to include the metadata right along side the code it is describing (i.e., in the source

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

file itself), rather than in a separate file where it is more difficult to see the connection to the code. This approach is possible with various techniques, such doclets or the annotations mechanism introduced in Java 5.

Annotations in particular are the choice du jour today. Unfortunately, when OSGi work started back in 1999, annotations were not an option since they didn't exist yet. Besides, there are many other good reasons to keep the metadata in a separate file, which brings us to the second school of thought.

The second school of thought argues that it is better to not bake metadata into the source code, since it becomes harder to change. Having metadata in a separate file offers you greater flexibility. Consider the following benefits from having separate module metadata:

- You do not need to recompile your bundle to make changes to its metadata.
- You do not need access to the source code to add or modify metadata, which is sometimes necessary when dealing with legacy or third-party libraries.
- You do not need to load classes into the JVM to access the associated metadata.
- Your code does not become dependent on OSGi API and can still potentially be used with other frameworks.
- You can use your same code in multiple modules, which is sometimes convenient or even necessary in some situations when packaging your modules.
- You can easily use your code on older or smaller JVMs that do not support annotations.

So whether your preferred approach is annotations or not, you can see there is a good deal of flexibility gained by maintaining the module metadata in the manifest file.

2.4.2 The bundle's role in logical modularity

Similar to how the bundle JAR file physically encapsulates the member classes, the bundle's role in logical modularity is to logically encapsulate member classes. What precisely does this mean? It specifically relates to code visibility. Imagine you have a utility class in some `util` package, which is not part of your project's public API. To use this utility class from different packages in your project, it must be `public`. Unfortunately, this means anyone can access the utility class, even though it is not part of your public API.

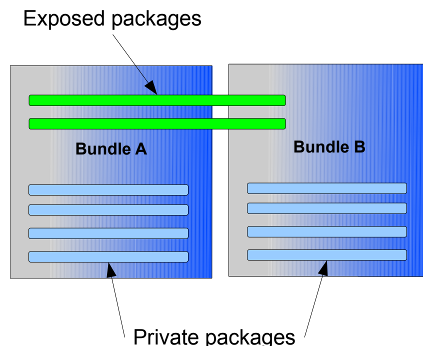


Figure 2.8 Packages (and the therefore classes within them) contained within a bundle are private to that bundle unless explicitly exposed, allowing them to be shared with other bundles

The logical boundary created by a bundle changes this, giving classes inside of the bundle different visibility rules to external code, as shown in Figure 2.8. This means `public` classes inside of your bundle JAR file are not necessarily externally visible. You might be thinking, “What?” This is not a misstatement, it is a major differentiator between bundles and standard JAR files. The only externally visibly bundle code is that which is explicitly exposed via bundle metadata. This logical boundary effectively extends standard Java access modifiers (i.e., `public`, `private`, `protected`, and package private) with module private visibility (i.e., only visible within the module). For those readers familiar with .NET, you may be thinking this is similar to the `internal` access modifier, which marks something as being visible within an Assembly, but private to other Assemblies. You'd be correct.

As you can see, the bundle concept plays important roles in both physical and logical modularity. Now we can start to examine how we use metadata to describe bundles.

2.5 Defining bundles with metadata

In this section we will discuss OSGi bundle metadata in detail and we'll use our paint program as a practical use case to help you understand the theory. The main purpose of bundle metadata is to precisely describe the modularity-related characteristics of a bundle so the OSGi framework can handle it appropriately, such as resolving dependencies and enforcing encapsulation. The module-related metadata captures the following pieces of information about the bundle:

- Human-readable information – this is optional information intended purely as an aide to humans who are using the bundle.
- Bundle identification – this is required information to identify a bundle.
- Code visibility – this is required information for defining which code is internally visible and which internal code is externally visible.

We will look at each of these areas in the following subsections. However, since OSGi relies on the manifest file, we have included a sidebar to explain its persnickety syntax details and OSGi's extended manifest value syntax. Luckily, there are tools for editing and generating bundle metadata, so we don't always have to create it manually, but it is still worthwhile to understand the syntax details.

JAR file manifest syntax

The JAR file manifest is composed of groups of name-value pairs (i.e., attributes). The general format for an attribute declaration is:

```
name: value
```

The name is not case sensitive and can contain alphanumeric, underscore, and hyphen characters. Values can contain any character information, except for carriage returns and

line feeds. The name and the value must be separated by a colon and a space. A single line cannot exceed 72 characters. If a line must exceed this length, then it is necessary to continue it on the next line, which is accomplished by starting the next line with a single space character followed by the continuation of the value. Manifest lines in OSGi can grow quite long, so it is useful to know this.

An attribute group is defined by placing attribute declarations on successive lines (i.e., one line after the other) in the manifest file. An empty or blank line between attribute declarations is used to signify different attribute groups. OSGi only uses the first group of attributes, called the main attributes. The order of attributes within a group is not important. If you look in a manifest file, you may see something like:

```
Manifest-Version: 1.0
Created-By: 1.4 (Sun Microsystems Inc.)
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.api
Bundle-Version: 1.0.0.SNAPSHOT
Bundle-Name: Simple Paint API
Export-Package: org.foo.api
Import-Package: javax.swing,org.foo.api
Bundle-License: http://www.apache.org/licenses/LICENSE-2.0
```

We will get into the exact meaning of most of these attributes throughout the remainder of this section. But for now we will focus a little more on the syntax. While the standard Java syntax is a name-value pair, OSGi defines a common structure for OSGi-specified attribute values. Most OSGi manifest attribute values are a list of clauses separated by commas, such as:

```
Property-Name: clause, clause, clause
```

Where each clause is further broken down into a target and a list of name-value pair parameters separated by semi-colons, such as:

```
Property-Name: target1; parameter1=value1; parameter2=value2,
 target2; parameter1=value1; parameter2=value2,
 target3; parameter1=value1; parameter2=value2
```

Parameters are actually divided into two types, called attributes and directives. A directive alters framework handling of the associated information and are explicitly defined by the OSGi specification. Attributes are just arbitrary name-value pairs. We will see how to use directives and attributes later. Slightly different syntax is used to differentiate directives (:=) from attributes (=), which looks something like this:

```
Property-Name: target1; dir1:=value1; attr1=value2,
 target2; dir1:=value1; attr1=value2,
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
target3; dir1:=value1; attr1=value2
```

Keep in mind you can have any number of directives and attributes assigned to each target, all with different values. Values containing whitespace or separator characters should be quoted to avoid parsing errors. Sometimes you will have lots of targets having the same set of directives and attributes. In such a situation, OSGi provides a shorthand way to avoid repeating all of the duplicated directives and attributes, as follows:

```
Property-Name: target1; target2; dir1:=value1; attr1=value2
```

This is equivalent to listing the targets separately with their own directives and attributes. This is pretty much everything you need to understand the structure of OSGi manifest attributes. Not all OSGi manifest values will conform to this common structure, but the majority do, so it makes sense for you to become familiar with it.

2.5.1 Human-readable information

Most bundle metadata is intended to be read and interpreted by the OSGi framework in its effort to provide a general modularity layer for Java. Some bundle metadata, however, serves no purpose other than helping humans understand what a bundle does and from where it comes. The OSGi specification defines several pieces of metadata for this purpose, but none of it is required, nor does it have any impact on modularity. The OSGi framework completely ignores it.

DESCRIBING OUR PAINT PROGRAM

The following code snippet depicts human-readable bundle metadata for our paint program's `org.foo.shape` bundle (the other program bundles are described similarly).

```
Bundle-Name: Simple Paint API                                          #1
Bundle-Description: Public API for a simple paint program.             #2
Bundle-DocURL: http://www.manning.com/osgi-in-action/                  #3
Bundle-Category: example, library                                      #4
Bundle-Vendor: OSGi in Action                                          #5
Bundle-ContactAddress: 1234 Main Street, USA                           #6
Bundle-Copyright: OSGi in Action                                       #7
```

The `Bundle-Name` attribute (#1) is intended to be a short name for the bundle. You are free to name your bundle anything you want. Even though it is supposed to be a short name, there is no enforcement of this; just use your best judgment. The `Bundle-Description` attribute (#2) is intended to allow you to be a little more long winded in describing the purpose of your bundle. To provide even more documentation about your bundle, the `Bundle-DocURL` at (#3) allows you to specify a URL to refer to documentation. `Bundle-Category` (#4) defines a comma-separated list of category

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

names. OSGi does not define any standard category names, so you are free to choose your own. The remaining attributes, `Bundle-Vendor` (#5), `Bundle-ContactAddress` (#6), and `Bundle-Copyright` (#7), provide information about the bundle vendor.

Human-readable metadata is reasonably straightforward. The fact that the OSGi framework ignores it means you can pretty much do what you want to with it. But don't fall into a laissez-faire approach just yet, the remaining metadata requires more precision. Next, we will look at how we use metadata to identify bundles.

2.5.2 Bundle identification

The human-readable metadata from the previous subsection helps us understand what a bundle does and from where it comes. Some of this human-readable metadata also appears to play a role in identifying a bundle. For example, `Bundle-Name` seems like it could be a form of bundle identification. It is not. The reason is somewhat historical. Earlier versions of the OSGi specification did not provide any means to uniquely identify a given bundle. `Bundle-Name` was created to be purely informational and therefore no constraints were placed on its value. As part of the OSGi R4 specification process, the idea of an unique bundle identifier was proposed. For backwards compatibility reasons, `Bundle-Name` could not be commandeered for this purpose since it would not be possible to place new constraints on it and maintain backwards compatibility. Instead, a new manifest entry was introduced, called `Bundle-SymbolicName`.

In contrast to `Bundle-Name`, which is only intended for users, `Bundle-SymbolicName` is only intended for the OSGi framework to help uniquely identify a bundle. The value of the symbolic name follows rules similar to Java package naming in that it is a series of dot-separated strings where reverse domain naming is recommended to avoid name clashes. While the dot-separated construction is enforced by the framework, there is no way to enforce the reverse domain name recommendation. You are free to choose a different scheme, but if you do, keep in mind that the main purpose is to provide unique identification, so try choose a scheme that won't lead to name clashes.

IDENTIFYING OUR PAINT PROGRAM (PART 1)

For our paint program, we divided it into bundles based on packages, so we can use each bundle's root package as our symbolic name since it already follows a reverse domain name scheme. For the public API bundle, we declare the symbolic name in manifest file as:

```
Bundle-SymbolicName: org.foo.shape
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Although it would be possible to solely use Bundle-SymbolicName to uniquely identify a bundle, it would be awkward to do so over time. Consider what would happen when you released a second version of your bundle, you'd need to change the symbolic name to keep it unique, e.g., org.foo.shapeV2. While this is possible, it is cumbersome and worse, this versioning information would be opaque to the OSGi framework, which means the modularity layer could not take advantage of it. To remedy this situation, a bundle is uniquely identified not only by its Bundle-SymbolicName, but also by its Bundle-Version, whose value conforms to the OSGi version number format (see sidebar). This pair of attributes forms a unique identifier and also allows the framework to capture the time-ordered relationship among versions of the same bundle.

IDENTIFYING OUR PAINT PROGRAM (PART 2)

For example, the following metadata uniquely identifies our paint program's public API bundle:

```
Bundle-SymbolicName: org.foo.shape
Bundle-Version: 2.0.0
```

While technically only Bundle-SymbolicName and Bundle-Version are related to bundle identification, the Bundle-ManifestVersion attribute also plays a role. Starting with the R4 specification, it became mandatory for bundles to specify Bundle-SymbolicName. This was a substantial change in philosophy. To maintain backwards compatibility with legacy bundles created before the R4 specification, OSGi introduced the Bundle-ManifestVersion attribute. Currently, the only valid value for this attribute is "2", which is the value for bundles created for the R4 specification or later. Any bundles without Bundle-ManifestVersion are not required to be uniquely identified, but bundles with it must be.

IDENTIFYING OUR PAINT PROGRAM (PART 3)

The following example illustrates the proper OSGi R4 metadata to identify our shape bundle:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.shape
Bundle-Version: 2.0.0
```

We have defined the identification metadata for our public API above. The identification metadata for the other paint program bundles would be defined in a similar fashion. Now that we have bundle identification out of the way, we are ready to look at code visibility, which is perhaps the most important area of metadata.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

OSGi version number format

One important concept you will visit over and over again in OSGi is a version number. The OSGi specification defines a common version number format used in a number of places throughout the specification. For this reason it is worthwhile spending a few paragraphs exploring exactly what a version number is in the OSGi world.

A version number is composed of three separate numerical component values separated by dots; for example, `1.0.0` is a valid OSGi version number. The first value is referred to as the major number, the second value as the minor number, and the third value as the micro number. These names reflect the relative precedence of each component value and is very similar to other version numbering schemes, where version number ordering is based on numerical comparison of version number components in decreasing order of precedence; in other words 2.0.0 is newer than 1.2.0 and 1.10.0 is newer than 1.9.9.

A fourth version component is possible, which is called a qualifier. The qualifier can contain alphanumeric characters; for example, `1.0.0.alpha` is a valid OSGi version number with qualifier. When comparing version numbers, the qualifier is compared using string comparison. As Figure 2.9 shows, this does not always lead to intuitive results; for example, while `1.0.0.beta` is newer than `1.0.0.alpha`, `1.0.0` is older than both.

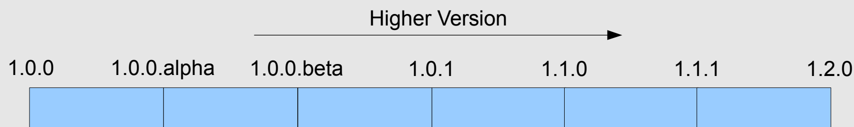


Figure 2.9 OSGi versioning semantics can sometimes lead to non intuitive results

In places where a version metadata is expected, if it is omitted, then it defaults to `0.0.0`. If a numeric component of the version number is omitted, then it defaults to `0`, while the qualifier defaults to an empty string. For example, `1.2` is equivalent to `1.2.0`. One tricky aspect is that it is not possible to have a qualifier without specifying all of the numeric components of the version. So, you cannot specify `1.2.build-59`, you must specify `1.2.0.build-59`.

OSGi uses this common version number format for versioning both bundles and Java packages. In later chapters [ref?], we will discuss more high-level approaches for managing version numbers for your packages, bundles, and applications.

2.5.3 Code visibility

Human-readable and bundle identification metadata are valuable, but they don't go very far in allowing you to describe your bundle's modularity characteristics. The OSGi specification defines metadata for comprehensively describing which code is visible internally in a bundle and which internal code is visible externally. Code visibility information is at the core of the modularity mechanisms of the OSGi framework.

Metadata for code visibility captures the following information:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

1. Internal bundle class path – this is the code forming the bundle.
2. Exported internal code – this is explicitly exposed code from the bundle class path for sharing with other bundles.
3. Imported external code – this is external code on which bundle class path code depends.

Each of these areas captures separate, but related information about which Java classes are reachable in your bundle and by your bundle. We will cover each in detail, but before we do that, let's step back and dissect how we use JAR files and the Java class path in traditional Java programming. This will give us a basis for comparison to OSGi's approach to code visibility.

Code visibility in standard JAR files and the class path

Generally speaking, we compile Java source files into classes, then use the `jar` tool to create a JAR file of our generated classes. If the JAR file has a `Main-Class` attribute in the manifest file, then we can run our application like this:

```
java -jar app.jar
```

If not, we add it to the class path and start the application something like this:

```
java -cp app.jar org.foo.Main
```

Figure 2.10 shows us the stages the JVM goes through, first it searches for the class specified in `Main-Class` attribute or the one specified on the command line. If it finds the class, it searches it for a "`static public void main(String[])`" method. If such a method is found, it invokes it to start the application. As the application executes, any additional classes needed by the application are found by searching the class path, which is composed of the application classes in the JAR file and the standard JRE classes (and anything you may have added to the class path). Classes are loaded as they are needed.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

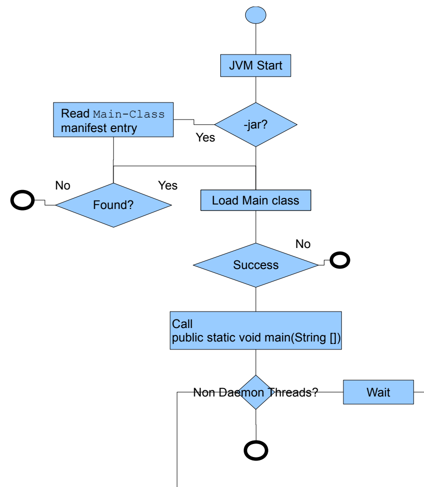


Figure 2.10 Flow diagram showing the steps the jvm goes through to execute a Java program from the classpath.

This represents a high-level understanding of how Java executes an application from a JAR file. But this high-level view conceals a few implicit decisions made by standard JAR file handling, such as:

1. Where to search inside the JAR file for a requested class?
2. Which internal classes should be externally exposed?

With respect to the first decision, JAR files have an implicit policy of searching all directories relative to the root of the JAR file as if they were package names corresponding to the requested class (e.g., the class `org.foo.Bar` is in `org/foo/Bar.class` inside the JAR file). With respect to the second decision, JAR files have an implicit policy of exposing all classes in root-relative packages to all requesters. This is a highly deconstructed view of the behavior of JAR files, but it helps to illustrate the implicit modularity decisions of standard JAR files. These implicit code visibility decisions are put into effect when we place a JAR file on the class path for execution.

While executing, the JVM finds all needed classes by searching the class path as shown in Figure 2.11. But what is exact purpose of the class path with respect to modularity? The class path defines which external classes are visible to our JAR file's internal classes. In fact every class reachable on the class path is visible to our application classes, even if they are not needed.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

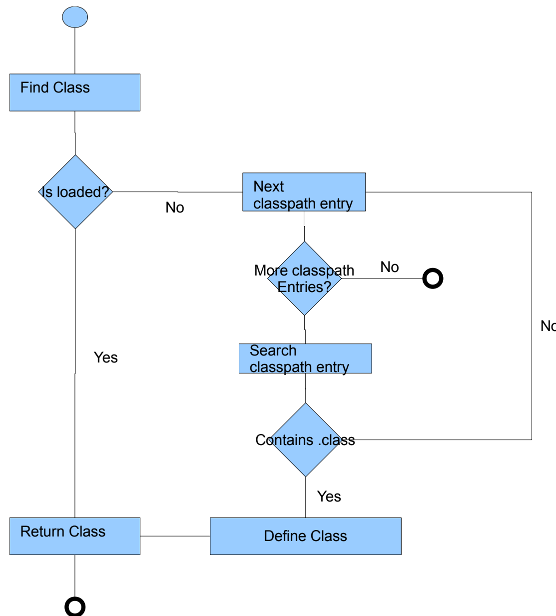


Figure 2.11 Flow diagram showing the steps the jvm goes through to load a class from the classpath

With this view of how standard JAR files and the class path mechanism work, let's look into the details of how OSGi handles these same code visibility concepts, which is quite a bit different. We will start with how OSGi searches bundles internally for code, followed by how OSGi externally exposes internal code, and finally how external code is made visible to internal bundle code. Let's get started.

Internal bundle class path

Recall the implicit internal class searching rule of standard JAR files, which is to search all directories from the root of the JAR file as if they were package names. OSGi uses a more explicit approach to define how bundle JAR files are searched, called the bundle class path. Like the standard Java class path concept, the bundle class path is a list of locations to search for classes. The difference is the bundle class path refers to locations *inside* the bundle JAR file.

When a given bundle class needs another class in the same bundle, the entire bundle class path of the containing bundle is searched to find the class. Classes in the same bundle have access to all code reachable on their bundle class path. Let's examine the syntax for declaring it.

BUNDLE-CLASSPATH

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

An ordered, comma-separated list of relative bundle JAR file locations to be searched for class and resource requests.

Bundles declare their internal class path using the `Bundle-ClassPath` manifest header. The bundle class path behaves in the same way as the global class path in terms of the search algorithm, so we can refer to Figure 2.11 to see how this behaves, but in this case the scope is limited to classes contained in the bundle. With `Bundle-ClassPath`, we can specify a delimited list of paths within the bundle where the class loader should look for classes or resources, for example:

```
Bundle-ClassPath: .,other-classes/,embedded.jar
```

This tells the OSGi framework where to search inside the bundle for classes. The "." signifies the bundle JAR file itself. For this example, the bundle itself will be searched first for root-relative packages, then in the folder called `other-classes`, and finally in the `embedded jar` within our bundle. The ordering is important, since the bundle class path entries are searched in the declared order.

The `Bundle-ClassPath` is somewhat unique, since OSGi manifest headers do not normally have default values. If you don't specify a value, the framework supplies a default value of ".". Why does `Bundle-ClassPath` have a default value? The answer is related to how standard JAR files are searched for classes. The bundle class path value of "." corresponds to this internal search policy of standard JAR files. Putting "." on your bundle class path likewise treats all root-relative directories as if they were packages when searching for classes. Making "." the default, gives both standard and bundle JAR files the same default internal search policy.

NOTE

It is important to understand that the default value of `Bundle-ClassPath` is "." if and only if there is no specified value, which is not the same as saying the value "." is included on the bundle class path by default. In other words, if you specify a value for `Bundle-ClassPath`, then "." is only included if you explicitly specify it in your comma-separated list of locations. If you specify a value and do not include ".", then root-relative directories will not be searched when looking for classes in the bundle JAR file.

As you can see, the internal bundle class path concept is actually quite powerful and flexible when it comes to defining the contents and internal search order of bundles; refer to the sidebar for some examples of when this flexibility is useful. Next, we will learn how we can expose internal code for sharing with other bundles.

Bundle class path flexibility

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

You may be wondering why you'd want to package classes in different directories or embed JAR files within the bundle JAR file? Good question!

First, the bundle class path mechanism does not apply only to classes, but also to resources. A common use case is to place images in an `image/` directory to make it explicit within the JAR file where certain content can be found. Also, in web applications classes are embedded within the JAR file under `WEB-INF/lib/` or `WEB-INF/classes/` directories.

In other situations, you may have a legacy or proprietary JAR file you are not able to change. By embedding the JAR file into your bundle and adding bundle metadata you can use it without changing the original JAR. It may also just be convenient to embed a JAR file when you want your bundle to have a private copy of some library, this is especially useful when you want to avoid sharing static library members with other bundles.

Embedding JAR files is not strictly necessary, since you can also unpack a standard JAR file into your bundle to achieve the same effect. As an aside, you could also see a performance improvement by not embedding JAR files, since OSGi framework implementations must extract the embedded JAR files to access them.

EXPORTING INTERNAL CODE

`Bundle-ClassPath` affects the visibility of classes within a bundle, but how do we share classes between bundles? The first stage is to export the packages you wish to share with others.

Externally useful classes are those composing the public API of the code contained in the JAR file, while non-useful classes are those forming the implementation details. Standard JAR files do not provide any mechanism to differentiate externally useful classes from non-useful ones, but OSGi does. In fact, while a standard JAR file exposes everything relative to the root by default, an OSGi bundle exposes nothing by default. A given bundle must explicitly describe in its metadata the internal classes it wishes to expose to other bundles; it does so using the `Export-Package` manifest header.

EXPORT-PACKAGE

A comma-separated list of internal bundle packages to expose for sharing with other bundles.

Instead of exposing individual classes, OSGi defines sharing among bundles at the package level. While this makes the task of exporting code a little simpler, it can still be a major undertaking for large projects, so we will discuss some tools to simplify this in section/appendix [ref]. When you include a package in an `Export-Package` declaration, every public class contained in the package is exposed to other bundles. A simple example for our paint program shape API bundle is as follows (graphically depicted in Figure 2.12):

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Export-Package: org.foo.shape
```

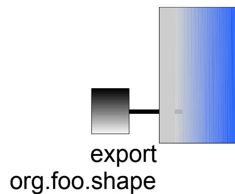


Figure 2.12 Graphical depiction of an exported package

Here we are exporting every class in the `org.foo.shape` package. You will likely want to export more than one package at a time from your bundles. We can export multiple packages by separating them with commas, such as:

```
Export-Package: org.foo.shape,org.foo.other
```

You can export as many packages as you want by separating them with commas. We can also attach attributes to exported packages. Since it is possible for different bundles to export the same packages, a given bundle can use attributes to differentiate its exports from other bundles. For example:

```
Export-Package: org.foo.shape; vendor="Manning", org.foo.other;
 vendor="Manning"
```

This attaches the `vendor` attribute with the value "`Manning`" to the exported packages. In this particular example, `vendor` is an arbitrary attribute because it has no special meaning to the framework. We defined this attribute name and value and the OSGi framework does not interpret this to mean anything other than being an attribute attached to the associated packages. When we talk about importing code, we will get a better idea of how arbitrary attributes are used in package sharing to differentiate among exported packages. As mentioned previously in the manifest syntax sidebar, OSGi also supports a shorthand format when you want to attach the same attributes to a set of target packages, like this:

```
Export-Package: org.foo.shape; org.foo.other; vendor="Manning"
```

This is equivalent to the previous example. This shorthand comes in handy, but it can only be applied if all attached attributes are the same for all packages. Using arbitrary attributes allows a bundle to differentiate its exported packages, but there is actually a more meaningful reason to use an attribute for differentiation: version management.

Code is constantly evolving. Packages contain classes that change over time. It is important to document such changes using version numbers. Version management is not a part of standard Java development, but it is inherent in OSGi-based Java development. In particular, OSGi not only supports bundle versioning as discussed previously, but it also supports package versioning, which means every shared package has a version number. Attributes are used to associate a version number with a package, such as:

```
Export-Package: org.foo.shape; org.foo.other; version="2.0.0"
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Here we attach the `version` attribute with value "`2.0.0`" to the exported packages, using OSGi's common version number format. In this case, the attribute is not arbitrary, because this attribute name and value is defined by the OSGi specification.

VERSIONING POLICY

The OSGi specification does not define a standard version policy, only a standard version format. You are free to define your own policy for the meaning of version number changes, but a common policy is that micro number changes are small changes (e.g., bug fixes) and are highly backwards compatible, while minor number changes are larger changes yet are still backwards compatible, and finally major number changes are big changes and not backwards compatible.

You may have noticed in some of our earlier `Export-Package` examples we did not specify a version at all. In that case, it defaults to "`0.0.0`", but it is not a good idea to use this version number.

With `Bundle-ClassPath` and `Export-Package`, we have a pretty good idea how we define and control the visibility of our bundle's internal classes, but not all the code we need is contained in our bundle JAR file. Next we will learn how to specify our bundle's dependencies on external code.

IMPORTING EXTERNAL CODE

Both `Bundle-ClassPath` and `Export-Package` deal with internal bundle code visibility. Normally, a bundle will also be dependent on external code. We need some way to declare which external classes are needed by our bundle so the OSGi framework can make them visible to it. Typically, the standard Java class path is used to specify which external code is visible to classes in your JAR files, but OSGi does not use this mechanism. OSGi requires all bundles to include metadata explicitly declaring their dependencies on external code, referred to as importing.

Importing external code is straightforward, if not tedious. You must declare imports for all classes required by your bundle, but not contained in your bundle. The only exception to this rule is for classes in the `java.*` packages, which are automatically made visible to all bundles by the OSGi framework.

`Import-Package` versus `import` keyword

You may be thinking you already do imports in your source code with the `import` keyword. Conceptually, the `import` keyword and declaring OSGi imports are similar, but they actually serve different purposes. The `import` keyword in Java is for namespace management; it allows you to use the short name of the imported classes, instead of using its fully qualified class name (i.e., you can refer to `SimpleShape`, rather than `org.foo.shape.SimpleShape`). You can `import` classes from any other package to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

use their short name, but it doesn't grant any visibility. In fact, you never need to use `import`, since you can just use the fully qualified class name instead. For OSGi, the metadata for importing external code is very important, since it is how the framework knows what your bundle needs.

As with exporting bundle packages, it is not necessary to explicitly declare an import for each external class your bundle uses; rather, you declare which packages it uses. The manifest header we use for importing external code is appropriately named `Import-Package`.

IMPORT-PACKAGE

A comma-separated list of packages needed by internal bundle code from other bundles.

The value of this header follows the common OSGi manifest header syntax. First, let's start with the simplest form. Consider our main paint program bundle which has a dependency on the `org.foo.shape` package. It needs to declare an import for this package as follows (graphically depicted in Figure 2.13):

```
Import-Package: org.foo.shape
```

This specifically tells the OSGi framework our bundle requires access to `org.foo.shape` in addition to the internal code visible to it from its bundle class path. We should emphasize this point. Bundle classes have access to all code either reachable from their bundle class path or explicitly imported in their metadata using `Import-Package`. Be aware that importing a package does not import its sub-packages; remember, there is no actual relationship among nested packages. If your bundle needs access to `org.foo.shape` and `org.foo.shape.other`, then it must import both packages as comma-separated targets, like this:

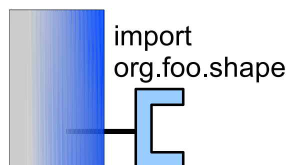


Figure 2.13 Graphical depiction of an imported package

```
Import-Package: org.foo.shape,org.foo.shape.other
```

Your bundles can import any number of packages by listing them on `Import-Package` and separating them using commas. It is actually not uncommon in larger projects for your `Import-Package` declaration to grow quite large. Remember that the JAR file manifest format is picky. If your `Import-Package` value grows too large, you will need to continue its value on the next line as discussed in section [ref].

You will likely want to narrow your bundle's package dependencies in some way. Recall how `Export-Package` declarations can include attributes to differentiate a bundle's exported packages. These export attributes can be used as matching attributes when

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

importing packages. For example, we previously discussed the following export and associated attribute:

```
Export-Package: org.foo.shape; org.foo.other; vendor="Manning"
```

A bundle with this metadata exports the two packages with the associated `vendor` attribute and value. It is possible to narrow your bundle's imported packages using the same matching attribute, such as:

```
Import-Package: org.foo.shape; vendor="Manning"
```

The bundle with this metadata is declaring a dependency on the package `org.foo.shape` with a `vendor` attribute matching the "`Manning`" value. The attributes attached to `Export-Package` declarations define the attribute's value, while attributes attached to `Import-Package` declarations define the value to match; essentially, they act like a filter. The details of how imports and exports are matched and filtered is something we will defer until section 2.7. For now it is sufficient to understand that attributes attached to imported packages are matched against the attributes attached to exported packages.

For arbitrary attributes, OSGi only supports equality matching. In other words, it either matches the specified value or it doesn't. We learned about one non-arbitrary attribute when discussing `Export-Package`, which was the `version` attribute. Since this attribute is defined by the OSGi specification, more flexible matching is supported. In fact, this is an area where OSGi excels. In the simple case, it treats a value as an infinite range starting from the specified version number; for example:

```
Import-Package: org.osgi.framework; version="1.3.0"
```

This statement declares an import for package `org.osgi.framework` for the version range of `1.3.0` to infinity, inclusive. This simple form of specifying an imported package version range implies an expectation that future versions of `org.osgi.framework` will always be backwards compatible with the lower version. In some cases, such as specification packages, it is reasonable to expect backwards compatibility. In situations where you wish to limit your assumptions about backwards compatibility, OSGi allows you to specify an explicit version range using interval notion, where the characters '`[`' and '`]`' indicate inclusive values and the characters '`(`' and '`)`' indicate exclusive values. Consider the following example:

```
Import-Package: org.osgi.framework; version="[1.3.0,2.0.0)"
```

This statement declares an import for package `org.osgi.framework` for the version range including `1.3.0` and up to, but excluding `2.0.0` and beyond. Table 2.3 illustrates the meaning of the various combinations of the version range syntax.

Table 2.3 Version range syntax and meaning

Syntax	Meaning
"[min,max)"	min <= x < max
"[min,max]"	min <= x <= max

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

"(min,max)"	min < x < max
"(min,max]"	min < x <= max
"min"	min <= x

If you want to specify a precise version range, then you must use a version range like `"[1.0.1,1.0.1]"`. You might be wondering why a single value, like `"1.0.1"`, is an infinite range rather than a precise version? Good question. The reason is partly historical. In the OSGi specifications prior to R4, all packages were assumed to be specification packages where backwards compatibility was guaranteed. Since backwards compatibility was assumed in OSGi specification prior to R4, it was only necessary to specify a minimum version. When the R4 specification added support for sharing implementation packages, it also needed to add support for arbitrary version ranges. It would have been possible at this time to redefine a single version to be a precise version, but this would be unintuitive for existing OSGi programmers. Also, the specification would have to define some syntax to represent infinity. In the end, the OSGi Alliance deemed it made the most sense to define versions ranges as presented here.

You may have noticed in some of our earlier `Import-Package` examples above we did not specify a version range at all. What happens in that case? When no version range is specified, it defaults to the value `"0.0.0"`, which you might expect from past examples. Of course, the difference here is it that the value `"0.0.0"` is interpreted as a version range from `0.0.0` to infinity.

Now we understand how we use `Import-Package` to express dependencies on external packages and `Export-Package` to expose internal packages for sharing. The decisions to use packages as the basis for inter-bundle sharing is not an obvious choice to everyone, so we discuss some arguments for doing so in the sidebar.

Depending on packages, not bundles

Importing packages seems fairly normal for most Java programmers, since we import the classes and packages we use in our source files. But the import statements in our source files are for managing namespaces not dependencies. OSGi's choice of using package-level granularity for expressing dependencies among bundles is novel, if not controversial, for Java-based module-oriented technologies. Other approaches typically adopt module-level dependencies, meaning dependencies are expressed in terms of one module depending on another. The OSGi choice of package-level dependencies has created some debate about which approach is better.

The main criticisms leveled against package-level dependencies is they are too complicated or fine grained. Some people believe it is easier for developers to think in

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

terms of requiring a whole JAR file, rather than individual packages. This argument doesn't really hold water, since a Java developer using any given technology must know something about its package naming. For example, if we know enough to realize we want to use the `Servlet` class in the first place, then we probably have some idea about which package it is in too.

Package-level dependencies are more fine-grained, which does result in more metadata. For example, if a bundle exported ten packages, then only one module-level dependency is needed to express a dependency on all of them, while package-level dependencies require ten. However, bundles rarely depend on all exported packages of a given bundle and this is more of a condemnation of tooling support. Remember how much of a nuisance it was to maintain import declarations before IDEs started doing it for us? This is starting to change for bundles too; in chapter [ref] we describe tools for generating your bundle metadata. Let's look at some of the benefits of package-level dependencies.

The difference between module- versus package-level dependencies is one of "who" versus "what". Module-level dependencies express which specific module you depend (i.e., who), while package-level dependencies express which specific packages you depend (i.e., what). Module-level dependencies are brittle, since they can only be satisfied by a specific bundle even if another bundle offers the same packages. Some people argue this is not an issue, because they want the specific bundled they have tested against or the packages are implementation packages and will not be provided by another bundle. While these arguments are reasonable, they usually break down over time.

For example, if your bundles grows too large over time you may wish to refactor it by splitting its various exported packages into multiple bundles. If you use module-level dependencies, such a refactoring will break existing clients, which tends to be a real bummer when the clients are from third parties and you cannot easily change them. This issue goes away when you use package-level dependencies. In most cases, a bundle doesn't usually depend on everything in another bundle, only a subset. Module-level dependencies are too broad and cause transitive fanout. The end result is you end up needing to deploy a lot of extra bundles you don't even use, just to satisfy all of the dependencies.

Package-level dependencies are the real dependencies of the code. It is possible to analyze a bundle's code and generate its set of imported packages; similar to how IDEs maintain import declarations in source files. Module-level dependencies cannot be discovered in such a fashion, since they do not exist in the code. Package-level dependencies sound great, right? You might now be wondering if they have any issues?

The main issue is OSGi must treat packages as an atomic unit. If this assumption were not made, then the OSGi framework would not be free to substitute a package from one bundle with the same package from another bundle. This means you cannot split a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

package across bundles; a single package must be contained within a single bundle. If packages were split across bundles, there would be no easy way for the OSGi framework to know when a package was complete. Typically, this is not such a major limitation. Other than this limitation, you can do anything with package-level dependencies you can with module-level dependencies. And truth be told, the OSGi specification does support module-level dependencies and some forms of split packages, but we will not discuss those until chapter [ref].

We've now covered the major constituents of the OSGi module layer: `Bundle-ClassPath`, `Export-Package`, and `Import-Package`. We've discussed these in the context of our paint program which we will see running in the next section, but the final piece of the puzzle we need to look at is how these various code visibility mechanims fit together in a running application.

2.5.4 Class search order

We have talked a lot about code visibility, but in the end all of the metadata we discussed allows the OSGi framework to perform sophisticated searching on the behalf of bundles for their contained and needed classes. Under the covers, when an importing bundle needs a class from an exported package, it asks the exporting bundle for it. The framework uses class loaders to do this, but the exact details of how it asks are unimportant. Still, it is important to understand the ordering of this class search process.

When a bundle needs a class at execution time, the framework searches for the class in the following order:

1. If the class is from a package starting with "`java.`", then the parent class loader is asked for the class. If the class is found, then it is used. If there is no such class, the search ends with an exception.
2. If the class is from a package imported by the bundle, then the framework asks the exporting bundle for the class. If the class is found, then it is used. If there is no such class, the search ends with an exception.
3. The bundle class path is searched for the class. If it is found, then it is used. If there is no such class, the search ends with an exception.

These steps are important since they also help the framework ensure consistency. Specifically, step (1) ensures all bundles use the same core Java classes, while step (2) ensures imported packages are not split across the exporting and importing bundles.

That's it! We've finished our introduction to bundle metadata. We have not covered everything you can possibly do, but have covered the most important bundle metadata for getting started creating bundles; we will cover additional modularity issues in chapter 5. Next we will put all of our metadata in place for our paint program, then step back a little bit and review our current design. Before moving on, if you are wondering if it is possible to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

have a JAR file that is both a bundle and an ordinary JAR file, then we discuss this in the following sidebar.

Is a bundle a JAR file or a JAR file a bundle?

Maybe you are interested in adding OSGi metadata to your existing JAR files or you want to create bundles from scratch, but you'd still like to use them in non-OSGi situations too. We've said before a bundle is just a JAR file with additional module-related metadata in its manifest file, but how accurate is this statement? Does it mean any OSGi bundle can be used as a standard JAR file? What about using a standard JAR file as a bundle? Let's answer the second question first, since it is easier.

A standard JAR file can actually be installed into an OSGi framework unchanged. Unfortunately, it doesn't really do anything useful. Why? The main reason is because a standard JAR file does not expose any of its content; in OSGi terms, it doesn't export any packages. The default `Bundle-ClassPath` for a JAR file is ".", but the default for `Export-Package` is nothing. So even though a standard JAR file is a bundle, it is not a very useful bundle. At a minimum, you need to add an `Export-Package` declaration to its manifest file to explicitly expose some (or all) of its internal content. What about bundle JAR files? Can they be used as a standard JAR file outside of an OSGi environment? The answer is, it depends.

It is possible for you to create bundles that function equally well in or out of an OSGi environment, but not all bundles can be used as standard JAR files. It all comes down to which features of OSGi your bundle uses. Of the metadata features we have learned about so far, there is only one which can cause issues: `Bundle-ClassPath`. Recall that the internal bundle class path is a comma separated list of locations inside the bundle JAR file and may contain:

- A "." representing the root of the bundle JAR file itself.
- A relative path to an embedded JAR file.
- A relative path to an embedded directory.

Only bundles with a class path entry of "." can be used as standard JAR files. Why? It is like we said in section [ref], the OSGi notion of "." on the bundle class path is equivalent to standard JAR file class searching, which is to search from the root of the JAR file as if all relative directories are package names. If a bundle specifies a relative path to an embedded JAR file or directory, then it requires special handling only available in an OSGi environment. Luckily, it is not too difficult to avoid using embedded JAR files and directories. Embedded directories are really not too common and embedded JAR files can be avoided by simply exploding them into the bundle JAR file.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

It is a good idea to try to keep your bundle JAR files compatible with standard JAR files if you can, but it is still best to use them in an OSGi environment. Without OSGi, you lose dependency checking, consistency checking, and boundary enforcement, not to mention all of the cool lifecycle and service stuff we will discuss in the coming chapters.

2.6 Finalizing our paint program design

So far, we have defined three bundles for our paint program:a shape API bundle, a shape implementation bundle, and a main paint program bundle. Let's look at the complete metadata for each. The shape API bundle is described by the following manifest metadata:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.shape
Bundle-Version: 2.0.0
Bundle-Name: Paint API
Import-Package: javax.swing
Export-Package: org.foo.shape; version="2.0.0"
```

The bundle containing the shape implementations is described by the following manifest metadata:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.shape.impl
Bundle-Version: 2.0.0
Bundle-Name: Simple Shape Implementations
Import-Package: javax.swing, org.foo.shape; version="2.0.0"
Export-Package: org.foo.shape.impl; version="2.0.0"
```

And lastly, the main paint program bundle is described by the following manifest metadata:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.paint
Bundle-Version: 2.0.0
Bundle-Name: Simple Paint Program
Import-Package: javax.swing, org.foo.shape; org.foo.shape.impl;
 version="2.0.0"
```

As you can see in Figure 2.14, these three bundles directly mirror the logical package structure of the paint program.

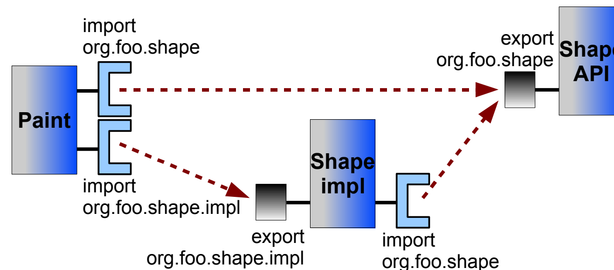


Figure 2.14 Structure of paint program's bundles

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

This approach is certainly reasonable, but could it be improved? To some degree, we can only answer this question if we know more about the intended uses of the paint program, but let's look a little more closely at it anyway.

2.6.1 Improving our paint program modularization

In our current design, one aspect that sticks out is the shape implementation bundle. Is there a downside to keeping all shape implementations in a single package and a single bundle? Perhaps it is better to reverse the question. Is there any advantage to separating the shape implementations into separate bundles? Imagine use cases where not all shapes are necessary; for example, small devices may not have enough resources to support all shape implementations. If we separate the shape implementations into separate packages and separate bundles, then it gives us more flexibility when it comes to creating different configurations of our application.

This is a good issue to keep in mind when modularizing your applications. Optional components or components with the potential to have multiple alternative implementations are good candidates to be in separate bundles. Breaking your application into multiple bundles gives you more flexibility, because you are limited to deploying configurations of your application based on the granularity of your defined bundles. Sounds good, right? You might then wonder why we don't just divide our applications into as many bundles as we can?

There is a price to pay for the flexibility afforded by dividing our application into multiple bundles. Lots of bundles mean you have lots of artifacts which are versioning independently, creating lots dependencies and configurations to manage. So, it is probably not a good idea to create a bundle out of each of your project's packages, for example. You need to analyze and understand your needs for flexibility when deciding how best to divide your application. There is no single rule for every situation.

Returning to our paint program example, let's assume our ultimate goal is to enable the possibility for creating different configurations of our application with different sets of shapes. To accomplish this, we move each shape implementation into its own package (e.g., `org.foo.shape.circle`, `org.foo.shape.square`, `org.foo.shape.triangle`). We can now bundle each of these shapes separately; the following metadata captures the circle bundle:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.shape.circle
Bundle-Version: 2.0.0
Bundle-Name: Circle Implementation
Import-Package: javax.swing, org.foo.shape; version="2.0.0"
Export-Package: org.foo.shape.circle; version="2.0.0"
```

The metadata for the square and triangle bundles is nearly identical, except with the correct shape name substituted where appropriate. The shape implementation bundles have dependencies on Swing and the public API; they export their implementation-specific shape

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

package. These changes also require changes to the metadata of the paint program implementation bundle; we modify its metadata as follows:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.paint
Bundle-Version: 2.0.0
Bundle-Name: Simple Paint Program
Import-Package: javax.swing, org.foo.shape; org.foo.shape.circle;
 org.foo.shape.square; org.foo.shape.triangle; version="2.0.0"
```

The paint program implementation bundle depends on Swing, the public API bundle, and all three shape bundles. Figure 2.15 depicts the new structure of our paint program.

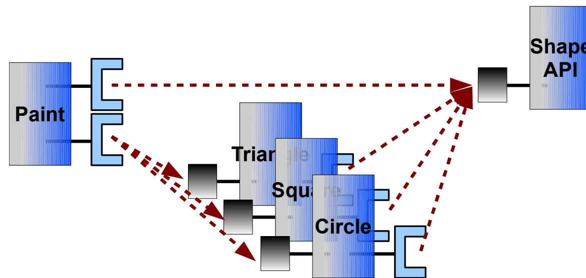


Figure 2.15 Logical structure of paint program with separate modules for each shape implementation

Now we have five bundles (shape API, circle, square, triangle, and paint). Great. But what do we do with these bundles? The initial version of the paint program had a static `main()` method on `PaintFrame` to launch it, do we still use it to launch the program? We could use it by putting all of our bundle JAR files on the class path, since all of our example bundles can function as standard JAR files, but this would defeat the purpose of modularizing the application. There would be no enforcement of modular boundaries or consistency checking. To get these benefits, we need launch our paint program using the OSGi framework. Let's look at what we need to do.

2.6.2 Launching the new paint program

The focus of this chapter is on using the modularity layer, but we cannot launch our application without a little help from the lifecycle layer. Instead of putting the cart before the horse and talking about the lifecycle layer now, we created a generic OSGi bundle launcher to launch our paint program for us. This launcher is very simple. You execute it from the command line and specify a path to a directory containing bundles, it will create an OSGi framework instance and deploy all bundles in the specified directory. The cool part is this generic launcher hides all of the details and OSGi-specific API from us; we will discuss the launcher in detail in chapter 12.

Even though the launcher is pretty cool, we still need some way to kick start the application, such as a static `main()` method. In fact, we can use the original paint program's static `main()` method to launch our new modular version. To get this to work with the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

bundle launcher, we simply need to add the following metadata from the original paint program to the paint program bundle manifest:

```
Main-Class: org.foo.paint.PaintFrame
```

Like in the original paint program, this is standard JAR file metadata for specifying the class containing the static `main()` method of the application. Note that this feature is not defined by the OSGi specification, but is a feature of our bundle launcher. To build and launch our newly modularized paint program, simply go into the `chapter02/paint-modular/` directory in the companion code and type `ant`. This will compile all code and package the modules. Typing `java -jar launcher.jar bundles/` will start the paint program.

The paint program starts up as it apparently always has, but it gains all the modularity goodness of OSGi. That's all there is to it. We have used the OSGi modularity layer to create a nicely modular application. The metadata-based approach of OSGi didn't require any code changes to our application, although we did move some classes around to different packages to improve our logical and physical modularity.

The goal of the OSGi framework is to shield us from a lot of the complexities, but sometimes it is beneficial for us to peek behind the curtain, such as to help us debug our OSGi-based applications when things go wrong. In the next section, we will look at some of the work the OSGi framework is doing for us to get a deeper understanding of how everything fits together.

2.7 OSGi dependency resolution

We learned how we describe the internal code composing our bundles with `Bundle-ClassPath`, expose internal code for sharing with `Export-Package`, and declare dependencies on external code with `Import-Package`. Although we hinted at how the OSGi framework uses the exports from one bundle to satisfy the imports of another, we didn't really go into it in detail. The `Export-Package` and `Import-Package` metadata declarations included in bundle manifests form the backbone of the OSGi bundle dependency model, which is predicated on package sharing among bundles.

In this section we will explain how OSGi resolves bundle package dependencies and ensures package consistency among bundles. After this section, you will have a clear understanding of how bundle modularity metadata is used by the OSGi framework. You might wonder why this is necessary, since it seems like OSGi framework implementation details. Admittedly, this section covers some of the more complex details of the OSGi specification, but in reality it is helpful when defining your bundle metadata if you understand a little bit of what is going on behind the scenes. Further, this information can come in handy when debugging your OSGi-based applications. Let's get started.

2.7.1 Resolving dependencies automatically

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Adding OSGi metadata to your JAR files represent extra work for you as a developer, so why do we want to do it? The main reason is so we can use the OSGi framework to support and enforce the inherent modularity of our bundles. One of the most important tasks performed by the OSGi framework is automating dependency management, which is called bundle dependency resolution. Technically, resolution is part of the lifecycle layer because it happens at execution time, but we will describe it in this chapter since it is intrinsically related to the bundle metadata.

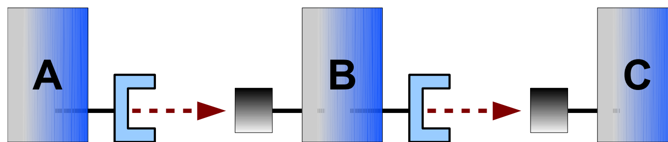


Figure 2.16 Transitive dependencies occur when bundle A depends on packages from bundle B and bundle B in turn depends on packages from bundle C. In order to use bundle A you need to find (resolve) both bundle B and bundle C.

A bundle's dependencies must be resolved by the framework before the bundle can be used, as shown in Figure 2.16. The framework's dependency resolution algorithm is very sophisticated; we will get into its gory details, but let's start off with a simple definition.

RESOLVING

The process of matching a given bundle's imported packages to exported packages from other bundles and doing so in a consistent way so any given bundle only has access to a single version of any type.

Resolving a bundle may cause other bundles to be resolved transitively, if exporting bundles themselves have not yet been resolved. The resulting set of resolved bundles are conceptually "wired" together in such a fashion that any given imported package from a bundle is wired to a matching exported package from another bundle, where a "wire" implies having access to the exported package. The final result is a graph of all bundles wired together where all imported package dependencies are satisfied. If any dependency cannot be satisfied, then the resolve fails and the instigating bundle cannot be used until its dependencies are satisfied. This description likely makes you want to ask three questions:

1. When does the framework resolve a bundle's dependencies?
2. How does the framework gain access to bundles to resolve them in the first place?
3. What does it really mean to "wire" an importing bundle to an exporting bundle?

The first two questions are somewhat related, since they both involve the lifecycle layer, which we will discuss in the next chapter. For the first question, it is sufficient to say the framework resolves a bundle automatically when another bundle attempts to use it. To

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

answer the second question, we will simply say that all bundles must be "installed" into the framework in order to be resolved (bundle installation will be covered in more depth in the next chapter). For the discussion in this section, we will always be talking about installed bundles. As for the third question, we will not answer it fully since the technical details of wiring bundles together is not really important, but for the curious we can explain it briefly, before looking into the resolution process in more detail.

At execution time, each OSGi bundle has a class loader associated with it, which is how the bundle gains access to all of the classes to which it should have access (i.e., the ones determined by the resolution process). When an importing bundle is wired to an exporting bundle, the importing class loader is given a reference to the exporting class loader so it can delegate requests for classes in the exported package to it. You don't need to worry about how this happens, just relax and let OSGi worry about it for you. Now let's look into the resolution process in more detail.

Simple cases

At first blush, resolving dependencies is fairly straightforward; the framework just needs to match exports to imports. Let's consider a snippet from our paint program example.

```
Bundle-Name: Simple Paint Program
Import-Package: org.foo.shape
```

From this we know the paint program has a single dependency on the `org.foo.shape` package. If only this bundle were installed in the framework, it would not be usable since its dependency would not be satisfiable. We have to supply the shape API in order to used the main paint program:

```
Bundle-Name: Paint API
Export-Package: org.foo.shape
```

When the framework tries to resolve the paint program, it knows it must find a matching export for `org.foo.shape`. In this case, it finds a match from shape API bundle. When the framework finds a matching candidate, it must determine if the candidate is resolved. If the candidate is already resolved, then the candidate can be chosen to satisfy the dependency. If the candidate is not yet resolved, then the framework must resolve it first before it can select it; this is the transitive nature of resolving dependencies. If our shape API bundle has no dependencies, it can always be successfully resolved. However we know from our example it does have some dependencies, namely `javax.swing`:

```
Bundle-Name: Paint API
Import-Package: javax.swing
Export-Package: org.foo.shape
```

So what happens when the framework tries to resolve the paint program? By default in OSGi it would not succeed, which means the paint program can not be used. Why? Because even though the `org.foo.shape` package from the API bundle satisfies main program's import, there is no bundle to satisfy shape API's import of `javax.swing`. In general to resolve this situation, we could install another bundle exporting the required package, such as:

```
Bundle-Name: Swing
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Export-Package: javax.swing
```

Now when the framework tries to resolve the paint program, it will succeed. The main paint program bundle's dependency is satisfied by the shape API bundle and its dependency is satisfied by the Swing bundle, which has no dependencies. After resolving the main paint program bundle, all three bundles are marked as resolved and the framework will not try to resolve them again (until certain conditions require it, which will be described in the next chapter). The framework ends up wiring the bundles together as depicted in Figure 2.17.

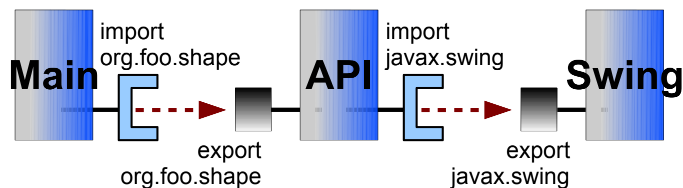


Figure 2.17 Transitive bundle resolution wiring

What does the wiring in Figure 2.17 tell us? It says when main bundle needs a class in package `org.foo.shape`, it will get it from the shape API bundle. It also says when the shape API bundle needs a class in package `javax.swing`, it will get it from the Swing bundle. Even though this example is simple, it is largely what the framework is trying to do when it resolves bundle dependencies.

System class path delegation

In actuality, the `javax.swing` case in the previous example is a little misleading if you are running your OSGi framework with a JRE that includes `javax.swing`. In such a case, you may want bundles to use Swing from the JRE. The framework can implicitly provide access using system class path delegation. We will look at this area a little bit in chapter 12, but this highlights a deficiency with the heavyweight JRE approach. If it is possible to install a bundle to satisfy the Swing dependencies why are they packaged in the JVM by default? Adoption of OSGi patterns could massively trim the footprint of future JVM implementations.

You might recall from section [ref] something about attaching attributes to exported packages. We said it was sufficient to understand that attributes attached to imported packages are matched against attributes attached to exported packages. Now we can more fully understand what this means. Let's modify our bundle metadata snippets to get a deeper understanding of how attributes factor into the resolve process. Assume we modify the swing bundle to look like this:

```
Bundle-Name: Swing
Export-Package: javax.swing; vendor="Sun"
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Here we modify Swing bundle to export `javax.swing` with an attribute `vendor` with value "`Sun`". If the other bundle's metadata are not modified and we perform the resolve process from scratch, what impact does this change have? This minor change actually has no impact at all. Everything still resolves as it did before and the `vendor` attribute never comes into play. Why not? Depending on your perspective this may or may not seem confusing. As we previously described attributes, imported attributes are matched against exported attributes. In this case, no import declarations mention the `vendor` attribute, so it is simply ignored. Let's revert the change to the Swing bundle and instead change the API bundle to look like this:

```
Bundle-Name: Paint API
Export-Package: org.foo.shape
Import-Package: javax.swing; vendor="Sun"
```

Attempting to resolve paint program bundle will now fail because no bundle is exporting the package with a matching `vendor` attribute for our API bundle. Putting the vendor attribute back on the Swing bundle export allows the main paint program bundle to successfully resolve again with the same wiring as depicted before in figure 2.17. Attributes on exported packages only have an impact if imported packages specify them, in which case the values must match or the resolve fails. You might recall we also talked about the `version` attribute in section [ref]. Other than the more expressive interval notation for specifying ranges, it works the same way as arbitrary attributes. For example, we can modify the shape API bundle as follows:

```
Bundle-Name: Paint API
Export-Package: org.foo.shape; vendor="Manning"; version="2.0.0"
Import-Package: javax.swing; vendor="Sun"
```

And modify our paint program bundle as follows:

```
Bundle-Name: Simple Paint Program
Import-Package: org.foo.shape; vendor="Manning"; version="[2.0.0,3.0.0)"
```

In this case, the framework can still resolve everything because our shape API bundle's export matches our paint program bundle's import; the `vendor` attributes match and `2.0.0` is in the range of `2.0.0` inclusive to `3.0.0` exclusive. In this particular example, we actually have multiple matching attributes on our import declaration, which is treated like a logical `AND` by the framework. Therefore, if any of the matching attributes on an import declaration do not match a given export, then the export does not match at all.

Overall, attributes do not add much complexity to the resolution process, since they simply add additional constraints to the matching of imported and exported package names already taking place. Next we will look into slightly more complicated bundle resolution scenarios.

Multiple providers

In the previous section, dependency resolution was fairly straightforward since there was only one candidate to resolve each dependency. The OSGi framework doesn't restrict multiple bundles from exporting the same package. Actually, one of the main benefits of the OSGi framework is it supports side-by-side versions, meaning it is possible to use different

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

versions of the same package within the same running JVM. In highly collaborative environments of independently developed bundles, it is difficult to limit which versions of packages are used. Likewise, in really large systems, it is possible for different teams to use different versions of libraries in their subsystems; the use of different XML parser versions is a prime example.

Let's consider what happens when there are multiple candidates available to resolve the same dependency. Considering a case in which a web application needs to import the `javax.servlet` package and both a servlet API bundle and a Tomcat bundle provide the package (see Figure 2.18).

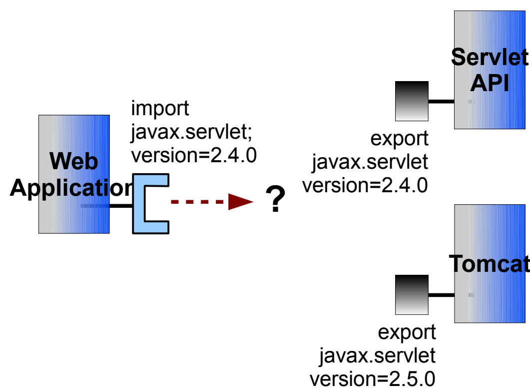


Figure 2.18 How does the framework choose between multiple exporters of a package?

When the framework tries to resolve the dependencies of the web application it sees that the web application requires `javax.servlet` with a minimum version of `2.4.0` and both the servlet API and Tomcat bundles meet this requirement. Since the web application can only be wired to one version of the package, how does the framework choose between the candidates? As you might intuitively expect, the framework favors the highest matching version, so in this case it selects Tomcat to resolve the web application's dependency. Sounds simple enough. What would happen if both bundles exported the same version, say `2.4.0`?

In this case, the framework chooses between candidates based on the order in which they are installed in the framework. Bundles installed earlier are given priority over bundles installed later; as we mentioned, the next chapter will show you what it means to install a bundle in the framework. If we assume the servlet API was installed before Tomcat, then the servlet API will be selected to resolve the web application's dependency. There is one more consideration which the framework makes when prioritizing matching candidates: maximizing collaboration.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

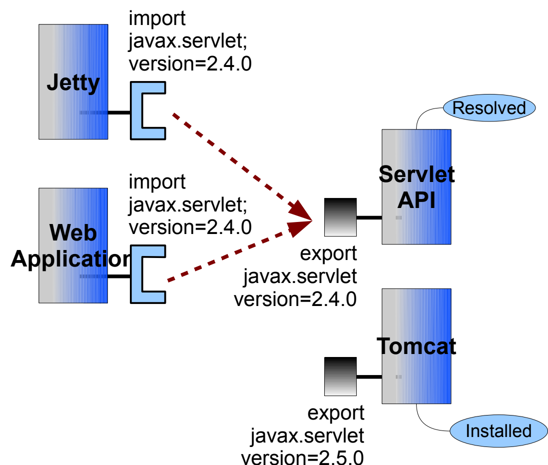


Figure 2.19 If a bundle is already resolved because it is in use by another bundle this bundle will be preferred to bundles that are only installed

So far we have been working under the assumption of starting the resolve process on a cleanly installed set of bundles. However, the OSGi framework allows bundles to be dynamically installed at any time during execution. In other words, the framework doesn't always start from a clean slate. It is possible for some bundles to be installed, resolved, and already in use when new bundles are installed. This creates another means to differentiate among exporters: already resolved exporters and not yet resolved exporters. The framework gives priority to already resolved exporters, so if it must choose between two matching candidates where one is resolved and one is not, then it chooses the resolved candidate. Consider again our example with the servlet API exporting version `2.4.0` of the `javax.servlet` package and Tomcat exporting version `2.5.0`. If the servlet API was already resolved, then the framework would choose it to resolve the web application's dependency, even though it is not exporting the highest version. Why?

It has to do with maximizing the potential for collaboration. Bundles can only collaborate if they are using the same version of a shared package. When resolving, the framework favors already resolved packages as a means to minimize the number of different versions of the same package being used. To summarize the priority of dependency resolution candidate selection:

1. Highest priority is given to already resolved candidates, where multiple matches of resolved candidates are sorted according to version and then installation order.
2. Next priority is given to unresolved candidates, where multiple matches of unresolved candidates are sorted according to version and then installation order.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

It looks like we have all our bases covered, right? Not quite. Next we will look at how an additional level of constraint checking is necessary to ensure bundle dependency resolution is consistent.

2.7.2 Ensuring consistency with "uses" constraints

From the perspective of any given bundle, there are a set of packages visible to it, which we will call its class space. Given our current understanding, we can define a bundle's class space as its imported packages combined with the packages accessible from its bundle class path as shown in figure 2.20.

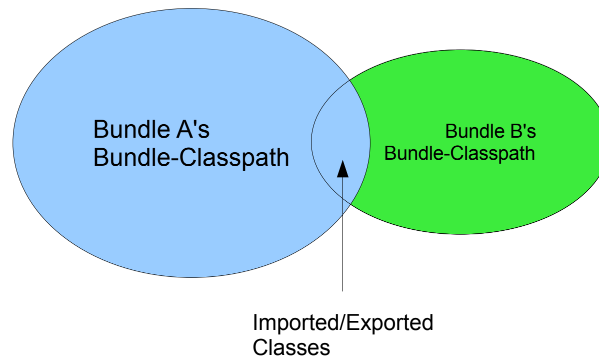


Figure 2.20 Bundle A's class space is defined as the union of it's bundle classpath with it's imported packages which are provided by bundle B's exports.

A bundle's class space must be consistent, which means only a single instance of a given package must be visible to the bundle. Here we define instances of a package as those with the same name and version, but from different providers. For example consider our previous example where both the servlet API and Tomcat bundles exported the `javax.servlet` package at version `2.4.0`. The OSGi framework strives to ensure the class spaces of all bundles remain consistent. Simply prioritizing how exported packages are selected for imported packages, as described in the last section, is not sufficient. Why not? Let's consider the simple API in the following code snippet:

```
package org.osgi.service.http;
import javax.servlet.Servlet;
public interface HttpService {
  void registerServlet(Sting alias, Servlet servlet, HttpContext ctx);
}
```

This is a snippet from an API we will meet later in chapter [ref], the details of what it does are unimportant at the moment, for now we just need to know it's method signature. Let's assume the implementation of this API is packaged as a bundle containing the `org.osgi.service.http` package, but not `javax.servlet`. This means it would have some metadata in its manifest like this:

```
Export-Package: org.osgi.service.http; version="1.0.0"
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Import-Package: javax.servlet; version="2.3.0"
```

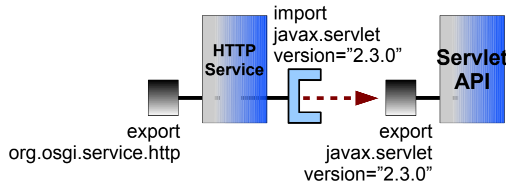


Figure 2.21 HTTP service dependency resolution

Let's assume our framework has our HTTP service bundle and a servlet library bundle installed, as depicted in Figure 2.21. Given these two bundles, the framework makes the only choice available, which is to select the version of `javax.servlet` provided by the Servlet API bundle. Now assume we install two more bundles into the framework, the Tomcat bundle exporting version `2.4.0` of `javax.servlet` and a bundle containing a client for the HTTP service importing version `2.4.0` of `javax.servlet`. When the framework resolves these two new bundles, it would do so as depicted in Figure 2.22.

The HTTP client bundle imports `org.osgi.service.http` and version `2.4.0` of `javax.servlet`, which the framework resolves to the HTTP service bundle and the Tomcat bundle, respectively. It seems everything is fine, all bundles have their dependencies resolved, right? Not quite. There is an issue with these choices for dependency resolution, can you see what it is?

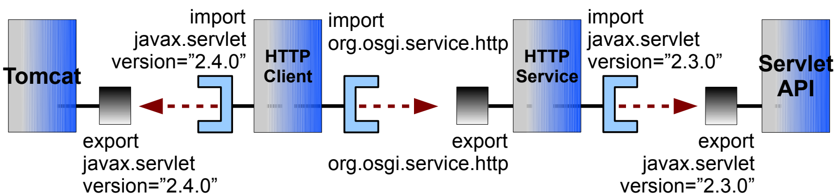


Figure 2.22 Subsequent HTTP client dependency resolution

Consider the servlet parameter in the `HTTPService.registerServlet()` method. Which version of `javax.servlet` is it? Since the HTTP service bundle is wired to the servlet API bundle, its parameter type is version `2.3.0` of `javax.servlet.Servlet`. When the HTTP client bundle tries to invoke `HTTPService.registerServlet()`, which version of `javax.servlet.Servlet` will be the instance it passes? Since it is wired to the Tomcat bundle, it will create a `2.4.0` instance of `javax.servlet.Servlet`. The class spaces of the HTTP service and client bundles are not consistent; two different versions of `javax.servlet` are reachable from both. At execution time, this would result in class cast exceptions when the HTTP service and client bundles interact. What went wrong?

The framework made the best choices at the time it resolved the bundle dependencies, but due to the incremental nature of the resolve process, it was not able to make the best

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

overall choice. If we installed all four bundles together, then the framework would have resolved the dependencies in a consistent way using its existing rules. Figure 2.23 shows the dependency resolution when all four bundles are resolved together.

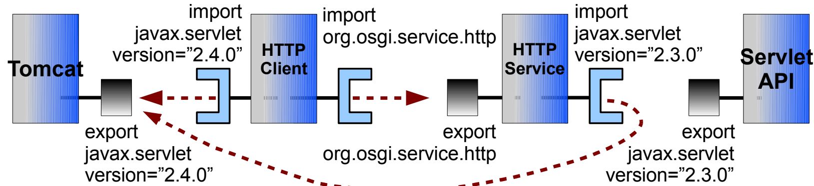


Figure 2.23 Consistent dependency resolution of HTTP service and client bundles

Since there is only one version of `javax.servlet` in use, we know the class spaces of the HTTP service and client bundles are consistent, allowing them to interact without issue. But is this a general remedy to class space inconsistencies? Unfortunately, it isn't as we shall see in the next chapter since OSGi allows us to dynamically install and uninstall bundles at any time. Moreover, inconsistent class spaces do not only result from incremental resolving of dependencies. It is also possible to resolve a static set of bundles into inconsistent class spaces due to inconsistent constraints. For example, imagine our HTTP service bundle required precisely version `2.3.0` of `javax.servlet`, while the client bundle required precisely version `2.4.0`. While these constraints are clearly inconsistent, the framework would happily resolve our example bundles given our current set of dependency resolution rules. Why doesn't it detect this inconsistency?

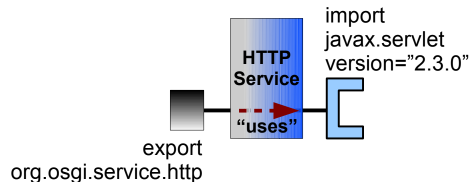


Figure 2.24 Bundle export "uses" import

INTER- VERSUS INTRA-BUNDLE DEPENDENCIES

The difficulty is `Export-Package` and `Import-Package` only capture inter-bundle dependencies, but class space consistency conflicts result from intra-bundle dependencies. Recall the `org.osgi.service.http.HttpService` interface; its `registerServlet()` method takes a parameter of type `javax.servlet.Servlet`, which means `org.osgi.service.http` uses `javax.servlet`. Figure 2.24 depicts this intra-bundle "uses" relationship between the HTTP service bundle's exported and imported packages.

How do these "uses" relationships arise? Our example shows the typical way, which is when the method signatures of classes in an exported package expose classes from other packages. This seems pretty obvious, since the used types are visible, but it is not always the case. You can also expose a type via a base class which is downcast by the consumer.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Since these types of "uses" relationships are important, how do we capture them in our bundle metadata?

THE "USES" DIRECTIVE

A directive attached to exported packages whose value is a comma-delimited list of packages exposed by the associated exported package.

The manifest syntax sidebar in section [ref] introduced the concept of a directive, but this is our first example of actually using one. Directives are additional metadata to alter how the framework interprets the metadata to which the directives are attached. The syntax for capturing directives is quite similar arbitrary attributes. For example, the following modified metadata for our HTTP service example shows how to use the "uses" directive:

```
Export-Package: org.osgi.service.http;
 uses:="javax.servlet"; version="1.0.0"
Import-Package: javax.servlet; version="2.3.0"
```

Notice that directives use the ":=" assignment syntax, but the ordering of the directives and the attributes is not important. In this particular example, we indicate `org.osgi.service.http` uses `javax.servlet`. How exactly does the framework use this information? Uses relationships among exported packages act like a grouping constraint for the packages. In this example, the framework ensures importers of `org.osgi.service.http` also use the same `javax.servlet` used by the HTTP service implementation.

This captures the previously missing intra-bundle package dependency. In this specific case, the "uses" relationship is on an imported package, but it can also be on exported packages. These sorts of "uses" relationships constrain which choices the framework can make when resolving dependencies, which is why they are also referred to as "uses" constraints. Abstractly, if package `foo` uses package `bar`, then importers of `foo` are

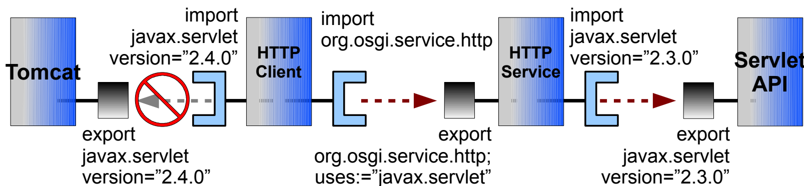


Figure 2.25 Uses constraints detect class space inconsistencies, so the framework can determine it is not possible to resolve the HTTP client bundle

constrained to the same `bar` if they use `bar` at all. Figure 2.25 depicts how this would have impacted our original incremental dependency resolutions.

For the incremental case, the framework can now detect inconsistencies in the class spaces and resolution will fail when we try to use the client bundle. Early detection is better than errors at execution time, since it alerts us to inconsistencies in our deployed set of

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

bundles. In the next chapter, we will learn about how we can cause the framework to re-resolve the bundle dependencies to remedy this situation.

We can further modify our example, to illustrate how "uses" constraints help find proper dependency resolutions. Assume the HTTP service bundle imports precisely version `2.3.0` of `javax.servlet`, but our client imports version `2.3.0` or greater. Typically, the framework will try to select the highest version of a package to resolve a dependency, but due to the "uses" constraint the framework will end up selecting a lower version instead, as depicted in Figure 2.26.

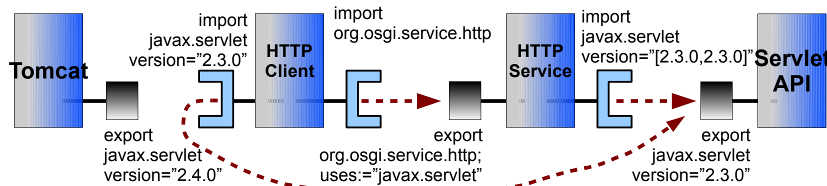


Figure 2.26 Uses constraints guide dependency resolution

If we look at the class space of the HTTP client, we can see how the framework ends up with this solution. The HTTP client's class space will contain both `javax.servlet` and `org.osgi.service.http`, since it imports these packages. From the perspective of the HTTP client bundle, it can use either version `2.4.0` or `2.3.0` of `javax.servlet`, but the framework only has one choice for `org.osgi.service.http`. Since `org.osgi.servicee.http` from the HTTP service bundle uses `javax.servlet`, the framework must choose the same `javax.servlet` package for any clients. Since the HTTP service bundle can only use version `2.3.0` of `javax.servlet`, this eliminates the Tomcat bundle as a possibility for the client bundle. The end result is a consistent class space, where a lower version of a needed package was correctly selected, even though a higher version was available.

Using uses

Let's finish up our discussion of "uses" constraints by touching on some final points. First, "uses" constraints are transitive, which means if a given bundle exports package `foo` which uses imported package `bar` and the selected exporter of `bar` uses package `baz`, then the associated class space for a bundle importing `foo` is constrained to have the same providers for both `bar` and `baz` if they are used at all. Also, even though "uses" constraints are important to capture, we don't want to just create blanket "uses" constraints, since it overly constrains dependency resolution. The framework has more leeway when resolving dependency on packages not listed in "uses" constraints, which is necessary to support side-by-side versions. For example, in larger applications, it is not uncommon for independently developed subsystems to use different versions of the same XML parser. If you specify "uses"

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

constraints too broadly, then this would not be possible. Accurate "uses" constraints are important, but luckily tools exist for generating them for exported packages.

Ok! We made it through the most difficult part and survived. Don't worry if you didn't understand every minute detail, since some of it may make more sense after you have more experience creating and using bundles. Let's turn our attention back to the paint program to review why we did all of this in the first place.

2.8 Reviewing the benefits of the modular paint program

Even though the amount of work to create the modular version of our paint program was not great, it was still more effort than if we left the paint program as it was. Why exactly did we create this modular version? Table 2.4 lists some of the benefits.

Table 2.4 Benefits of modularization in the paint program

Benefit	Description
Logical boundary enforcement	We can keep our implementation details private, since we are only exposing what we want to expose in the `org.foo.shape` public API package.
Reuse improvement	Our code is more reusable since we explicitly declare what each bundle depends on via `Import-Package` statements; this means we know what we need when using the code in different projects.
Configuration verification	We no longer have to guess if we have deployed our application properly, because OSGi verifies whether all needed pieces are present when launching the application.
Version verification	Similar to configuration verification, OSGi also verifies whether we have the correct versions of all our application pieces when launching the application.
Configuration flexibility	We are more easily able to tailor our application to different scenarios by creating new configurations; think of this as paint program a la carte.

Some of these benefits are more obvious than the others. Others we can demonstrate quite easily. For example, assume we forgot to deploy the shape API bundle in our launcher which we can simulate by deleting the `bundles/shape-2.0.jar` before launching the paint program as we did in section [ref]. You will see that it throws an exception like this:

```
org.osgi.framework.BundleException: Unresolved constraint in bundle 1:
package; (&(package=org.foo.shape)(version>=2.0.0)(!(version>=3.0.0)))
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The exact syntax of this message will become familiar to you when you read chapter 4, but ignoring the syntax it tells you the application is missing the `org.foo.shape` package, which is provided by the API bundle. Due to the on-demand nature of Java class loading, such errors are typically only discovered during application execution when the missing classes are used. With OSGi we are able to discover such issues with missing bundles or incorrect versions immediately. Besides detecting errors, let's look at how OSGi modularity helps us create different configurations of our application.

Creating a different configuration of our paint program is as simple as creating a new static `main()` method for our launcher to invoke. Currently, we are using the original static `main()` method provided by `PaintFrame`. In truth, it is not very modular to have our static `main()` on our implementation class; it is better to create a separate class so we don't need to recompile our implementation classes when we want to change the application's configuration. Listing 2.2 shows the existing static `main()` method from the `PaintFrame` class.

Listing 2.2 Existing `PaintFrame.main()` method implementation

```
public class PaintFrame extends JFrame
    implements MouseListener, MouseMotionListener {
  ...
  public static void main(String[] args) throws Exception {
    SwingUtilities.invokeAndWait(new Runnable() {
      public void run() {
        PaintFrame frame = new PaintFrame();                          #1
        frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        frame.addWindowListener(new WindowAdapter() {                 #2
          public void windowClosing(WindowEvent evt) {
            System.exit(0);
          }
        });
        frame.addShape(new Circle());                                 #3
        frame.addShape(new Square());
        frame.addShape(new Triangle());
        frame.setVisible(true);                                       #4
      }
    });
  }
```

The existing static `main()` is quite simple. At (#1), we create a `PaintFrame` instance and at (#2) we add a listener to cause the VM to exit when `PaintFrame` window is closed. Starting at (#3) we inject the various shape implementations into the paint frame and at (#4) we make the application window visible. The important aspect from the point of view of modularity is at (#3) where the shape implementations are injected into the paint frame. Since the configuration decision of which shapes to inject is hardcoded into the method, if we want to create a different configuration, then we must recompile the implementation bundle.

For example, assume we want to run our paint program on a small device only capable of supporting a single shape. To do so, we could modify `PaintFrame.main()` to only inject a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

single shape, but this would not be sufficient. We would also need to modify our metadata for the bundle so it would no longer depend on the other shapes, since they are no longer needed. Of course, after making these changes we have now lost the first configuration. These types of issues are arguments why the static `main()` method should be in a separate bundle.

So let's correct this situation in our current implementation. First we will delete the `PaintFrame.main()` method and modify its bundle metadata as follows:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.paint
Bundle-Version: 2.0.0
Bundle-Name: Simple Paint Program
Import-Package: javax.swing, org.foo.shape; version="2.0.0"
Export-Package: org.foo.paint; version="2.0.0"
```

The main paint program bundle no longer has any dependencies on the various shape implementations, but it now needs to export the package containing the paint frame. We can take the existing static `main()` method body and put it inside a new class, called `org.foo.fullpaint.FullPaint`, with the following bundle metadata:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.fullpaint
Bundle-Version: 1.0.0
Bundle-Name: Full Paint Program Configuration
Import-Package: javax.swing, org.foo.shape; org.foo.paint;
 org.foo.shape.circle; org.foo.shape.square; org.foo.shape.triangle;
 version="2.0.0"
Main-Class: org.foo.fullpaint.FullPaint
```

To launch this full version of our paint program, we simply use our bundle launcher to deploy all of the associated bundles including this `FullPaint` bundle. Likewise, we could create a different bundle containing the `org.foo.smallpaint.SmallPaint` class in Listing 2.3 to launch a small configuration of our paint program containing only the circle shape.

Listing 2.3 New launcher for "smaller" paint program configuration

```
package org.foo.smallpaint;

public class SmallPaint {
  public static void main(String[] args) throws Exception {
    SwingUtilities.invokeAndWait(new Runnable() {
      public void run() {
        PaintFrame frame = new PaintFrame();
        frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        frame.addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent evt) {
            System.exit(0);
          }
        });
        frame.addShape(new Circle());                                    #A
        frame.setVisible(true);
      }
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    });
  }
```
#A Only the circle shape implementation is injected

The metadata for the bundle containing our small paint program configuration is:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.smallpaint
Bundle-Version: 1.0.0
Bundle-Name: Reduced Paint Program Configuration
Import-Package: javax.swing, org.foo.shape; org.foo.paint;
 org.foo.shape.circle; version="2.0.0"
Main-Class: org.foo.smallpaint.SmallPaint
```

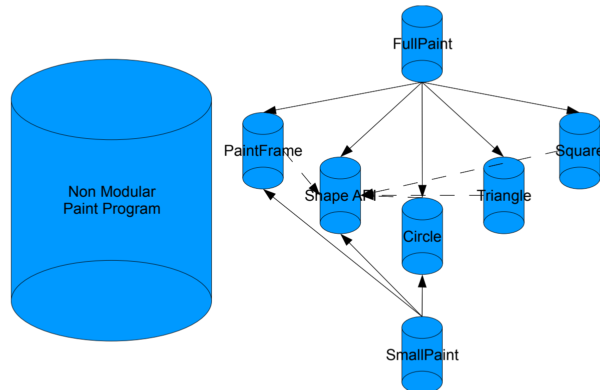


Figure 2.27 Modular and non modular versions of the paint program

This small configuration only depends on Swing, the public API, the paint program implementation, and the circle implementation. When launching the full configuration, all shape implementations are required, but for the small configuration only the circle implementation is required. Now we can deploy the appropriate configuration of our application based on the target device and have OSGi verify the correctness of it all. Pretty sweet. For completeness we've added a quick sketch of the before and after view of the paint program as shown in figure 2.27.

2.9 Summary

We've covered a lot of ground in this chapter, some of the highlights include:

- Modularity is a form of separation of concerns which provides both logical and physical encapsulation of classes.
- Modularity is desirable since it allows us to break our applications into logically independent pieces that can be independently changed and reasoned about.
- A bundle is the name for a module in OSGi, which is a JAR file containing code, resources, and modularity metadata.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

- Modularity metadata details human readable information, bundle identification, and code visibility.
- Bundle code visibility is composed of an internal class path, exported packages, and imported packages.
- The OSGi framework uses the metadata about imported and exported packages to automatically resolve bundle dependencies and ensure type consistency before a bundle can be used.
- Imported and exported packages capture inter-bundle package dependencies, but "uses" constraints are necessary to capture intra-bundle package dependencies to ensure complete type consistency.

From here, we will move onto the lifecycle layer, where we enter execution time aspects of OSGi modularity. While this chapter was all about describing our bundles to the OSGi framework, the lifecyle layer is all about actually using our bundles and the facilities provided by the OSGi framework at execution time.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

3
Learning Lifecycle

In the last chapter we looked at the OSGi modularity layer and introduced you to bundles; a bundle is OSGi terminology for a module, which is a JAR file with the extra modularity metadata. We use bundles to define both the logical (i.e., code encapsulation and dependencies) and physical (i.e., deployable units) modularity of our applications.

The OSGi modularity layer goes to great lengths to ensure class loading happens in a consistent and predictable way. However, to avoid putting the cart before the horse, in the last chapter we had to gloss over the details of how we actually installed bundles into an OSGi framework. No longer, in this chapter we will look at the next layer of the OSGi stack, the lifecycle layer.

As we saw in Chapter 2, to use a bundle we install it into a running instance of the OSGi framework. So, creating a bundle is the first half of leveraging OSGi's modularity features, the second half is using the OSGi framework as a runtime to manage and execute bundles. The lifecycle layer is quite unique in allowing you to create externally (and remotely) managed applications or completely self-managed applications (or any combination of the two). It also introduces dynamism that is not normally part of an application.

This chapter will familiarize you with the features of the lifecycle layer and explain how to effectively use them. In the next section, we take a closer look at what lifecycle management is and why you should care about it, followed by the definition of the OSGi bundle lifecycle. In the subsequent section, we learn about the API you use for managing the lifecycle of your bundles. Throughout this chapter we will bring all the points back home via examples of a simple OSGi shell and a lifecycle-aware version of our paint program.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

3.1 Introducing lifecycle management

The OSGi lifecycle layer provides a management API and a well-defined lifecycle for bundles in an OSGi runtime. The lifecycle layer serves two different purposes:

- External to your application, the lifecycle layer precisely defines the bundle lifecycle operations. These lifecycle operations allow you to manage and evolve your application by dynamically changing the composition of bundles inside a running framework.
- Internal to your application, the lifecycle layer defines how your bundles gain access to their execution context, which provides them with a way to interact with the OSGi framework and the facilities it provides at execution time.

But let's take a step back. It's all very well stating what the OSGi lifecycle layer does, but this won't necessarily convince you of it's worth. Instead, let's give you quick example of how it can improve your applications with a real-world scenario.

3.1.1 What is lifecycle management

Imagine you have a business application able to report management events via JMX. Do you always want to enable or even install the JMX layer? You could imagine running in a light-weight configuration and only enabling the JMX notifications on demand? The lifecycle layer allows you to externally install, start, update, stop and uninstall different bundles to customize your application's configuration at execution time.

Further, imagine if there is some critical failure event within your application that must trigger the JMX layer to send out a notification whether or not the administrator had previously enabled or installed it. The lifecycle layer also provides programmatic access to bundles so they can internally modify their application's configuration at execution time.

Generally speaking, programs (or parts of a program) are subject to some sort of lifecycle whether this is explicit or not. There are typically four distinct phases of software lifecycle as shown in Figure 3.1.

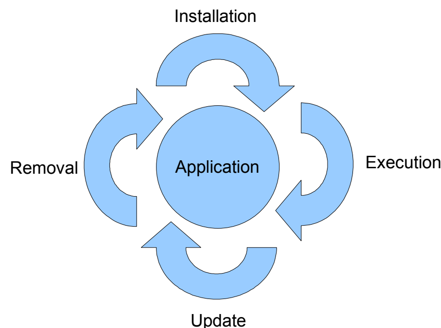


Figure 3.1 The four phases of software lifecycle, where an application is installed so we can execute it, later it be updated to a newer version, or utlimately removed if it is no longer needed

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

If you are creating an application, think about the typical lifecycle of the application as a whole. First you need to install it. Assuming all its dependencies are satisfied, you can execute it, which allows it to start acquiring resources. When the application is no longer needed you stop the application, which allows it to release any resources and perhaps persist any important state. Over time you might want to update the application to a newer version. Ultimately, the application may be removed because it is no longer needed. For non-modular applications, the lifecycle operates on the application as a whole, but as we will see for modular applications, fine-grained lifecycle management is possible for individual pieces of the application.

Let's review some of the more popular models for creating applications in Java and how they manage the software lifecycle.

Java lifecycles

Standard Java

For the purposes of this discussion, we will equate an application in standard Java to a JAR file containing the `Main-Class` header in its manifest, which allows it to be easily executed. In standard Java development, the lifecycle of an application is very simple. Such a JAR-based Java application is **installed** when downloaded. It is **executed** when the user launches a JVM process, typically by double-clicking on it. The application is stopped when the program terminates. **Updating** is usually done by replacing the JAR with a newer version. **Removal** is achieved by deleting the jar from the filesystem.

Servlet

In servlet development, the lifecycle of the web application is managed by the servlet container. The application is **installed** via a container specific process, sometimes this involves dropping a WAR file containing the application in a certain directory in the file system or uploading a WAR file via a web management interface. The servlet container calls various lifecycle API methods on the sub-components of the WAR file such as `Servlet.init()` and `Servlet.destroy()` during the **execution** phase of the application's lifecycle. In order to **update** the application a completely new WAR file is generated. The existing WAR must be stopped and the new WAR file started in it's place. The application is **removed** by a container specific process, again sometimes removing the WAR from the file system or interacting with a management interface.

Netbeans

No idea of process – does anyone else know

Maven

Some idea of process but others are probably best to write

Technology X

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

As you can see there are many different lifecycle processes in use in Java today. In traditional Java applications, the lifecycle is largely managed by the platform-specific mechanism of the underlying operating system via installers and double-clicking on desktop icons. For modular development approaches, such as Servlets, Java EE, or Netbeans, each has it's own specific mechanism of handling the lifecycle of their own components. This leads us to the question of why do we need lifecycle management at all?

3.1.2 Why lifecycle management?

Cast your mind back to our earlier discussion about why you should modularize your application code into separate bundles. We talked about the benefits of separating different concerns into separate bundles and avoiding tight coupling among them. The OSGi module layer provides you with the necessary means to do this at the class level, but it doesn't address *when* a particular set of classes or objects are needed in an application.

An explicit lifecycle API lets the providing application take care of how to configure, initialize, and maintain a piece of code that is installed so it can decide how it should operate at execution time. For example, if a database driver is in use, should it start any threads or initialize any cache tables to improve performance? If it does any of these things when are these resources released? Do they exist for the lifetime of the application as a whole and if not how are they removed? Since the OSGi specification provides an explicit lifecycle API, this allows you to take any bundle providing the functionality you need and let it worry about how to manage its internal functions. In essence: compose versus control.

By allowing you to architect your application such that parts of it may come and go at any point in time greatly increases your flexibility. You can easily manage installation, update, and removal of an application and its required modules. You can configure or tailor applications to suit specific needs, breaking the monolithic approach of standard development approaches. Instead of “you get what you get”, wouldn't it be great if you could offer “you get what you need”? Hopefully, this discussion piqued your interest. Now let's focus specifically on defining the OSGi bundle lifecycle and the management API associated with it.

3.2 OSGi bundle lifecycle

The OSGi lifecycle layer is how we use our bundles; it is where the rubber meets the road. The module metadata from the previous chapter is all well and good, but creating bundles in and of itself is only useful if we actually use them. We need to interact with the OSGi lifecycle layer in order to use our bundles. Unlike the modularity layer which relies on metadata, the lifecycle layer relies on API. Since introducing API can be a boring endeavor (JavaDoc anyone?), we move in a top-down fashion and show what the lifecycle layer API allows us to do using an example.

It is important to note that the OSGi core framework does not mandate any particular mechanism of interacting with the lifecycle API (e.g., command line, GUI, or XML

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

configuration file), the core is purely a Java API. In fact, this turns out to be extremely powerful as it makes it possible to design as many different ways of managing the OSGi framework as you can think of; in the end, we are only limited by our imagination as developers.

Since there is no standard way for users to interact with the lifecycle API, we *could* use a framework-specific mechanism. However, doing so would actually be doing you, the reader, a disservice since it is a great opportunity for learning. Instead of reusing someone else's work in this chapter, we lead you through some very basic steps to develop our very own command-line interface for interacting with the OSGi framework. This gives us the perfect tool, along side the paint program, to explore the rich capabilities provided by the OSGi lifecycle API.

SHELLS, SHELLS, EVERYWHERE

For readers with some familiarity using OSGi frameworks, you are likely aware that most OSGi framework implementations (e.g., Apache Felix, Eclipse Equinox, Knopflerfish) have their own shells for interacting with a running framework. The OSGi specification does not define a standard shell (although there has been some work toward this goal recently), but shells need not be tied to a specific framework and can be implemented as bundles, just like we are going to do here.

3.2.1 Introducing lifecycle to the paint program

Enough with the talk, let's see the lifecycle API in action by kicking off our shell application and using it to install the paint program. To do this, type the following into your operating system console (windows users substitute \ for /):

```
cd chapter03/shell-example/
java -jar launcher.jar bundles
Bundle: org.foo.shell started with bundle id 1 - listening on port 7070
```

Our shell is created as a bundle which has, upon starting, begun listening for user input on a telnet socket. This allows clients to connect and perform install, start, stop, update, and uninstall actions on bundles. It also provides some basic diagnostic facilities. Lets connect to our newly launched framework and use the telnet console to install our paint example:

```
telnet localhost 7070
-> install file:../paint-example/bundles/paint-3.0.jar
Bundle: 3
-> install file:../paint-example/bundles/shape-3.0.jar
Bundle: 4
-> start 3
-> install file:../paint-example/bundles/circle-3.0.jar
Bundle: 5
-> install file:../paint-example/bundles/square-3.0.jar
Bundle: 6
-> start 5
-> start 6
-> install file:../paint-example/bundles/triangle-3.0.jar
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Bundle: 7
-> start 7
-> stop 5
```

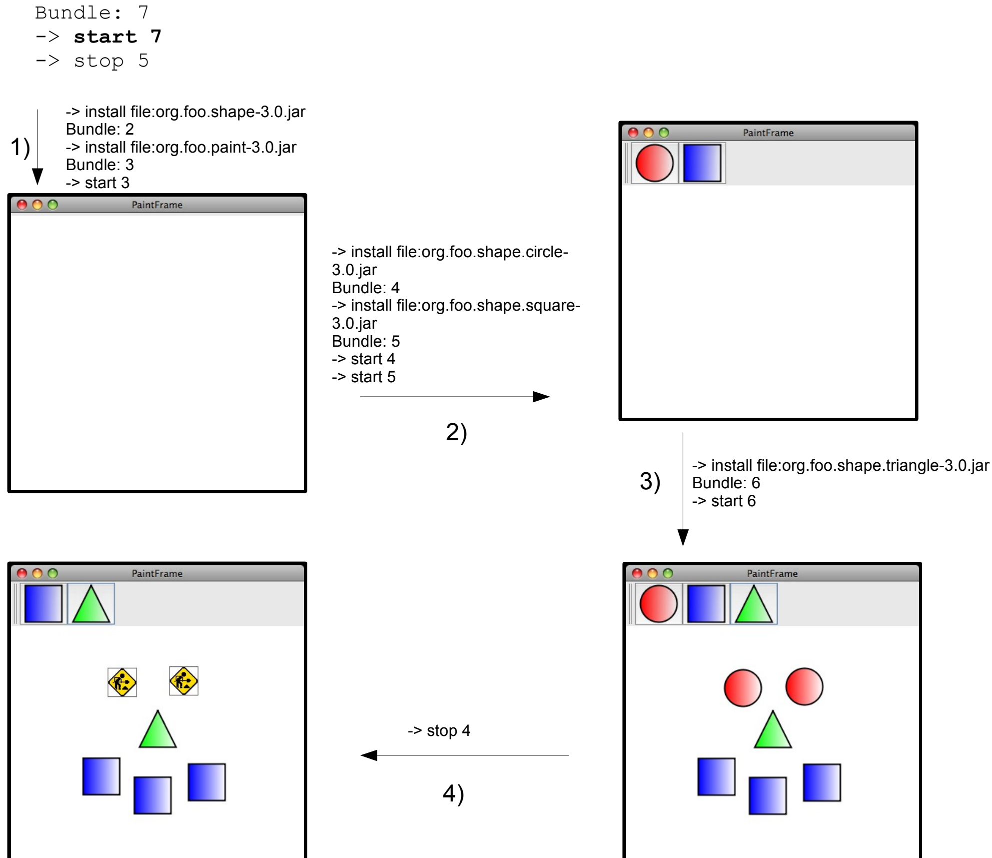


Figure 3.2 Execution-time evolution – Dynamically add and remove shapes from the paint program as if by magic.

From Figure 3.2, you can see in step (1) we first install the shape API bundle, then we install and start the paint program bundle. This causes an empty paint frame to appear with no available shapes, which makes sense since we haven't installed any other bundles yet. In step (2) we install and start the circle and square bundles. As if by magic, the two shapes dynamically appear in the paint frame's toolbar and are available for drawing. In step (3) we install and start the triangle bundle, then we draw some shapes on the paint canvas. So what happens if we stop a bundle? In step (4) we stop the circle bundle, which we can see is replaced in the canvas with the placeholder icon (i.e., the construction worker) from `DefaultShape`.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

This shows you in practice how the lifecycle API can be used to build a highly dynamic application, but what's actually going on in this example? To understand this, let's take a top-down approach, using our shell and paint example for context:

- First, in section 3.2.2 we will explain the framework's role in the application's lifecycle.
- Second, in section 3.2.3 we will look at the changes we need to make to the bundle manifest to hook our bundles into the OSGi framework.
- Third, in section 3.2.4 we will investigate the key API interfaces used by the OSGi lifecycle; `BundleActivator`, `BundleContext` and `Bundle`.
- Finally, in section 3.2.5 we will wrap up our top-down approach with a review of the OSGi lifecycle state diagram.

Let's get started.

3.2.2 The OSGi framework's role in lifecycle

In standard Java programming, you use your JAR files by placing them on the class path. This is not the approach for using bundles. A bundle can only be used when it is installed into a running instance of the OSGi framework. Conceptually, you can think of installing a bundle into the framework as being similar to putting a JAR file on the class path in standard Java programming. In other words, installing a bundle makes it available for use, just like putting a JAR on the class path makes it available for use.

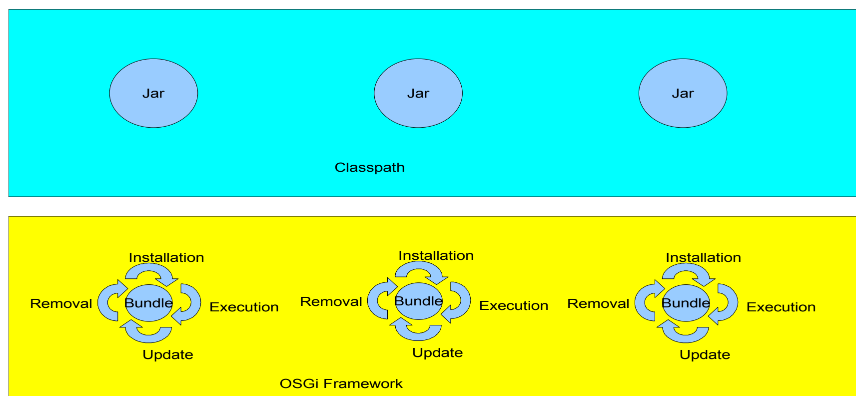


Figure 3.1: Classpath v.s. OSGi framework with full lifecycle management.

This simplified view does hide some important differences from the standard class path as you can see in figure Figure 3.1. One big difference is the OSGi framework supports full lifecycle management of bundle JAR files, including install, resolve, start, stop, update, and uninstall. At this point, we have only really touched upon installing bundles and resolving

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

their dependencies. The remainder of this chapter will fully explain all of the lifecycle activities and how they are related to each other. For example, we've already mentioned the framework doesn't allow an installed bundle to be used, until its dependencies (i.e., `Import-Package` declarations) are satisfied.

Another huge difference from the standard class path is inherent dynamism. The OSGi framework supports the full bundle lifecycle at execution time. This is similar to modifying what's on the class path dynamically.

As part of the lifecycle management, the framework maintains a persistent cache of installed bundles. This means the next time you start the framework, any previously installed bundles are automatically reloaded from the bundle cache and the original JAR files are no longer necessary. Perhaps we can characterize the framework as a fully manageable, dynamic, and persistent class path. Sounds cool, huh? Let's move on to discussing how we have to modify our bundle metadata to allow it to hook into the lifecycle layer API.

3.2.3 The bundle activator manifest entry

So now we know you have a good idea of what the framework is doing when we install our bundles, but how do we tell the framework how to kickstart our bundles at execution time? The answer, as with the rest of the modularity information, is via the bundle metadata. Let's look at the JAR file manifest describing the shell bundle we will create in Listing 3.1:

Listing 3.1 Shell bundle manifest headers

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2                                              #1
Bundle-SymbolicName: org.foo.shell                                     #1
Bundle-Version: 1.0                                                    #1
Bundle-Activator: org.foo.shell.Activator                              #2
Import-Package: org.osgi.framework;version="[1.3,2.0)",org.osgi.servic #3
 e.packageadmin;version="[1.2,2.0)",org.osgi.service.startlevel;versio
 n="[1.1,2.0)"
Bundle-Name: remote_shell                                              #4
Bundle-DocURL: http://code.google.com/p/osgi-in-action/                #4
```

You should already be familiar with most of these headers from the last chapter, but to recap most of these entries are related to the class-level modularity of the bundle, i.e.:

- (#1) defines the bundles identity,
- (#3) specifies the packages on which this bundle depends, and
- (#4) declares additional human-readable information.

The only new header is `Bundle-Activator` at (#2). This is our first sighting of the OSGi lifecycle API in action! The `Bundle-Activator` header specifies the name of a class on the bundles class path implementing the `org.osgi.framework.BundleActivator` interface. This interface provides our bundle with a hook into the lifecycle layer and the ability to customize what happens when it is “started” or “stopped”.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

IS AN ACTIVATOR NECESSARY?

Keep in mind, not all bundles need an activator. An activator is only necessary if you are creating a bundle wishing to specifically interact with OSGi API or needing to perform some custom initialization/deinitialization actions. If you are creating a simple library bundle, then it is not necessary to give it an activator since it is possible to share classes without one. This doesn't mean your bundles won't be able to do anything. In fact, bundles do not need to be started at all to do useful things. Just remember the paint program we created in the last chapter; none of our bundles had activators nor did any of them need to be started, but we still created a fully functioning application.

In order to understand what is going on in our shell example we now have to introduce you to three interfaces (`BundleActivator`, `BundleContext`, and `Bundle`), which are the heart and soul of the lifecycle layer API.

3.2.4 Introducing the lifecycle API

The last section described how our shell bundle declares a `BundleActivator` to hook into the framework at execution time. We can now look into the details of this interface and the other lifecycle API made available from it to our bundle. This is our bundle's hook into the world of OSGi!

BUNDLE ACTIVATOR

As we have seen, adding an activator to our bundle is straightforward, since we only need to create a class implementing the `BundleActivator` interface, which looks like this:

```
public interface BundleActivator {
  public void start(BundleContext context) throws Exception;
  public void stop(BundleContext context) throws Exception;
}
```

For our shell example, the activator allows it to become lifecycle aware and gain access to framework facilities. Listing 3.2 shows the activator for our shell bundle.

Listing 3.2 Simple shell bundle activator

```
package org.foo.shell;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Activator implements BundleActivator {                  #1
  private volatile Binding m_binding;                                #2

  public void start(BundleContext context) {                         #3
    int port = getPort(context);                                     #4
    int max = getMaxConnections(context);                            #5
    m_binding = getTelnetBinding(context, port, max);                #6
    m_binding.start();                                               #7
    System.out.println("Bundle " +                                   #8
      context.getBundle().getSymbolicName() +
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
        " started with bundle id" +
        context.getBundle().getBundleId() +
        " listening on port " + port);
  }

  public void stop(BundleContext context) {                                 #9
    m_binding.stop();                                                       #10
  }

    ...
}
...

public interface Binding {                                                  #11
  public void start();                                                      #12
  public void stop() throws InterruptedException;                           #13
}
```

You can see at (#1) our bundle activator implements the OSGi `BundleActivator` interface, with the `start()` method implemented at (#3) and `stop()` at (#9). When our bundle is installed and started, the framework constructs an instance of this activator class and invokes the `start()` method. When the bundle is stopped, the framework invokes the `stop()` method.

The `start()` method is the actual starting point for your bundle, sort of like the static `main()` method in standard Java. After it returns, your bundle is expected to function until the `stop()` method is invoked at some later point. Typically, a bundle performs any necessary initialization in its `start()` method including (but not limited to):

- Opening network sockets,
- Initializing cache tables,
- Registering event callback interfaces, or
- Starting threads.

The `stop()` method should undo anything done by the `start()` method. Here we should mention a couple of technical, but potentially important details about the handling of the `BundleActivator` instance.

- First, the activator instance on which `start()` is called, will be the same instance on which `stop()` is called.
- Second, after `stop()` is called, the activator instance is discarded and will not be used again.
- If the bundle is subsequently restarted after being stopped, a new activator instance will be created and the `start()` and `stop()` methods will be invoked on it as appropriate.

As you can see, the rest of the activator isn't very complicated. At (#4) we get the port on which the bundle will listen for connection requests and at (#5) we get the number of

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

allowed concurrent connections. We create a `TelnetBinding` at (#6), which will do the work of listening on a socket for user input and process it; the details of creating the telnet binding are omitted here for reasons of simplicity and brevity. The next step is to start the binding, which creates a new `Thread` to run our shell. How this happens is left to the binding, which we start at (#7).

The point about the binding starting its own thread is important because the activator methods shouldn't do much work. This is best practice as with most callback patterns, which are supposed to return quickly, allowing the framework to carry on managing other bundles. However it is also important to point out that the OSGi specification does not mandate you start a new thread if your applications startup doesn't warrant it – the ball is in your court so to speak.

THREADING

OSGi is designed around the normal Java thread abstraction. Unlike other, more heavyweight frameworks, it assumes you do your own thread management. You gain a lot of freedom by this, but at the same time you have to make sure your programs are correctly synchronized and thread safe. In this simple example, nothing special is needed, but, in general, it is likely `stop()` will be called on a different thread than `start()` (for this reason we make the member at (#2) `volatile`). The OSGi libraries are thread safe and callbacks are normally done in a way to give you some guarantees. For example, in the case of the bundle activator, `start()` and `stop()` are guaranteed to be called in order and not concurrently, but they still might be executed by different threads. Your code must take this into account as far as other threads and visibility of member variables are concerned.

For the activator `stop()` method at (#9), all we do is tell the binding to stop listening to user input and cease to execute. We should make sure it really does stop by waiting until its thread is finished; the binding method waits for its thread to stop. Sometimes you might have special cases for certain situations because, as you will see later on, it might be the shell thread itself calling the `stop()` method, which would cause the bundle to freeze. We'll cover these and other advanced use cases in section [ref]. In general, if you use threads in your bundles, do so in such a way that all threads are stopped when the `stop()` method returns.

Now we have seen how we can handle starting and stopping our application. But what if we want to interact with the OSGi framework itself? We will now switch our focus to the `BundleContext` object passed into the `start()` and `stop()` methods of our activator at (#3) and (#9); this allows the bundle to interact with the framework and manage other bundles.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

BUNDLE CONTEXT

As we learned in the previous section, the framework calls the `start()` method of a bundle's activator when it is started and the `stop()` method when it is stopped. Both methods receive an instance of the `BundleContext` interface. The methods of the `BundleContext` interface can be roughly divided into two categories.

- The first category is related to deployment and lifecycle management, which we will focus on in this chapter.
- The second category is related to bundle interaction via services, which will be discussed in the next chapter.

We are interested in the first category of methods, since they give us the ability to install and manage the lifecycle of other bundles, access information about the framework, and retrieve basic configuration properties. Listing 3.3 captures these methods from BundleContext.

Listing 3.3 `BundleContext` methods related to lifecycle management

```
public interface BundleContext {
  ...
  String getProperty(String key);
  Bundle getBundle();
  Bundle installBundle(String location, InputStream input)
    throws BundleException;
  Bundle installBundle(String location) throws BundleException;
  Bundle getBundle(long id);
  Bundle[] getBundles();
  void addBundleListener(BundleListener listener);
  void removeBundleListener(BundleListener listener);
  void addFrameworkListener(FrameworkListener listener);
  void removeFrameworkListener(FrameworkListener listener);
  ...
}
```

We will cover most of these methods in this chapter. The second category of `BundleContext` methods related to services will be covered in the next chapter.

UNIQUE CONTEXT

One important aspect of the bundle context object is it represents the unique execution context of the bundle. Since it represents the execution context, it is only valid while the associated bundle is active, which is explicitly from the moment the activator `start()` method is invoked until the activator `stop()` method completes and the entire time in between. Most bundle context methods throw an exception if used when the associated bundle is not active. It is a unique execution context since each activated bundle receives its own context object. The framework uses this context for security and resource allocation purposes for each individual bundle.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Given this capability-like nature of `BundleContext` objects, they should be treated as sensitive or private objects and not passed freely among bundles.

Our shell activator in Listing 3.2 uses the bundle context to get its configuration property values at (#4) and (#5). It also passes the context into the telnet binding at (#6), which client connections will use to interact with the running framework. Lastly, it uses the context at (#8) to obtain our bundle's `Bundle` object to access our identification information. We will crack open the details of to actually use these interfaces when we get to section [ref], but for now we will continue our top-down description by looking at the final interface, namely `org.osgi.framework.Bundle`.

BUNDLE

For each installed bundle, the framework creates a `Bundle` object to logically represent it. The `Bundle` interface defines the API to manage an installed bundle's lifecycle; a portion of the interface is presented in Listing 3.4.

Listing 3.4 `Bundle` interface methods related to lifecycle management

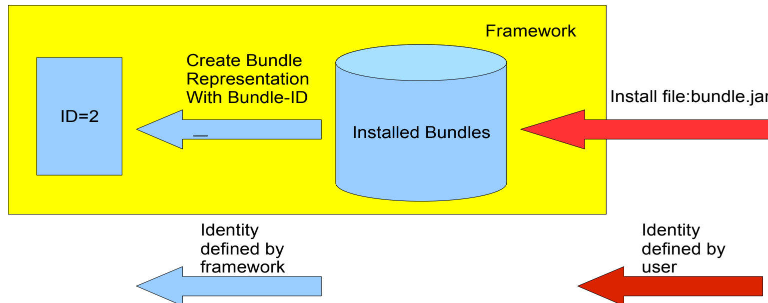
```
public interface Bundle {
  ...
  BundleContext getBundleContext();
  long getBundleId();
  Dictionary getHeaders();
  Dictionary getHeaders(String locale);
  String getLocation();
  int getState();
  String getSymbolicName();
  Version getVersion();
  void start(int options) throws BundleException;
  void start() throws BundleException;
  void stop(int options) throws BundleException;
  void stop() throws BundleException;
  void update(InputStream input) throws BundleException;
  void update() throws BundleException;
  void uninstall() throws BundleException;
  ...
}
```

As we cover most of these methods, we will see that most lifecycle operations are method on the `Bundle` object. Unlike `BundleContext` objects, `Bundle` objects do not represent sensitive capabilities, so you can freely pass them around among bundles, although this is rarely necessary.

Each installed bundle is uniquely identified in the framework by its `Bundle` object. From the `Bundle` object, you can also access two additional forms of bundle identification: the bundle identifier and the bundle location. You might be thinking, “Didn't we talk about bundle identification metadata back in chapter 2?” Yes, we did, but don't get confused. The identification metadata described in section [ref] statically identified the bundle's JAR file and its contents. The bundle identifier and bundle location are for execution-time identification,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

meaning they are associated with the `Bundle` object. You might be wondering why we need two different execution-time identifiers?



Ch3Figure 3.3: Difference between the bundle identifiers.

The main difference between the two is who defines the identifier. The bundle identifier is a Java language `long` value assigned by the framework in ascending order as bundles are installed. The bundle location is a `String` value assigned by the installer of the bundle.

BUNDLE LOCATION INTERPRETATION

The bundle location does have a unique characteristic since most OSGi framework implementations interpret it as a URL pointing to the bundle JAR file. The framework then uses this URL to download the contents of the bundle JAR file during bundle installation. The specification does not actually define the location string as an URL, nor is it actually required since you can install bundles from an input stream as well.

Both the bundle identifier and location values uniquely identify the `Bundle` object and persist across framework executions when the installed bundles are reloaded from the framework's cache.

You still might be thinking, "I am not convinced all of these identification mechanisms are necessary. Couldn't we just find the `Bundle` object using the bundle's symbolic name and version from chapter 2?" In reality, yes, we could because the framework only allows one bundle with a given symbolic name and version to be installed at a time. This means the bundle symbolic name and version pair also acts as an execution-time identifier.

Why so many forms of identification?

History plays a role here. As mentioned in chapter 2, the notion of using bundle symbolic name and version to uniquely identify a bundle did not exist in versions of the specification prior to R4. Therefore, prior to R4 it made sense to have internally and externally assigned identifiers. Now it makes less sense, since the bundle symbolic name

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

and version pair are externally defined and explicitly recognized internally by the framework.

There is still a role for the bundle identifier since in some cases the framework treats a lower identifier value as being better than a higher one when deciding between two otherwise equal alternatives, such as when there are two bundles exporting the same version of a given package. The real loser here is the bundle location, which really doesn't serve a very useful purpose other than potentially giving the initial URL of the bundle JAR file.

While one instance of `Bundle` exists for each bundle installed into the framework, at execution time there is also a special instance of `Bundle` to represent the framework itself. This special bundle is called the system bundle and, although the API is the same, it merits its own discussion.

The system bundle

At execution time the framework is represented as a bundle with an identifier of zero, called the system bundle. You do not actually install the system bundle, it always exists while the framework is running.

The system bundle follows the same lifecycle as normal bundles, so it can be manipulated with the same operations as normal bundles. Lifecycle operations performed on the system bundle have special meanings when compared to normal bundles, however. One example of the special meaning is evident when you stop the system bundle. Intuitively, stopping the system bundle shuts down the framework in a well-behaved manner. It will stop all other bundles first and will then shut itself down completely.

With that, we conclude our high-level look at the major API players in the lifecycle layer (`BundleActivator`, `BundleContext`, and `Bundle`). We now know:

- `BundleActivator` is the entry point for our application, much like static `main()` in a standard Java application,
- `BundleContext` provides our applications with the methods to manipulate the OSGi framework at execution time, and
- `Bundle` represents an installed bundle in the framework allowing state manipulations to be performed on it.

With this knowledge in hand, we will now complete our top-down approach by defining the overall bundle lifecycle state diagram and see how these interfaces relate to it.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

3.2.5 Lifecycle state diagram

Up until now we have been holding off on explicitly describing the complete bundle lifecycle in favor of getting a high-level view of the API comprising the lifecycle layer. This allowed us to quickly get our hands a little dirty. Now we can better understand how these APIs relate to the complete bundle lifecycle state diagram, which is depicted in Figure 3.4.

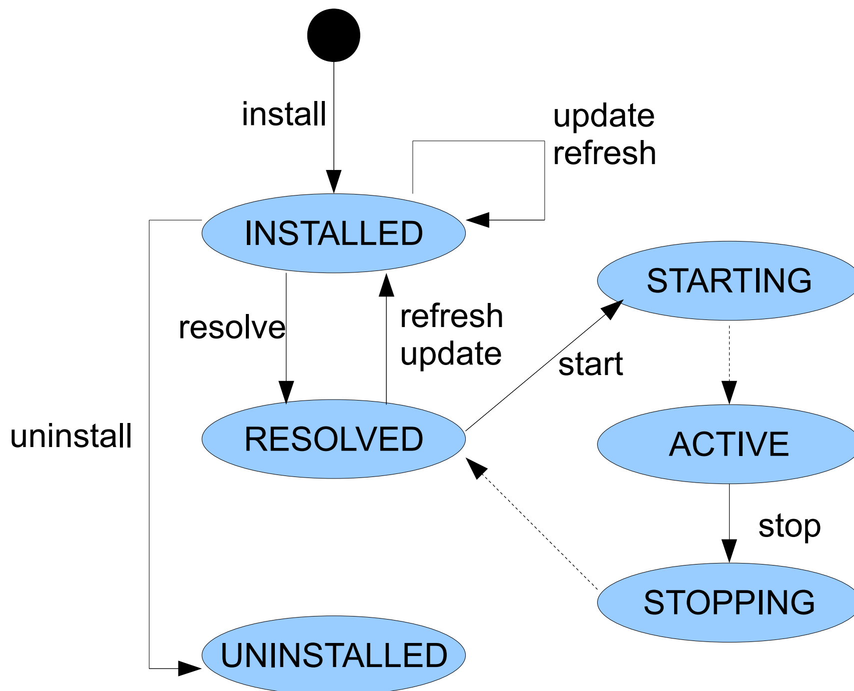


Figure 3.4 OSGi bundle lifecycle

The entry point of the bundle lifecycle is the BundleContext.installBundle() operation, which creates a bundle in the `INSTALLED` state. From the figure, you can see there is no direct path from `INSTALLED` to `STARTING`. This is because the framework ensures all dependencies of a bundle are satisfied before it can be used (i.e., no classes can be loaded from it). The transition from the `INSTALLED` to `RESOLVED` state represents this guarantee. The framework will not allow a bundle to transition to `RESOLVED` unless all its dependencies are satisfied. If it cannot transition to `RESOLVED`, then by definition it cannot transition to `STARTING`. Often the transition to `RESOLVED` happens implicitly when the bundle is started or another bundle tries to load a class from it, but we will see later in this chapter it is also possible to explicitly resolve a bundle.

The transition from the `STARTING` to `ACTIVE` state is always implicit. A bundle is in the `STARTING` state while its activator's `start()` method executes. Upon successful completion

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

of the `start()` method the bundle's state transitions to `ACTIVE`, but if the activator throws an exception it transitions back to `RESOLVED`.

A n `ACTIVE` bundle can be stopped, which also results in a transition back to the `RESOLVED` state via the `STOPPING` state. The `STOPPING` state is an implicit state like `STARTING` and the bundle is in this state while its activator's `stop()` method executes. The reason a stopped bundle transitions back to `RESOLVED` instead of `INSTALLED` is because its dependencies are still satisfied and do not need to be resolved again. It is possible to force the framework to resolve a bundle again by refreshing it or updating it, which we will discuss later. Refreshing or updating a bundle transitions it back to the `INSTALLED` state.

A bundle in the `INSTALLED` state can be uninstalled, which will transition it to the `UNINSTALLED` state. If you uninstall an active bundle, the framework will automatically stop the bundle first, which results in the appropriate state transitions to the `RESOLVED` state and then transition it to the `INSTALLED` state before uninstalling it[1]. A bundle in the `UNINSTALLED` state will remain there as long as it is still needed (we will explain later what this means in xxx), but it can no longer transition to another state. Now we understand the complete bundle lifecycle, so let's discuss how these operations impact the framework's bundle cache and subsequent restarts of the framework.

3.2.6 Bundle cache and framework restarts

To use our bundles, we have to install them into the OSGi framework. Check. But what does this actually mean? Technically, we know we must invoke `Bundle.installBundle()` to install a bundle. In doing so, we must specify a location typically interpreted as an URL to the bundle JAR file or an input stream from which the bundle JAR file is read. In either case the framework reads the bundle JAR file and saves a copy in a private area known as the bundle cache. This means two things:

1. Installing a bundle into the framework is a persistent operation.
2. Once installed, the original copy of the bundle JAR file is no longer needed by the framework.

The exact details of the bundle cache are dependent the framework implementation; the specification does not dictate the format or structure other than it be persistent across framework executions. So if you start an OSGi framework, install a bundle, shutdown the framework, and then restart it, the bundle you installed will still be there. If you compare it to using the class path where you manually manage everything, having the framework cache and manage the artifacts relieves you from a lot of effort.

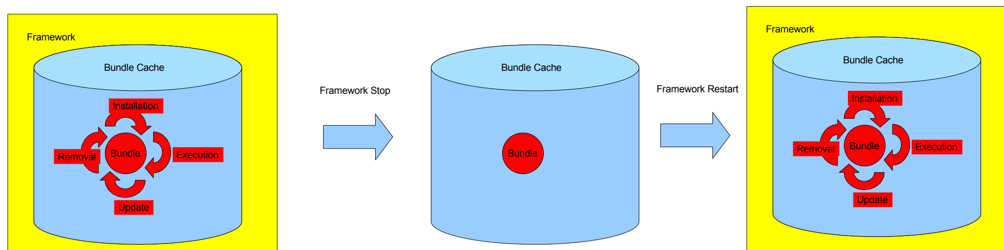
In terms of your application, you can think of the bundle cache as the deployed configuration of your application. This is similar to the discussion about creating different configurations of the paint program from the last chapter. You application's configuration is

[1]This is a change in the new version 4.2 of the OSGi specification. You cannot go to UNINSTALLED from RESOLVED you have to go to INSTALLED first and only INSTALLED goes to UNINSTALLED. This is an errata from the R4.2 spec

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

whichever bundles you install into the framework. You maintain and manage the configuration using the APIs and techniques discussed in this chapter.

Bundle installation is not the only lifecycle operation to impact the bundle cache. When a bundle is started using `Bundle.start()`, the framework persistently marks the bundle as started, even if `Bundle.start()` throws an exception, such as when the bundle cannot be resolved or the bundle's `BundleActivator.start()` method throws an exception. When a bundle is persistently marked as started, subsequent executions of the framework will not only reinstall the bundle, but will start it too. From a management perspective, you deploy a configuration of your application by installing a set of bundles and activating them. Subsequent framework executions will automatically restart your application. If you stop a bundle using `Bundle.stop()`, this removes the persistently started status of the bundle, so subsequent framework executions will no longer restart the bundle, although it will still be reinstalled. This is actually another means for modifying your application's configuration.



Ch3Figure 3.5: Bundle cache during framework restarts

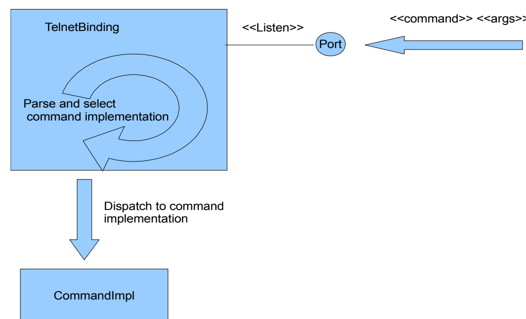
You might be wanting to ask, “What about updating and uninstalling a bundle? These must impact the bundle cache, right?” The short answer is yes, but this is not the whole answer. `Bundle.update()` and `Bundle.uninstall()` impact the bundle cache by saving a new bundle JAR file or removing an existing bundle JAR file, respectively. However, the impacts of these operations may not impact the cache immediately. We will explain these oddities when we discuss the relationship between the modularity and lifecycle layers in section [ref]. Next we will delve into the details of our shell bundle as we explore how to actually use the lifecycle layer API.

3.3 Leveraging the lifecycle API in your bundles

So far we haven't implemented very much functionality for our shell, we just created the activator to start it up and shut it down. In this section we will show how to implement the bulk of its functionality. We will use a simple command pattern to provide the executable actions to allow us to interactively install, start, stop, update, and uninstall bundles. We will even add a persistent history to keep track of previously executed commands.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

A high-level understanding of our approach might be useful before we start. The main piece is the telnet binding, which listens to the configured port for connection requests. It spawns a new thread for each connecting client. The client sends command lines to its thread, where a command line consists of a command name and the arguments for the command. The thread parses the command line, selects the appropriate command, and invokes it with any specified arguments.



Ch3Figure 3.6: TelnetBinding Overview

Commands simply process the arguments passed into them. We will not discuss the implementation of the telnet binding and the connection thread, but full source code is available in the companion code. We will dissect the command implementations to illustrate how to use `Bundle` and `BundleContext`. Ok, let's get the ball rolling by showing how we configure our bundle.

3.3.1 Configuring bundles

Our shell needs two configuration properties, one for the port and one for the maximum number of concurrent connections. In traditional Java programming, we would use the `System.getProperty()` method to retrieve them. When creating a bundle, you can use the `BundleContext` object to retrieve configuration properties instead. The main benefit of this approach is it avoids the global aspect of `System.getProperty()` and allows properties per framework instance.

The OSGi specification does not specify how to set bundle configuration properties, so different frameworks handle this differently; typically they provide a configuration file where the properties are set. However, the specification does require bundle configuration properties be backed by system properties, so you can still set use system properties in a pinch. Retrieving bundle configuration property values is standardized via the `BundleContext.getProperty()` method as shown in Listing 3.5.

Listing 3.5 Bundle configuration by example

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
package org.foo.shell;

import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;

public class Activator implements BundleContext {
  ...

  private int getPort(BundleContext context) {
    String port = context.getProperty("org.foo.shell.port");             #1
    if (port != null) {
      return Integer.parseInt(port);
    }
    return 7070;
  }

  private int getMaxConnections(BundleContext context) {
    String max = context.getProperty("org.foo.shell.connection.max");    #2
    if (max != null) {
      return Integer.parseInt(max);
    }
    return 1;
  }
}
```

This listing continues our activator implementation from Listing 3.2; in our activator we used these two methods to get configuration properties. Here at (#1) and (#2) we can see the methods actually use the `BundleContext.getProperty()` method to retrieve the properties. This method looks in the framework properties to find the value of the specified property. If it cannot find the property, it then searches the system properties, returning `null` if the property is not found. For our shell, we return default values if no configured value is found. The OSGi specification also defines some standard framework properties, shown in Table 3.1. If you need to use these standard properties, you can use the constants for them defined in the `org.osgi.framework.Constants` class.

Table 3.1 Standard OSGi framework properties

Property name	Description
`org.osgi.framework.version`	the OSGi framework version
`org.osgi.framework.vendor`	the framework implementation vendor
`org.osgi.framework.language`	the language being used; see ISO 639 for possible values
`org.osgi.framework.os.name`	the host computer operating system
`org.osgi.framework.os.version`	the host computer operating system version number

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

`org.osgi.framework.processor`	the host computer processor name

So there we have it. Our first real interaction with the OSGi framework. This is only a small part of the API you can use in your bundles, but we will cover a lot of ground in the next section, so don't worry. And for those of you thinking, "Hey, this configuration mechanism seems overly simplistic!" You are correct. There are other, more sophisticated ways to configure your bundle, but we won't discuss them until chapter [ref]. Bundle properties are the simplest method available and should only be used for properties that don't change much. In this regard, they might not be the best choice for our shell, but it just depends on what we want to achieve; for example, it makes it difficult to change the shell's port dynamically. For now, we will keep things simple, so this is sufficient.

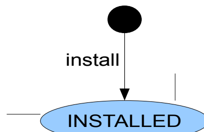
3.3.2 Deploying bundles

Each bundle installed into the framework is represented by a `Bundle` object and can be identified by its bundle identifier, location, or symbolic name. For most of the shell commands we are going to implement, we will use the bundle identifier to retrieve a `Bundle` object, since the bundle identifier is nice and concise. Since most of our commands will accept a bundle identifier as a parameter, let's look at how we can use the bundle context to access `Bundle` objects associated with other bundles using the identifier. As part of our design, we create an abstract `BasicCommand` class to define a shared method `getBundle()` to retrieve bundles by their identifier as shown below:

```
protected volatile BundleContext m_context;
...
public Bundle getBundle(String id) {
  Bundle bundle = m_context.getBundle(Long.parseLong(id.trim()));        #1

  if (bundle == null) {
    throw new IllegalArgumentException("No such bundle.");
  }
  return bundle;
}
```

All we do is call `BundleContext.getBundle()` on our context object at (#1) with the parsed bundle identifier, which is passed in as a `String`. The only special case we need to worry about is when no bundle with the given identifier exists. In such a case, we throw an exception.

INSTALL COMMAND



With this basic functionality in place we can start our first command. Listing 3.6 shows the implementation of an "install" command.

Listing 3.6 Bundle install command

```
package org.foo.shell;
```

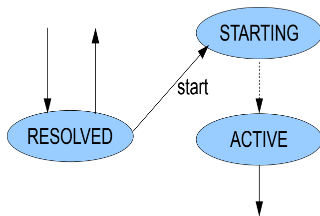
©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;

public class InstallCommand extends BasicCommand {
  public void exec(String args, PrintStream out, PrintStream err)
    throws Exception {
    Bundle bundle = m_context.installBundle(args);                      #1
    out.println("Bundle: " + bundle.getBundleId());                     #2
  }
}
```

We use `BundleContext.installBundle()` at (#1) to install a bundle. In most framework implementations the argument to `installBundle()` is conveniently interpreted as an URL in `String` form from which the bundle JAR file can be retrieved. Since the user enters the URL argument as a `String`, we can use it directly at (#1) to install the bundle. If the install succeeds, then a new `Bundle` object corresponding to the newly installed bundle is returned. The bundle will be uniquely identified by this URL, which will be used as its location. This location value will also be used in the future to determine if the bundle is already installed. If a bundle is already associated with this location value, then the `Bundle` object associated with the previously installed bundle is returned instead of installing it again. If the install operation was a success, our command outputs the installed bundle's identifier at (#2).

The bundle context also provides an overloaded `installBundle()` method for installing a bundle from an input stream. We do not show this method here, but this form of `installBundle()` accepts a location and an open input stream. When using this version of the method, the location is used purely for identification and the bundle JAR file is read from the passed in input stream. The framework is responsible for closing the input stream.

Start command



Now we have a command to install bundles, so the next operation we will want to do is start bundles. The "start" command shown in Listing 3.7 does just that.

Listing 3.7 Bundle start command

```
package org.foo.shell;

import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;

public class StartCommand extends BasicCommand {
  public void exec(String id) throws Exception {
    Bundle bundle = getBundle(id);                                      #1
    bundle.start();                                                     #2
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

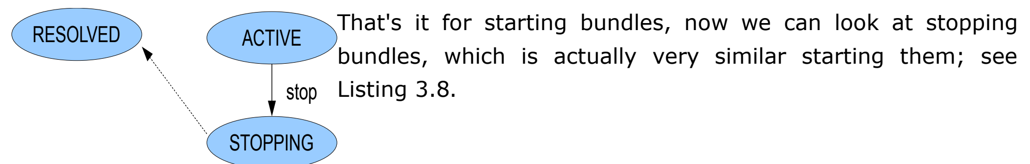
```
    }
  }
```

Again, the implementation is pretty easy. We use our method from the base command class to get the `Bundle` object associated with the user-supplied identifier at (#1), then we invoke `Bundle.start()` at (#2) to start the bundle associated with the identifier.

The result of `Bundle.start()` depends on the current state of the associated bundle. If the bundle is `INSTALLED`, then it will transition to `ACTIVE` via the `RESOLVED` and `STARTING` states. If the bundle is `UNINSTALLED`, then the method will throw an `IllegalStateException`. If the bundle is either `STARTING` or `STOPPING`, then `start()` blocks until the bundle enters either `ACTIVE` or `RESOLVED`. If the bundle is already `ACTIVE`, then calling `start()` again has no effect. A bundle must be resolved before it can be started. It is not necessary to explicitly resolve the bundle, since the specification requires the framework to implicitly resolve the bundle if it is not already resolved. If the bundle's dependencies cannot be resolved, then `start()` throws a `BundleException` and the bundle cannot be used until its dependencies are satisfied. If this happens, you will typically install additional bundles to satisfy the missing dependencies and try to start the bundle again.

If the bundle has an activator, the framework will invoke the `BundleActivator.start()` method when starting the bundle. Any exceptions thrown from the activator will result in a failed attempt to start the bundle and an exception being thrown from `Bundle.start()`. One last case where an exception may result is if a bundle tries to start itself; the specification says attempts to do so should result in an `IllegalStateException`.

STOP COMMAND



That's it for starting bundles, now we can look at stopping bundles, which is actually very similar starting them; see Listing 3.8.

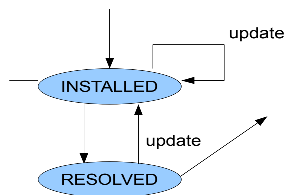
Listing 3.8 Bundle stop command

```
package org.foo.shell;

import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;

public class StopCommand extends BasicCommand {
  public void exec(String id) throws Exception {
    Bundle bundle = getBundle(id);                                    #1
    bundle.stop();                                                    #2
  }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Like starting a bundle, stopping a bundle is a simple call to Bundle.stop() at (#2) on the Bundle object retrieved from the specified identifier at (#1). Like before, we must be mindful of the bundle's state. If it is UNINSTALLED, an IllegalStateException will result. Either STARTING or STOPPING blocks until either ACTIVE or RESOLVED is reached, respectively. In the ACTIVE state, the bundle will transition to RESOLVED via the STOPPING state. If the bundle has an activator and the activator's stop() method throws an exception, then a BundleException will be thrown. Lastly, a bundle is not supposed to change its own state; trying to do so may result in an IllegalStateException. You may be wondering, "Why can't a bundle stop itself?" Good question. We will discuss this more in section xx.

Update command



Let's continue with the "update" command in Listing 3.9.

Listing 3.9 Bundle update command

```
package org.foo.shell;

import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;

public class UpdateCommand extends BasicCommand {
  public void exec(String id) throws Exception {
    Bundle bundle = getBundle(id);                                    #1
    bundle.update();                                                  #2
  }
}
```

By now you might have noticed the pattern we mentioned in the beginning. Most lifecycle operations are methods on the Bundle and BundleContext objects. The Bundle.update() method is no exception as you can see at (#2). The update() method is available in two forms: one with no parameters (shown) and one taking an input stream (not shown). Our update command uses the form without parameters at (#2), which reads the updated bundle JAR file using the original location value as a source URL. If the bundle being updated is in the ACTIVE state, then it must be stopped first, as required by the bundle lifecycle. We do not need to do this explicitly, since the framework will do it for us, but it is still good to understand that this occurs since it impacts the behavior of our applications. The update happens in either the RESOLVED or INSTALLED state and results in a new revision of the bundle in the INSTALLED state. If the bundle is in the UNINSTALLED

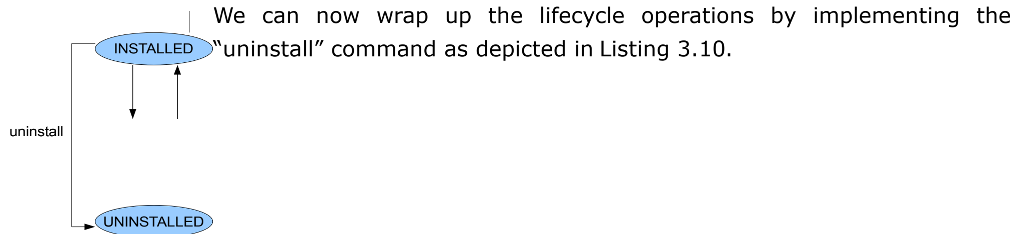
©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

state, then an IllegalStateException is thrown. As in the stop command, a bundle should not try to update itself.

THE BUNDLE-UPDATELOCATION ANTI-PATTERN

We should point out an anti-practice for updating a bundle. The OSGi specification actually provides a third option for updating bundles based on bundle metadata. A bundle may declare a piece of metadata in its bundle manifest called Bundle-UpdateLocation. If present, then the Bundle.update() with no parameters uses the update location value specified in the metadata as URL for retrieving the updated bundle JAR file. Using this approach is discouraged since it is confusing if you forget it is set and it doesn't make sense to bake this sort of information into the bundle itself.

UNINSTALL COMMAND



We can now wrap up the lifecycle operations by implementing the "uninstall" command as depicted in Listing 3.10.

Listing 3.10 Bundle uninstall command

```
package org.foo.shell;

import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;

public class UninstallCommand extends BasicCommand {
  public void exec(String id) throws Exception {
    Bundle bundle = getBundle(id);                                    #1
    bundle.uninstall();                                               #2
  }
}
```

To uninstall a bundle we call the Bundle.uninstall() method at (#2) after retrieving the Bundle object associated with the user-supplied bundle identifier at (#1). The framework will stop the bundle, if necessary. If the bundle is already UNINSTALLED, then an IllegalStateException is thrown. As with the other lifecycle operations, a bundle should not attempt to uninstall itself.

That's it. We created a telnet-based shell bundle we can use in any OSGi framework. There is one fly in the ointment. Most of our shell commands require the bundle identifier to perform their action, but how does the user know what identifier to use? We need some way to inspect the state of the framework's installed bundle set. We'll create a command for that next.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

3.3.3 Inspecting framework state

We need one more command to display information about the bundles currently installed in the framework. Listing 3.11 shows a simple implementation of a "bundles" commmand.

Listing 3.11 Bundle information example

```
package org.foo.shell;

import org.osgi.framework.Bundle;
import org.osgi.framework.BundleContext;
import org.osgi.framework.Constants;

public class BundlesCommand extends BasicCommand {
  public void exec(String args) throws Exception {
    Bundle[] bundles = m_context.getBundles();                           #1

    out.println("  ID      State      Name");

    for (Bundle bundle : bundles) {
      printBundle(
        bundle.getBundleId(),                                            #2
        getStateString(bundle.getState()),                               #3
        (String) bundle.getHeaders().get(Constants.BUNDLE_NAME),         #4
        bundle.getLocation(),                                            #5
        bundle.getSymbolicName());                                       #6
      }
    }

    private String getStateString(int state) {
      switch (state) {
        case Bundle.INSTALLED:
          return "INSTALLED";
        case Bundle.RESOLVED:
          return "RESOLVED";
        case Bundle.STARTING:
          return "STARTING";
        case Bundle.ACTIVE:
          return "ACTIVE";
        case Bundle.STOPPING:
          return "STOPPING";
        default:
          return "UNKNOWN";
    }
  }

  private void printBundle(long id, String state, String name,
    String location, String symbolicName) {...}
}
```

The implementation of this command is pretty easy too, since we only need to use `BundleContext.getBundles()` at (#1) to get an array of all bundles currently installed in the framework. The rest of the implementation loops through the returned array and prints

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

out information from each `Bundle` object. Here we print the bundle identifier (#2), lifecycle state (#3), name (#4), location (#5), and symbolic name (#6) for each bundle.

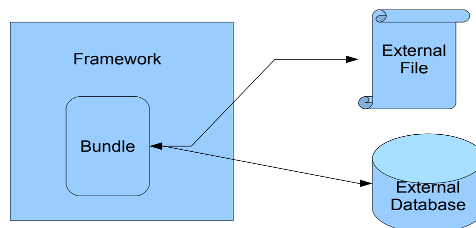
With this command in place we have everything we need for our simple shell. We can install, start, stop, update, and uninstall bundles and list the currently installed bundles. That was pretty easy, wasn't it? Think about the flexibility at your fingertips versus the amount of effort needed to create our shell. Now it is possible for you to create applications with easily deployable configurations of bundles that can be managed and evolved as necessary over time.

Before we move back to our paint program there are two final lifecycle concepts that are worth exploring in order to fully appreciate the approach we will take to make our our paint program dynamically extensible: persistence and events. We will describe them in the context of our shell example, but as you will see in the paint example in a couple of pages time they are generically useful tools to have in mind when building OSGi applications.

3.3.4 Persisting bundle state

As we mentioned in section [ref] when discussing bundle activators, the framework creates an instance of a bundle's activator class and uses the same instance for starting and subsequently stopping the bundle. An activator instance is only used once by the framework to start and stop a bundle, then it is discarded. If the bundle is subsequently restarted, then a new activator instance is created. Given this situation, how does a bundle persist state across stops and restarts? Stepping back even further, we mentioned how the framework saves installed bundles into a cache so they can be reloaded the next time the framework starts. How does a bundle persist state across framework sessions? There are several possibilities.

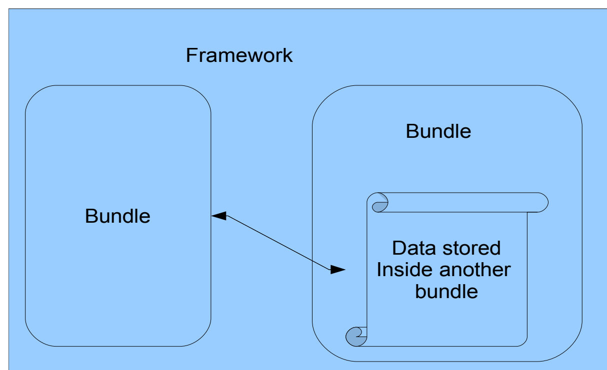
One possibility is to store the information outside the framework, such as in a database or even a file. The disadvantage of this approach is the state is not managed by the framework and may not be cleaned up when the bundle is uninstalled.



Ch3Figure 3.7: Storing state externally

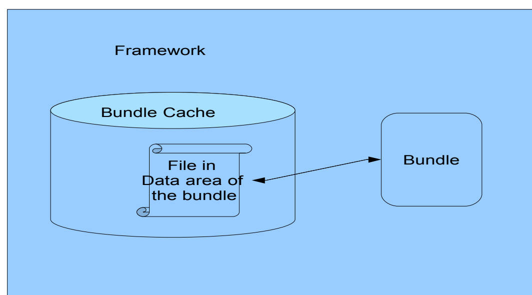
Another possibility is for a bundle to give its state to another bundle which is not being stopped, then it could get the state back after it restarts.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507



Ch3Figure 3.8: Storing state with other bundles

This approach is similar to the one offered by services, which are explained in the next chapter so we will not discuss it yet. For simplicity, it would be nice to just be able to use files, but have them managed by the framework. Such a possibility exists. The framework maintains a private data area in the file system for each installed bundle.



Ch3Figure 3.9: Storing state internally

The `BundleContext.getDataFile()` method provides access to your bundle's private data area. When using the private data area, you don't need to worry about where it is on the file system because the framework takes care of that for you, as well as cleaning up in the event of your bundle being uninstalled. For some, it may seem odd to not just directly use files to store your data; however, it would be impossible for your bundle to clean up during an uninstall. This is because a bundle is not notified when it is uninstalled. Further, this method simplifies running with security enabled since bundles can be granted permission to access their private area by the framework.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

For our shell example, we want to use the private area to persistently save our command history. Listing 3.12 shows how we want our "history" command to work; it prints the commands issued via the shell in reverse order.

Listing 3.12 Using the history command

```
-> history
bundles
uninstall 2
bundles
update 2
bundles
stop 2
bundles
start 2
bundles
install file:foo.jar
bundles
```

Listing 3.13 shows how we use the bundle's private storage area to save the command history when the bundle stops and read it in when it starts.

Listing 3.13 Bundle persistent storage example

```
package org.foo.shell;

import java.util.List;

public interface History {
  public List<String> get();
}

public class Activator implements BundleActivator {
  ...
  private void writeHistory(History history, BundleContext context) {
    List<String> list = history.get();
    File log = context.getDataFile("log.txt");                          #1
    if (log == null) {                                                  #2
      System.out.println(
        "Unable to persist history - no storage area");
    }
    if (log.exists() && !log.delete()) {
      throw new IOException("Unable to delete previous log file!");
    }
    write(list, log);
  }

  private List<String> readHistory(BundleContext context) {
    List<String> result = new ArrayList<String>();
    File log = context.getDataFile("log.txt");                          #3
    if ((log != null) && log.isFile()) {                                #4
      read(log, result);
    }
    return result;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    }
  }
```

We use `BundleContext.getDataFile()` at (#1) and (#3) to get a `File` object in our bundle's private storage area. The method takes a relative path as a `String` and returns a valid `File` object in the storage area. Once we get the `File` object, we can use it normally to create the file, make a subdirectory, or whatever we want. It is possible for a framework to return `null` when a bundle requests a file, so as you can see at (#2) and (#4) we need to handle this possibility. This can happen because the OSGi framework was designed to run on a variety of devices, some of which may not support a file system. For our shell, we just ignore it if there is no file system support, since the "history" command is non-critical funcionality.

If you want to retrieve a `File` object for the root directory of your bundle's storage area, you can call `getDataFile()` with an empty string. Your bundle is responsible for managing the content of its data area, but there is no need to worry about cleaning up when uninstalled, since the framework takes care of this.

PLAN AHEAD

One thing to keep in mind, your bundle might get updated. Due to this possibility, you should design your bundles so they properly deal with previously saved persistent state, since they may start with a private area from an older version of the bundle. The best approach is for your bundles to seamlessly migrate old state formats to new state formats if possible. One tricky issues, though, is the update lifecycle operation may also be used to downgrade a bundle. In this case, your bundle may have difficulty dealing with the newer state formats, so it is probably best if your implement your bundles to delete any existing state if they cannot understand it. Otherwise, you could always uninstall the newer bundle first, then install the older version instead of downgrading.

We could finish our "history" command, but let's try to make it a little more interesting by keeping track of what is going on inside the framework. The idea is we can not only record the issued commands, but also the impact they had on the framework. The next section shows how we can achieve this using framework's event notification mechanism.

3.3.5 Listening for events

The OSGi framework is a very dynamic execution environment. To create bundles, and ultimately applications, flexible enough to not only cope with, but leverage this dynamism you need to pay attention to run-time changes. While the lifecycle layer API provides access to a lot of information, it is not easy to poll for changes; it is much more convenient if we can be notified when changes occur. To make this possible, the OSGi framework supports two types of events: `BundleEvent`s and `FrameworkEvent`s. The former event type reports changes in the lifecycle of bundles, while the latter reports changes in the framework.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

You can use the normal Java listener pattern in your bundles to receive these events. The BundleContext object has methods to register BundleListener and FrameworkListener objects for receiving BundleEvent and FrameworkEvent notifications, respectively. Listing 3.14 shows how we implement our "history" command.

Listing 3.14 Bundle and Framework Event listener example

```
package org.foo.shell;

import java.io.PrintStream;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.osgi.framework.BundleEvent;
import org.osgi.framework.BundleListener;
import org.osgi.framework.FrameworkEvent;
import org.osgi.framework.FrameworkListener;

public class HistoryDecorator implements Command,                     #1
  History, FrameworkListener, BundleListener {
  private final List<String> m_history =
    Collections.synchronizedList(new ArrayList<String>());            #2
  private final Command m_next;

  public HistoryDecorator(Command next, List<String> history) {       #3
    m_next = next;
    m_history.addAll(history);
  }

  public void exec(String args, PrintStream out, PrintStream err)     #4
    throws Exception {
    try {
      m_next.exec(args, out, err);                                    #5
    } finally {
      m_history.add(args);                                            #6
    }
  }

  public List<String> get() {
    return new ArrayList<String>(m_history);
  }

  public void frameworkEvent(FrameworkEvent event) {                  #7
    m_history.add("\tFrameworkEvent(type=" + event.getType() +        #8
      ",bundle=" + event.getBundle() +
      ",source=" + event.getSource() +
      ",throwable=" + event.getThrowable() + ")");
  }

  public void bundleChanged(BundleEvent event) {                      #9
    m_history.add("\tBundleEvent(type=" + event.getType() +          #10
      ",bundle=" + event.getBundle() +
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
        ",source=" + event.getSource() + ")");
    }
  }
```

We use an interceptor pattern to wrap the actual commands so we can record the issued commands. Our wrapper can also record any events in the history by implementing the `BundleListener` and `FrameworkListener` interfaces at (#1). We will maintain a list of all issued commands and received events in the `m_history` member defined at (#2). Our history wrapper command forwards the command execution to the actual command at (#5) and stores it in the history list at (#6).

The wrapper implements the single `FrameworkListener.frameworkEvent()` method at (#7). Here we record the event information in the history list. The most important part of the event is its type. Framework events are of one of the following types:

- `FrameworkEvent.STARTED` – indicates the framework has performed all initialization and has finished starting up.
- `FrameworkEvent.INFO` – indicates some information of general interest in various situations.
- `FrameworkEvent.WARNING` – indicates a warning; not crucial, but may indicate a potential error.
- `FrameworkEvent.ERROR` – indicates an error; requires immediate attention.
- `FrameworkEvent.PACKAGES_REFRESHED` – indicates the framework has refreshed some shared packages; we will discuss what this means in xx.
- `FrameworkEvent.STARTLEVEL_CHANGED` – indicages the framework has changed its start level; we will discuss what this means in xx.

The wrapper implements the single `BundleListener.bundleChanged()` method at (#9). Here we also record the event information in the history list. Bundle events have one of the following types:

- `BundleEvent.INSTALLED` – indicates a bundle was installed.
- `BundleEvent.RESOLVED` – indicates a bundled was resolved.
- `BundleEvent.STARTED` – indicates a bundle was started.
- `BundleEvent.STOPPED` – indicates a bundle was stopped.
- `BundleEvent.UPDATED` – indicates a bundle was updated.
- `BundleEvent.UNINSTALLED` – indicates a bundle was uninstalled.
- `BundleEvent.UNRESOLVED` – indicates a bundle was unresolved.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

It is easy to register event listeners, BundleContext has add and remove methods for both BundleListener and FrameworkListener. Listing 3.15 shows how we add our listeners to the context.

Listing 3.15 Add bundle and framework listener

```
private void addListener(BundleContext context,
    BundleListener bundeListener, FrameworkListener frameworkListener) {
    context.addBundleListener(bundleListener);                          #1
    context.addFrameworkListener(frameworkListener);                    #2
}
```

Our example does not show how to remove the listeners, which requires calls to the removeBundleListener() and removeFrameworkListener() methods on the context. It is actually not necessary to remove the listeners, since the framework will do so automatically when our bundle is stopped; this makes sense since the bundle context is not valid once the bundle is stopped. You only need to explicitly remove your listeners if you want to stop listening to events while your bundle is active.

For the most part, the framework delivers events asynchronously. It is possible for framework implementations to deliver them synchronously, but typically they don't because it complicates concurrency handling. Sometimes you need synchronous delivery because you need to perform an action as the event is happening, so to speak. This is possible for BundleEvents by registering a listener implementing the SynchronousBundleListener interface instead of BundleListener. The two interfaces look the same, but the framework deliver events synchronously to SynchronousBundleListeners, meaning the listener is notified during the processing of the event. Synchronous bundle listeners are processed before normal bundle listeners. This allows you to take action when a certain operation is triggered; for example, you could give permissions to a bundle at the moment it is installed. The following event types are only sent to SynchronousBundleListeners:

- BundleEvent.STARTING – indicates a bundle is about to be started.
- BundleEvent.STOPPING – indicates a bundle is about to be stopped.

Synchronous bundle listeners are sometimes necessary (as we shall see in our paint example in the next section), but should be used with caution. They can lead to concurrency issues if you try to do too much within the callback; as always, keep your callbacks as short and simple as possible. In all other cases, the thread invoking the listener callback method is undefined. Events become much more important to you when you start to write more sophisticated bundles that take full advantage of the bundle lifecycle

3.3.6 How to kill yourself...

We've mentioned it numerous time: a bundle is not supposed to change its own state. But what if a bundle really wants to change its own state? Good question. This is one of the more

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

complicated aspects of the lifecycle layer and there are potentially negative issues in doing so.

The central issues is if a bundle stops itself, it finds itself in a state it should not be in. Its `BundleActivator.stop()` method has been invoked, which means its bundle context is no longer valid. Additionally, the framework has cleaned up its bookkeeping for the bundle and has released any framework facilities it was using, such as unregistering all of its event listeners. The situation is even worse if a bundle tries to uninstall itself, since the framework will likely release its class loader. In short, the bundle is in a very hostile environment and it might not be able to function properly.

Since its bundle context is no longer valid, a stopped bundle can't use the functionality provided by the framework anymore. Most method calls on an invalid bundle context will throw `IllegalStateException`s. Even if the bundle's class loader is released, this may not pose a serious issue if the bundle does not need any new classes, since the class loader will not be garbage collected until the bundle stops using it. However, we are not guaranteed to be able to load new classes if the bundle was uninstalled. In this case, the framework might have closed the JAR file associated with the bundle. Already loaded classes will continue to load, but all bets are off when attempting to load new classes.

Depending on your bundle, you could run into other issues too. If your bundle creates and uses threads, it is typically a good idea for it to wait for all of its threads to complete when its `BundleActivator.stop()` method is called. If the bundle tries to stop itself on its own thread, that same thread could end up in a cycle waiting for other sibling threads to complete. In the end, the thread waits forever. For example, our simple shell uses a thread to listen for telnet connections and then secondary threads to execute the commands issued on those connections. If one of the secondary threads attempts to stop the shell bundle itself, it ends up waiting in the shell bundle's `BundleActivator.stop()` method for the connection thread to stop all of the secondary threads. Since the calling thread is one of the secondary threads, it will end up waiting forever for the connection thread to complete. You have to be very careful of these types of situations and they are not always obvious.

Under normal circumstances, you shouldn't try to stop, uninstall, or update your own bundle. Ok, that should be enough disclaimers. Let's look at a case where you might need to do it any way. We can use our shell as an example, since it is provides a means to update bundles and it may need to update itself. What do we have to do to allow a user to update the shell bundle via the shell command line? We need to do two things to be safe:

1. Use a new thread when we stop, update, or uninstall our own bundle.
2. Do nothing in the new thread after calling stop, update, or uninstall.

We need to do this to prevent us from waiting forever for the shell thread to return when we get stopped and to avoid the potential ugliness of the hostile environment in which our thread will find itself. Listing 3.16 shows the changes to the implementation of the “stop” command to accommodate this scenario.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Listing 3.16 Example of how to stop yourself

```
package org.foo.shell;

import java.io.PrintStream;
import org.osgi.framework.Bundle;
import org.osgi.framework.BundleException;

class StopCommand extends BasicCommand {
  public void exec(String args, PrintStream out, PrintStream err)
    throws Exception {
    Bundle bundle = getBundle(args);

    if (bundle == m_context.getBundle()) {                                    #1
      new SelfStopThread(bundle).start();
    } else {
      bundle.stop();
    }
  }

  private static final class SelfStopThread extends Thread {
    private final Bundle m_self;

    public SelfStopThread(Bundle self) {
      m_self = self;
    }

    public void run() {
      try {
        m_self.stop();                                                        #2
      } catch (BundleException e) {
        // Ignore
      }
    }
  }
}
```

At (#1), we use the `BundleContext.getBundle()` method to get a reference to our own bundle representation and compare it to the target bundle. Remember there is only one instance of any given bundle so we can use referential equality. When the target is our bundle, we need to stop it using a different thread. For this reason, we create and start a new thread of type `SelfStopThread`, which will execute the `Bundle.stop()` method at (#2). There is one final point of note in this example. We changed the behavior of stopping a bundle in this case from synchronous to asynchronous. Ultimately, this shouldn't matter much, since the bundle will be stopped anyway.

We should also modify the implementation of the “update” and “uninstall” commands in the same way. If you followed our suggestion and implemented a “shutdown” command, then you will need to make changes to it along the lines of the “stop” command. Why? Because stopping the system bundle causes the framework to stop, which stops every other bundle. This means we will stop our own bundle indirectly, so we should make sure we are using a new thread.

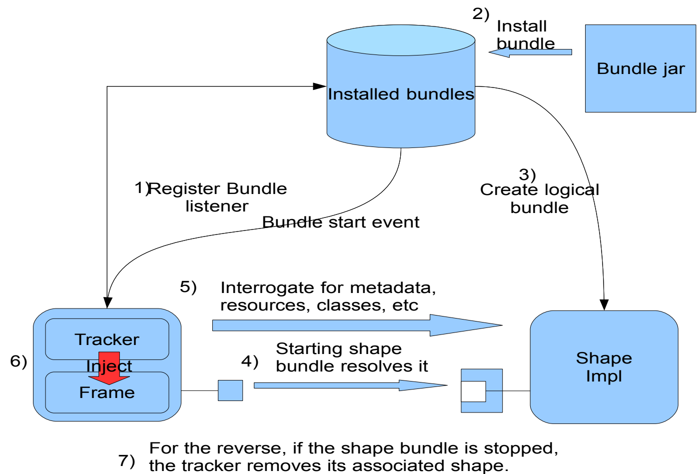
©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Hopefully, you now have a good understanding of what is possible with OSGi's lifecycle layer. Now we will apply all of this knowledge to our paint program.

3.4 Dynamically extending the paint program

Let's look at how the individual parts of the lifecycle layer can be used to dynamically extend the paint program. As you recall from last chapter, we first converted a non-modular version of it into a modular one using an interface-based programming approach for the architecture. This is great since we can reuse the resulting bundles with only a minimal amount of extra work. In fact, the bundles containing the shape implementations do not need to change at all, except for some additional metadata in their manifest. We just need to modify the paint program to make it possible for shapes to be added and removed at execution time.

THE EXTENDER PATTERN



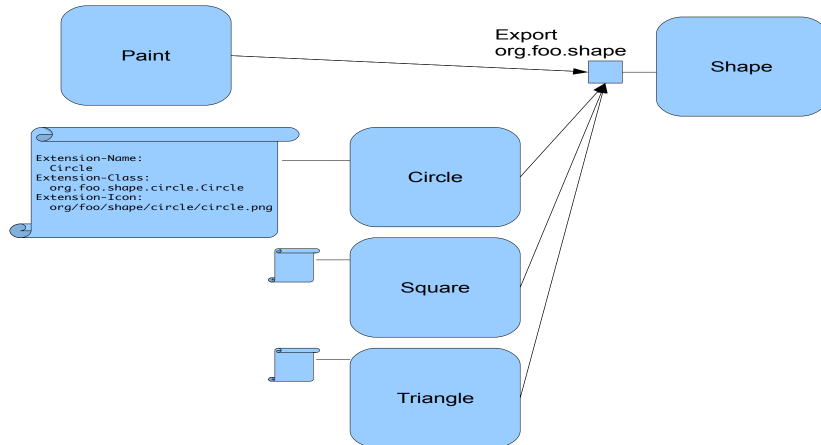
Ch3Figure 3.10: Extender Pattern Overview

The approach we are going to take is a well-known pattern in the OSGi world, called the extender pattern. The main idea behind the extender pattern is to model dynamic extensibility on the lifecycle events (e.g., installing, resolving, starting, stopping, etc.) of other bundles. Typically, some bundle in the application acts as the "extender". It listens for bundles being started and/or stopped. When a bundle is started, the extender probes it to see if it is an "extension" bundle. The extender looks in the bundle's manifest (using `Bundle.getHeaders()`) or the bundle's content (using `Bundle.getEntry()`) for specific metadata it recognizes. If the bundle does contain an extension, the extension is described by the metadata. The extender reads the metadata and performs the necessary tasks on behalf of the extension bundle to integrate it into the application. The extender also listens for the bundles to be stopped, in which case it removes the extension from

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

the application. That's the general description of the extender pattern, so let's look at how we will use it in the paint program.

We will treat our shape implementations as extensions. The extension metadata will be contained in the bundle manifest and will describe which class implements the shape contained in the shape bundle. The extender will use this information to load the shape class from the bundle, instantiate it, and inject it into the application. If a shape bundle is stopped, the extender will remove it from the application.



Ch3Figure 3.11: Paint Program as Extender Pattern

So let's dive in and start converting the application. The first thing we need to do is to define the extension metadata for shape bundles to describe their shape implementation. In Listing 3.17 we add a couple of constants to the `SimpleShape` interface for extension metadata property names; it is not strictly necessary to add these, but it is good programming practice to use constants.

Listing 3.17 Defining the SimpleShape interface

```
package org.foo.shape;

import java.awt.Graphics2D;
import java.awt.Point;

public interface SimpleShape {
  public static final String NAME_PROPERTY = "Extension-Name";            #A
  public static final String ICON_PROPERTY = "Extension-Icon";            #B
  public static final String CLASS_PROPERTY = "Extension-Class";          #C

  public void draw(Graphics2D g2, Point p);                                #D
}
```

#A The name of shape

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

#B Bundle resource file for the shape's icon
#C Bundle class name for the shape's class
#D Method to draw shape on the canvas

From these constants, it is fairly straightforward to see how we will describe a specific shape implementation. We only need to know the name, an icon, and the class implementing the shape. As an example, for the circle implementation we add the following entries to its bundle manifest:

```
Extension-Name: Circle
Extension-Icon: org/foo/shape/circle/circle.png
Extension-Class: org.foo.shape.circle.Circle
```

This metadata is arbitrary, we've just chosen them for our paint program. The name is just a string, while the icon and class refer to a resource file and a class inside the bundle JAR file, respectively. We add similar metadata to the manifests of all shape implementation bundles, which converts them all to extensions. Next, we need to tweak the architecture of the paint program to make it to cope with dynamic addition and removal of shapes. Figure 3.12 captures the updated design.

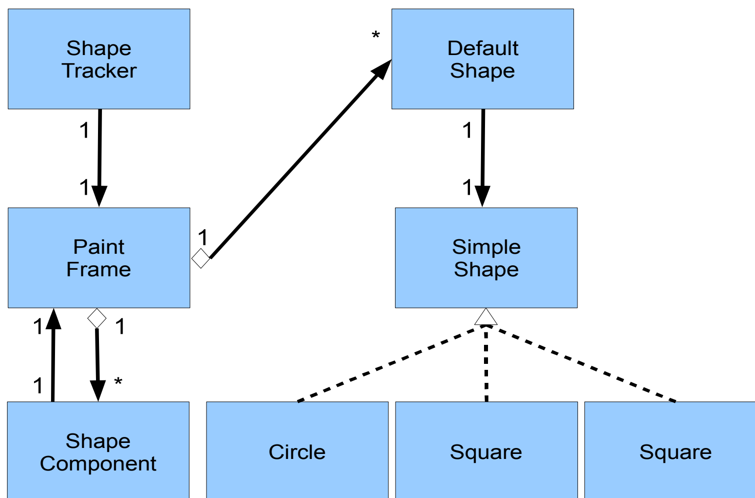


Figure 3.12 Dynamic paint program class relationships

Comparing the new design to the original, we added two new classes: `ShapeTracker` and `DefaultShape`. They help us dynamically adapt the paint frame to deal with `SimpleShape` implementations dynamically appearing and disappearing. In a nutshell, the `ShapeTracker` is used to track when extension bundles start or stop, in which case it adds or removes `DefaultShape`s to/from the `PaintFrame`, respectively.

The concrete implementation of the `ShapeTracker` is actually a subclass of another class, called `BundleTracker`. The latter class is a generic class for tracking when bundles

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

are started or stopped. Since BundleTracker is somewhat long, we will divide it across multiple listings; the first part is shown in Listing 3.18.

Listing 3.18 `BundleTracker` class declaration and constructor

```
package org.foo.paint;

import java.util.*;
import org.osgi.framework.*;

public abstract class BundleTracker {
  final Set m_bundleSet = new HashSet();
  final BundleContext m_context;
  final SynchronousBundleListener m_listener;
  boolean m_open;

  public BundleTracker(BundleContext context) {                          #1
    m_context = context;
    m_listener = new SynchronousBundleListener() {                       #2
      public void bundleChanged(BundleEvent evt) {                       #3
        synchronized (BundleTracker.this) {
          if (!m_open) {                                                 #4
            return;
          }
          if (evt.getType() == BundleEvent.STARTED) {
            if (!m_bundleSet.contains(evt.getBundle())) {
              m_bundleSet.add(evt.getBundle());                          #5
              addedBundle(evt.getBundle());                              #6
            }
          } else if (evt.getType() == BundleEvent.STOPPING) {
            if (m_bundleSet.contains(evt.getBundle())) {
              m_bundleSet.remove(evt.getBundle());                       #7
              removedBundle(evt.getBundle());                            #8
            }
          }
        }
      }
    };
  }
```

The bundle tracker is constructed with a BundleContext object at (#1), which is used to listen for bundle lifecycle events. The tracker uses a `SynchronousBundleListener` at (#2) for listening to events because a regular `BundleListener` doesn't get notified when a bundle enters the `STOPPING` state, only `STOPPED`. We need to react on the `STOPPING` event instead of the `STOPPED` event because it is still possible to use the stopping bundle since it hasn't actually been stopped yet; a potential subclass might need to do this if it needed to access the stopping bundle's `BundleContext` object. The bundle listener's single method is implemented at (#3), where it makes sure the tracker is tracking bundles at (#4). If so, for started events it adds the associated bundle to its bundle list at (#5) and invokes the abstract `addedBundle()` method at (#6). Likewise, for stopping events it removes the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

bundle from its bundle list at (#7) and invokes abstract removedBundle() method at (#8). Listing 3.19 shows the next portion of the BundleTracker.

Listing 3.19 Opening and using a BundleTracker

```
public synchronized void open() {                                    #1
  if (!m_open) {
    m_open = true;
    m_context.addBundleListener(m_listener);                         #2
    Bundle[] bundles = m_context.getBundles();
    for (int i = 0; i < bundles.length; i++) {
      if (bundles[i].getState() == ACTIVE) {
        m_bundleSet.add(bundles[i]);                                 #3
        addedBundles(bundles[i]);                                    #4
      }
    }
  }
}

public synchronized Bundle[] getBundles() {                          #5
  return (Bundle[]) m_bundleSet.toArray(
    new Bundle[m_bundleSet.size()]);
}

protected abstract void addedBundle(Bundle bundle);                  #6

protected abstract void removedBundle(Bundle bundle);                #7
```

To start a BundleTracker instance tracking bundles, you must invoke its open() method at (#1). This methods registers a bundle event listener at (#2) and processes any existing ACTIVE bundles by adding them to its bundle list at (#3) and invoking the abstract addedBundle() method at (#4). The getBundles() method at (#5) provides access to the current list of active bundles being tracked. Since BundleTracker is abstract, subclasses must provide implementations of addedBundle() at (#6) and removedBundle() at (#7) to perform custom processing of added and removed bundles, respectively. The last portion of the BundleTracker is in Listing 3.20.

Listing 3.20 Disposing of a BundleTracker

```
public synchronized void close() {                                   #1
  if (m_open) {
    m_open = false;
    m_context.removeBundleListener(m_listener);                      #2
    Bundle[] bundles = (Bundle[]
      m_bundleSet.toArray(new Bundle[m_bundleSet.size()]);
    for (int i = 0; i < bundles.length; i++) {
      if (m_bundleSet.remove(bundles[i])) {                          #3
        removedBundle(bundles[i]);                                   #4
      }
    }
  }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
}
```

Calling `BundleTracker.close()` at (#1) stops it from tracking bundles. It will remove its bundle listener at (#2) and remove each currently tracked bundle from its bundle list at (#3) and invoke the abstract `removedBundle()` method at (#4).

STANDARDIZING BUNDLE TRACKERS

The need to track bundles is a very useful building block. In fact, it is so useful that the OSGi Alliance decided to create a standard `BundleTracker` for the R4.2 specification. The R4.2 `BundleTracker` is more complicated than the one presented here, but it follows the same basic principles; we show how to use it in chapter [ref].

Now that we know how the `BundleTracker` works, we return to our subclass of it, called `ShapeTracker`. The heart of this subclass is the `processBundle()` method depicted in Listing 3.21, which processes added and removed bundles.

Listing 3.21 Processing shapes in `ShapeTracker`

```
private void processBundle(int action, Bundle bundle) {
  Dictionary dict = bundle.getHeaders();
  String name = (String) dict.get(SimpleShape.NAME_PROPERTY);          #1
  if (name == null) {
    return;
  }

  switch (action) {
    case ADDED:                                                          #2
      String iconPath = (String) dict.get(SimpleShape.ICON_PROPERTY);
      Icon icon = new ImageIcon(bundle.getResource(iconPath));
      String className = (String) dict.get(SimpleShape.CLASS_PROPERTY);
      m_frame.addShape(name, icon,
        new DefaultShape(m_context, bundle.getBundleId(), className));
      break;
    case REMOVED:                                                        #3
      m_frame.removeShape(name);
      break;
  }
}

Dictionary dict = bundle.getHeaders();
String name = (String) dict.get(SimpleShape.NAME_PROPERTY);
if (name == null) {
  return;
}
```

ShapeTracker overrides `BundleTracker`'s `addedBundle()` and `removedBundle()` abstract methods to invoke `processBundle()` in either case. We can see at (#1), we determine if the bundle is an extension by probing its manifest for the `Extension-Name` property. Any bundle without this property in its manifest is simply ignored. If the bundle being added contains a shape, the code at (#3) grabs the metadata from the bundle's

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

manifest headers and adds the shape to the paint frame wrapped as a `DefaultShape`. For the icon metadata, we use `Bundle.getResource()` to load it. If the bundle being removed contains a shape, we remove the shape from the paint frame at (#3).

The `DefaultShape`, shown in Listing 3.22, serves two purposes. It implements the `SimpleShape` interface and is responsible for lazily creating the actual shape implementation using the `Extension-Class` metadata. It also serves as a placeholder for the shape if and when the shape is removed from the application. We did not have to deal with this situation in the original paint program, but now shape implementations can appear or disappear at any time when bundles are installed, started, stopped, and uninstalled. In such situations, the `DefaultShape` draws a placeholder icon on the paint canvas for any departed shape implementations.

Listing 3.22 DefaultShape example

```
class DefaultShape implements SimpleShape {
  private SimpleShape m_shape;
  private ImageIcon m_icon;
  private BundleContext m_context;
  private long m_bundleId;
  private String m_className;

  public DefaultShape() {}                                              #A

  public DefaultShape(BundleContext context, long bundleId,             #B
    String className) {                                                 #B
    m_context = context;                                                #B
    m_bundleId = bundleId;                                              #B
    m_className = className;                                            #B
  }

  public void draw(Graphics2D g2, Point p) {
    if (m_context != null) {                                            #C
      try {                                                             #C
        if (m_shape == null) {                                          #C
          Bundle bundle = m_context.getBundle(m_bundleId);              #C
          Class clazz = bundle.loadClass(m_className);                  #C
          m_shape = (SimpleShape) clazz.newInstance();                  #C
        }                                                               #C
        m_shape.draw(g2, p);                                            #C
        return;                                                         #C
      } catch (Exception ex) {}                                         #C
    }

    if (m_icon == null) {                                               #D
      try {                                                             #D
        m_icon = new ImageIcon(this.getClass().getResource(             #D
          "underc.png"));                                               #D
      } catch (Exception ex) {                                          #D
        ex.printStackTrace();                                           #D
        g2.setColor(Color.red);                                         #D
        g2.fillRect(0, 0, 60, 60);                                      #D
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
            return;                                              #D
          }                                                      #D
        }                                                        #D
        g2.drawImage(m_icon.getImage(), 0, 0, null);            #D
      }
    }
```

#A default constructor to create placeholder shape
#B constructor with extension data
#C create extension if any and delegate
#D draw default image if no extension

In summary, when the paint application is started, its activator creates and opens a `ShapeTracker`. This tracks started and stopping bundles, interrogating them for extension metadata. For every started extension bundle, it adds a new `DefaultShape` for the bundle to the paint frame, which creates the extension if needed using the extension metadata. When the bundle stops, the `ShapeTracker` removes the shape from the paint frame. When a drawn shape is no longer available, the `DefaultShape` is used to draw a placeholder shape on the canvas instead. If the departed shape reappears, the placeholder is removed and the real shape is drawn on the canvas again.

Now we have a dynamically extensible paint program, as was demonstrated at the beginning of the chapter in section [ref]. While we didn't show the activator we added to the paint program, it is reasonably simply and only creates the framework and shape tracker on start and disposes of them on stop. Overall, this is a good example of how easy it is make a modularized application take advantage of the lifecycle layer to make it dynamically extensible. It wasn't that difficult, was it? What we're still missing is a discussion about how the lifecycle and modularity layers interact with each other, which we will get into next

3.5 Lifecycle and modularity

There is a two-way relationship between OSGi's lifecycle and modularity layers. The lifecycle layer is used to manage which bundles are actually installed into the framework, which obviously impacts how the modularity layer resolves dependencies among bundles. The modularity layer uses the metadata in bundles to make sure all their dependencies are satisfied before they can be used. This symbiotic relationship creates a chicken and egg situation when you want to use your bundles; to use a bundle you have to install it, but to install a bundle you must be a bundle to use the lifecycle layer to install it. This close relationship is also obvious in how the framework resolves bundle dependencies, especially when bundles are dynamically installed and/or removed. Let's explore this relationship by first looking into bundle dependency resolution.

3.5.1 Resolving Bundles

The actual act of resolving of a bundle happens at the discretion of the framework as long as it happens before any classes are loaded from the bundle. Often when resolving a given bundle, the framework will end up resolving another bundle to satisfy a dependency of the original bundle. This can lead to cascading dependency resolution, because in order for the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

framework to use a bundle to satisfy the requirements of another bundle, the satisfying bundle too must be resolved and so on. Since the framework resolves dependencies when needed, it is possible for you to mostly ignore transitioning bundles to the RESOLVED state; you can just start a bundle and know the framework will resolve it before starting it, if possible. This is great compared to the standard Java way, where you can run into missing dependencies at any point during the lifetime of your application.

But what if you want to make sure a given bundle resolves correctly? For example, maybe you want to know in advance whether an installed bundle can be started. In this case, there is a way to ask the framework to resolve the bundle directly, but it is not a method on Bundle like most other lifecycle operations. Instead, you use the Package Admin Service. The Package Admin Service is represented as an interface and is depicted in Listing 3.23.

Listing 3.23 Package Admin Service interface

```
public interface PackageAdmin {
  static final int     BUNDLE_TYPE_FRAGMENT   = 0x00000001;
  Bundle getBundle(Class clazz);
  Bundle[] getBundles(String symbolicName, String versionRange);
  int getBundleType(Bundle bundle);
  ExportedPackage getExportedPackage(String name);
  ExportedPackage[] getExportedPackages(Bundle bundle);
  ExportedPackage[] getExportedPackages(String name);
  Bundle[] getFragments(Bundle bundle);
  RequiredBundle[] getRequiredBundles(String symbolicName);
  Bundle[] getHosts(Bundle bundle);
  void refreshPackages(Bundle[] bundles);
  boolean resolveBundles(Bundle[] bundles);
}
```

You can explicitly resolve a bundle with the resolveBundles() method, which takes an array of bundles and returns a boolean flag indicating whether the bundles could be resolved or not. The Package Admin Service can actually do a bit more than resolving bundles and it is a fairly important part of the framework; it also supports the following operations among others:

- Determine which bundle owns a particular class – In rare circumstances you might need to know which bundle owns a particular class; you can accomplish this with the getBundle() method, which takes a Class and returns the Bundle to which it belongs.
- Introspect how the framework resolved bundle dependencies – You can use the getExportedPackage() family of methods to find out which bundles are importing a given package, while other method inspect other types of dependencies we will not talk about until chapter 5, such as getRequiredBundles() and getFragments().
- Refresh the dependency resolution for bundles – Since the framework installed set of bundles can evolve over time, sometimes you need to have the framework recalculate bundle dependencies. You can do this with the refreshBundles() method.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The most important feature of the Package Admin Service is not the ability to resolve bundles or introspect dependencies, it is the ability to refresh bundle dependencies, which is another tool needed for managing your bundles. But before we get into the details of refreshing bundles, let's finish our discussion of explicitly resolving bundles.

To demonstrate how to use the Package Admin Service to explicitly resolve a bundle, we will create a new "resolve" command for our shell to instigate bundle resolution, as shown in Listing 3.24.

Listing 3.24 Resolve Bundle example

```
package org.foo.shell;

import java.io.PrintStream;
import java.util.*;
import org.osgi.framework.Bundle;
import org.osgi.service.packageadmin.PackageAdmin;

public class ResolveCommand extends BasicCommand {

  public void exec(String args, PrintStream out, PrintStream err)
    throws Exception {
    if (args == null) {
      getPackageAdminService().resolveBundles(null);                    #1
    } else {
      List<Bundle> bundles = new ArrayList<Bundle>();
      StringTokenizer tok = new StringTokenizer(args);
      while (tok.hasMoreTokens()) {
        bundles.add(getBundle(tok.nextToken()));                        #2
      }
      getPackageAdminService().resolveBundles(bundles.toArray(new       #3
        Bundle[bundles.size()]));
    }
  }

  private PackageAdmin getPackageAdminService() {...}                   #4
}
```

We won't discuss the details of how you obtain the Package Admin Service until the next chapter; for now, we will just use the `getPackageAdminService()` method at (#4). If our "resolve" command is executed with no arguments, then we invoke `resolveBundles()` with `null` at (#1), which will cause the framework to attempt to resolve all unresolved bundles. Otherwise, we parse the argument as a list of whitespace-separated bundle identifiers. For each identifier we get its associated `Bundle` object and add it to a list at (#2). Once we have retrieved the complete list of bundles, we pass them in as an array to `resolveBundles()` at (#3). The framework attempts to resolve any unresolved bundles of those specified.

It is worthwhile to understand the framework may resolve additional bundles to those that were specified. The specified bundles are the root of the framework's resolve process;

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

the framework will resolve any additional unresolved bundles necessary to resolve the specified roots.

Resolving a bundle is a fairly easy process, since the framework does all of the hard work for us. You'd think that'd be it. As long as your bundle's dependencies are resolved you have nothing to worry about, right? It turns out, the dynamic nature of the bundle lifecycle makes this not the case. Sometimes you need have the framework recalculate a bundle's dependencies. You probably are wondering, “Why?” We will tell you all about it in the next section.

3.5.2 Refreshing bundles

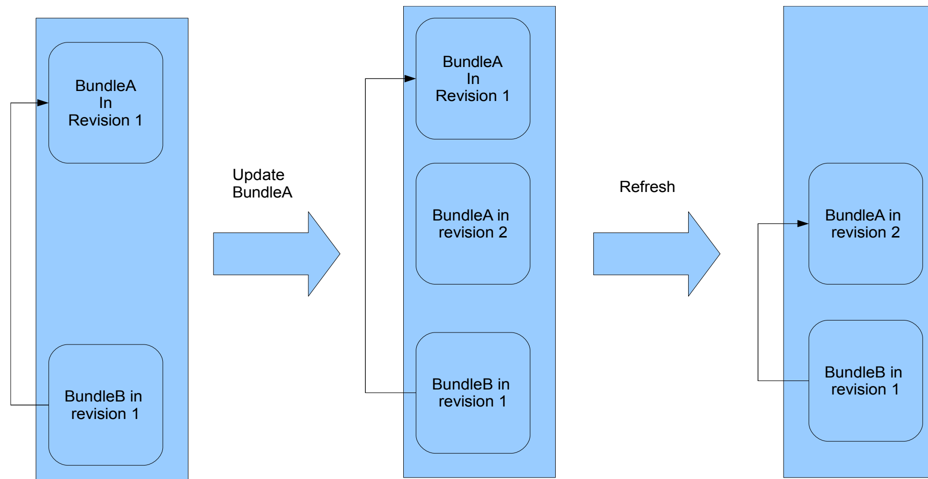
The lifecycle layer allows you to deploy and manage your application's bundles. Up until now we have focused on installing, resolving, and starting your bundles, but there are other interesting bundle lifecycle operations. How about updating or uninstalling a bundle? In and of themselves, these operations are as conceptually simple as the other lifecycle operations. We certainly understand what it means to update or uninstall a bundle. The details are a little more complicated. When you update or uninstall a resolved bundle, you stand a good chance of disrupting your system. This is the place where you can really start to see the impacts of the framework's dynamic lifecyce management.

The simple case is updating or uninstalling a self-contained bundle. In this case, the disruption is limited to the specific bundle. Even if the bundle imports packages from other bundles, the disruption is limited to the specific bundle being updated or uninstalled. In either case, the framework stops the bundle if it is active. In the case of updating, the framework updates the bundle's content and restarts it if it was previously active. Complications arise if other bundles depend on the bundle being updated or uninstalled. Such dependencies can cause a cascading disruption to your application, if the dependent bundles also have bundles depending on them.

Why do dependencies complicate the issue? Consider updating a given bundle. Other dependent bundles have potentially loaded classes from the old version of the bundle. They cannot just start loading classes from the new version of the bundle, because they would see old versions of the classes they already loaded mixed with new versions of classes. This would be inconsistent. In the case of an uninstalled bundle, the situation is more dire, since we cannot just pull the rug out from under the dependent bundles.

It is worthwhile to limit the disruptions caused by bundle updates or uninstalls. The framework provides such control by making updating and uninstalling bundles a two-step process. Conceptually, the first step prepares the operation and the second step, called refreshing, enacts its. Refreshing actually recalculates the dependencies of the impacted bundles. How does this help? It allows you to control when the changeover to the new bundle version or removal of a bundle occurs for updates and uninstalls, respectively.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507



Ch3Figure 3.13: Updating and Refreshing bundles

We say this is a two-step process, but what is happening in the first step? For updates, the new bundle version is put in place, but the old version is still kept around so bundles dependent on it can continue loading classes from it. You might be thinking to yourself, "Does this mean there are two versions of the bundle installed at the same time?" Effectively, the answer to this is, "yes." And each time you perform an update without a refresh, you are introducing yet another version. For uninstalls, the bundle is removed from the installed list of bundles, but it is not actually removed from memory. Again, the framework keeps it around so dependent bundles can continue to load classes from it.

For example, if we imagine we want to update a set of bundles, it would be fairly inconvenient if the framework refreshed all dependent bundles after each individual update. With this two-step approach, we can update all bundles in our set and then trigger one refresh of the framework at the end. You can experience a similar situation if you install a bundle providing a newer version of a package. Existing resolved bundles importing an older version of the package will not be automatically rewired to the new bundle unless they are refreshed. Again, it is nice to be able to control the point of time when this happens. It is actually a fairly common scenario when updating your application that some of your bundles get updated, some get uninstalled, and some get installed.

The way to trigger a refresh is to use the Package Admin Service again. To illustrate how to use it, we add a "refresh" command to our shell as shown in Listing 3.25.

Listing 3.25 Refresh command example

```
package org.foo.shell;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
import java.io.PrintStream;
import java.util.*;
import org.osgi.framework.Bundle;
import org.osgi.service.packageadmin.PackageAdmin;

public class RefreshCommand extends BasicCommand {

  public void exec(String args, PrintStream out, PrintStream err)
    throws Exception {
    if (args == null) {
      getPackageAdminService().refreshPackages(null);                    #1
    } else {
      List<Bundle> bundles = new ArrayList<Bundle>();
      StringTokenizer tok = new StringTokenizer(args);
      while (tok.hasMoreTokens()) {
        bundles.add(getBundle(tok.nextToken()));                         #2
      }
      getPackageAdminService().refreshPackages(                          #3
        bundles.toArray(new Bundle[bundles.size()]));
    }
  }

  private PackageAdmin getPackageAdminService() {...}                    #4
}
```

Just like in the “resolve” command, we rely on our magic method to get the Package Admin Service at (#4.) We use the `PackageAdmin.refreshPackages()` method to refresh bundles. If no arguments are given to the command, then we simply pass in `null` to the Package Admin Service at (#1). This results in the framework refreshing all previously updated and uninstalled bundles since the last refresh. This captures the update and uninstall cases we presented above, but it doesn't help us with the rewiring case. We achieve that by passing in the specific bundles we want refreshed. For this case, the “refresh” command accepts an argument of whitespace-separated bundle identifiers. We parse their identifiers out of the supplied argument, retrieve their associated `Bundle` object, and add them to a list to be refreshed at (#2). We then pass in the array of bundles to refresh to the Package Admin Service at (#3).

The `PackageAdmin.refreshPackages()` method updates or removes packages exported by the bundles being refreshed. The method returns to the caller immediately and performs the following steps on a separate thread:

1. First, it computes the graph of affected dependent bundles starting from the specified bundles (or from all updated or uninstalled bundles if `null` is specified). Any bundle wired to a package currently exported by a bundle in the graph is added to the graph. The graph is fully constructed when there is no bundle outside the graph wired to a bundle in the graph.
2. Each bundle in the graph in the `ACTIVE` state is stopped, moving it to the `RESOLVED` state.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

3. Then each bundle in the graph in the `RESOLVED` state, including those which were stopped, is unresolved and moved to the `INSTALLED` state; this means the bundles' dependencies are no longer resolved.
4. Each bundle in the graph in the `UNINSTALLED` state is removed from the graph and completely removed from the framework (i.e., is free to be garbage collected). Now we are back to a fresh starting state for the affected bundles.
5. For the remaining bundles in the graph, the framework restarts any previously `ACTIVE` bundles, which resolves them and any bundles on which they depend.
6. When everything is done, the framework fires an event of type `FrameworkEvent.PACKAGES_REFRESHED`.

It is possible, as a result of the previous steps, some of the previously `ACTIVE` bundles can no longer be resolved; maybe a bundle providing a required package was uninstalled. In such cases, or for any other errors, the framework fires an event of type `FrameworkEvent.ERROR`.

Listing 3.26 depicts a shell session showing how the "resolve" and "refresh" commands are used in combination to manage a system.

Listing 3.26 Example resolve and refresh commmand usage

```
-> install file:foo.jar
Bundle: 2
-> bundles
  ID      State       Name
[   0] [   ACTIVE] System Bundle
                   Location: System Bundle
                   Symbolic-Name: system.bundle
[   1] [   ACTIVE] Simple Shell
                   Location: file:org.foo.shell-1.0.jar
                   Symbolic-Name: org.foo.shell
[   2] [INSTALLED] Foo Bundle
                   Location: file:foo.jar
                   Symbolic-Name: org.foo.foo
-> resolve 2                                                    #1
-> bundles
  ID      State       Name
[   0] [   ACTIVE] System Bundle
                   Location: System Bundle
                   Symbolic-Name: system.bundle
[   1] [   ACTIVE] Simple Shell
                   Location: file:org.foo.shell-1.0.jar
                   Symbolic-Name: org.foo.shell
[   2] [ RESOLVED] Foo Bundle
                   Location: file:foo.jar
                   Symbolic-Name: org.foo.foo
-> refresh 2                                                    #2
-> bundles
  ID      State       Name
[   0] [   ACTIVE] System Bundle
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
                      Location: System Bundle
                      Symbolic-Name: system.bundle
[    1] [    ACTIVE] Simple Shell
                      Location: file:org.foo.shell-1.0.jar
                      Symbolic-Name: org.foo.shell
[    2] [INSTALLED] Foo Bundle
                      Location: file:foo.jar
                      Symbolic-Name: org.foo.foo
```

We install a bundle and resolve it using the "resolve" command at (#1), which transitions it to the `RESOLVED` state. Using the "refresh" command at (#2), we transition it back to the `INSTALLED` state. At this point, we've achieved a lot in understanding our understanding of the lifecycle layer, but before we can finish there are still some nuances to explain when it comes to updating bundles. Let's get to it.

3.5.3 When updating is not really updated

One of the gotchas many people run into when updating a bundle is it may or may not be using its new classes after the update operation. When we said previously that updating a bundle is a two-step process, where the first step prepares the operation and the second step enacts it; well, this is not entirely accurate when you update a bundle. The specification says the framework should actually enact the update immediately, so after the update the bundle should theoretically be using its new classes, but it does not necessarily start using them immediately. So in some situations after updating a bundle new classes are used and in other situation old classes used. Sounds confusing, doesn't it? It is. Why was it defined this way? Why not just wait to enact the new revision completely until a refresh?

The answer, as you might guess, is historical. The original R1 specification defined the update operation to actually update a bundle. End of story. There was no Package Admin Service. With experience it became clear clear that the specified definition of update was insufficient. There were too many details left for framework implementations to decide, such as when to dispose of old classes and start using new class. This led to inconsistencies, which made it difficult to manage the bundle lifecycle across different framework implementations. This situation resulted in the introduction of the Package Admin Service in the R2 specification to resolve the inconsistencies around update once and for all. Unfortunately, the original behavior of update was left intact, due to backwards compatibility concerns. These concerns leave us with the less than clean approach to bundle update that we have today, but at least it is fairly consistent across framework implementations.

Back to the issue of an updated bundle sometimes using old or new classes, as arcane as it may be, there actually is a way to understand what is going. Whether the new or the old classes of your bundle are used after an update depends on two factors:

1. If the classes are from a private package or an exported package.
2. If the classes are from an exported package, whether or not they are being used by another bundle.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Regarding the first factor, if the classes come from a private bundle package (i.e., it is not exported), then the new classes will become available immediately no matter what. If they are from an exported package, then their visibility depends on whether other bundles are using them. If no other bundles are using the exported packages, then the new classes will become available immediately. The old version of the classes are no longer needed. On the other hand, if any other bundles are using the exported packages, then the new classes will not become available immediately, since the old version is still required. In this case, the new classes will not be made available until the `PackageAdmin.refreshPackages()` method is called.

There is yet another nuance to this. In chapter 5 we will learn that bundles can also import the same packages they export. If a bundle imports a package it exports and the imported package from the the updated bundle matches the exported package from the old version, then the updated bundle will actually have its import wired to the old exported packages. This might work out well in some cases where you are just fixing a bug in a private package, for example. However, it potentially does lead to odd situations, since the updated bundle is using new versions of private classes along side old versions of exported classes. If you need to avoid this, then you should specify version ranges when your bundle imports its own packages to avoid the situation.

If the updated bundle imports its own package, but the import doesn't match the old version of the exported package, we have a different situation. This situation is similar to if the bundle only exports the package. In this case, the new classes from the exported packages become available immediately to the updated exporting bundle and for future resolves of other bundles, but not to existing importing bundles, which continue to see the old version. This situation generally requires `PackageAdmin.refreshPackages()` to bring the bundles back to a useful state.

Some of these issues can be avoided through interface-based programming and bundle partitioning. For example, if you can separate shared API (i.e., the API through which bundles interact) into interfaces and you place those interfaces into a separate set of packages contained in a separate bundle, then you can sometimes simplify this situation. In such a setup, both the client bundles and the bundles implementing the interfaces will have dependencies on the shared API bundle, but not each other. In others, we limit the coupling between clients and the providers of the functionality.

3.6 Summary

In this chapter we have seen that whether your desire is to simply deploy the bundles needed to execute your application or to create a sophisticated auto-adaptive system, the lifecycle layer provides you with everything you need. Let's review what we've learned:

- A bundle can only be used by installing it into a running instance of the OSGi framework.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

- The lifecycle layer API is composed of the main interfaces: `BundleActivator`, `BundleContext`, and `Bundle`.
- A `BundleActivator` is how a bundle hooks into the lifecycle layer to become lifecycle-aware, which allows it to gain access to all framework facilities for inspecting and modifying the framework state at execution time.
- The framework associates a lifecycle state with each installed bundle and the `BundleContext` and `Bundle` lifecycle interfaces make it possible to transition bundles though these states at execution time.
- Monitoring bundle lifecycle events is a form of dynamic extensibility available in the OSGi framework based on the dynamically changing installed set of bundles (a.k.a., the extender pattern).
- The lifecycle and module layers have a close relationship, which is witnessed when bundles are updated and uninstalled. The Package Admin Service is used to manage this interaction.

Now we will move onto the next layer of the OSGi framework, which is the service layer. Services promote interface-based programming among our bundles and provide another form of dynamic extensibility.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

4
Studying Services

So far we've seen two layers of the OSGi framework. The modularity layer helped us separate our application into well-defined, re-usable bundles, and the lifecycle layer built on the modularity layer to help us manage and evolve our bundles over time. Now we're going to make things even more dynamic with the third and final layer of OSGi: services!

We start off this chapter with a general discussion about services to make sure we are all thinking about the same thing. We then look at when you should (and shouldn't) use services and walk through an example to demonstrate the OSGi service model. At this point, you should understand the basics, so we'll take a closer look at how best to handle the dynamics of OSGi services, including common pitfalls and how to avoid them.

With these techniques in mind we'll update our ongoing paint program to use services and see how the service layer relates to the modularity and lifecycle layers. We conclude with a review of standard OSGi framework services and learn more about the “compendium”. As you can see we have many useful and interesting topics to cover, so let's get started and talk about services!

4.1 The what, why and when of services

Before looking at OSGi services, we should first try to agree on what we mean by a service, since the term can mean different things to different people depending on your background. Once we know the “what”, we also need to know why and when to use services, so we'll get to that too.

4.1.1 What is a service?

You might think a service is something you access across the network, like retrieving stock quotes or searching Google. In fact, the classical view of a service is something much

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

simpler, namely: “work done for another”. This definition could easily apply to a simple method call between two objects, because the callee is doing work for the caller. So how does a service differ from a method call? Well, a service implies a contract between the provider of the service and its consumers. Consumers typically aren't worried about the exact implementation behind a service (or even who provides it) as long as it follows the agreed contract, suggesting that services are to some extent substitutable. Using a service also involves a form of discovery or negotiation, implying each service has a set of identifying features (see Figure 4.1).

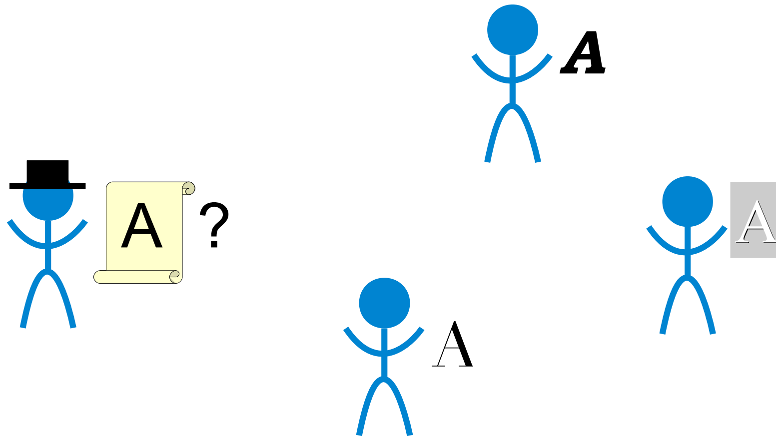


Figure 4.1 Services follow a contract and involve some form of discovery

If you think about it, Java interfaces provide part of a contract and Java class-linking is a type of service lookup because it “discovers” methods based on signatures and class hierarchy. Different method implementations can also be substituted by changing the JAR files on the class path. So a local method call could easily be seen as a service; although it would be even better if you could use a high-level abstraction to find services or if there was a more dynamic way to switch between implementations at execution time. Thankfully, OSGi helps with both by recording details of the service contract, such as interface names and metadata, and by providing a registry API to publish and discover services. You'll hear more about this later on in 4.2, for now let's continue to look at services in general.

You might be thinking a Java method call in the same process can't possibly be a service, because it doesn't involve a remote connection or a distributed system. In reality, as we shall see throughout this chapter, services do *not* have to be remote and there are many benefits to using a service-oriented approach in a purely local application.

Components versus services

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

When people discuss services they often talk about components in the same context, so it is useful for us to consider how services and components compare and overlap. In fact, service-oriented design and component-oriented design are extremely complimentary. The key semantic difference between these two approaches is:

- in a component-oriented approach the architect focuses on the provider's view, but
- in a service-oriented approach the architect focuses on the consumer's view.

Typically in a component-oriented approach, the architect is focused on ensuring that the component he or she provides is packaged in such a way that it makes his or her life easier. We know when it comes to packaging and deploying Java code there are often a range of different scenarios in which the code will be used. For example: live deployments may need extra security constraints applied; testing deployments may need extra assertion filters applied; and development deployments may need minimal dependencies for fast turn around. The component design approach tries to make it as easy as possible for the architect to chose what functionality they want to deploy without hard coding this into their application.

This contrasts with a service-oriented approach where the architect is focused on supplying a function, or set of functions, to consumers who typically have very little interest in how the internals of the individual component is constructed, but have very specific requirements for how they want the function to behave. For example: acid transactions; low latency; or encrypted data.

We will see in chapter 11 that component-oriented approaches can easily be built on top of the OSGi services model. With this in mind let's continue our introduction to services by considering what exactly are the benefits of services.

4.1.2 Why use services?

The main drive behind using services is to get others to do work on your behalf, rather than attempting to do everything yourself. This idea of delegation fits in very well with many object-oriented design techniques, such as Class-Responsibility-Collaboration (CRC) cards [ref1]. CRC cards are a role-playing device used by development teams to think about what classes they need, as well as which class will be responsible for which piece of work and how the various classes should collaborate to get work done. Techniques like CRC cards try to push work out to other components wherever possible, which leads to lean, well-defined, maintainable components. Think of this like a game of pass-the-parcel, where each developer is trying to pass parcels of work onto other developers – except in this game when the music stops you want the smallest pile of parcels!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

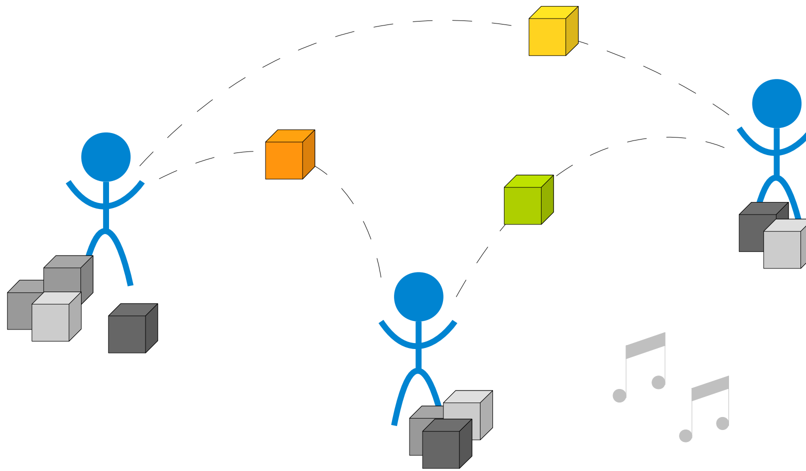


Figure 4.2 Using CRC to place responsibilities can be like playing pass-the-parcel

A service-oriented approach also promotes:

- less coupling between providers and consumers, so it's easier to re-use components,
- more emphasis on interfaces (the abstract) rather than superclasses (the concrete),
- clear descriptions of dependencies, so you know how it all fits together, and
- support for multiple competing implementations, so you can swap parts in and out.

In other words, it encourages a "plug-and-play" approach to software development, which means much more flexibility during development, testing, deployment, and maintenance. You don't mind where a service comes from, as long as it does what you want. Still not convinced? Let's see how each of these points help you build a better application.

Less coupling

One of the most important aspects of a service is the contract. Every service needs some form of contract, otherwise how could a consumer find it and use it? The contract should include everything a consumer needs to know about the service, but no more. Putting too much detail in a contract tightens the coupling between the provider and consumer, and limits the possibility of swapping in other implementations later on. To put it in clothing terms, you want it nice and stretchy to give your application room to breathe!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

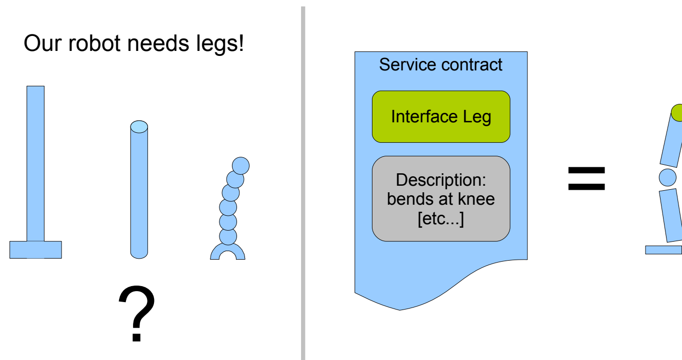


Figure 4.3 Why you need contracts

A good service contract clearly and cleanly defines the boundary between major components and helps with development and maintenance. Once the contract is defined developers can work on implementing service providers and consumers in parallel to reduce development time and can use scripted or "mock" services to perform early testing of key requirements. So contracts are good news for everyone, but how do we define one in Java?

MORE EMPHASIS ON INTERFACES

Java interfaces can form part of a service contract. They list the various methods that make up a service along with expected parameters and return types. Once defined, developers can start programming against the agreed upon set of interfaces without having to wait for others to finish their implementations (Figure 4.4). Interfaces also have several advantages over concrete classes. A Java class can implement several interfaces, whereas it could only ever extend one concrete class. This is essential if you want flexibility over how you implement related services. Interfaces also provide a higher level of encapsulation, because they cannot define any mutable fields.

You could decide to stop at this point and assemble your final application by creating the various components with "new" and wire their dependencies manually. Or you could use a dependency injection framework to do the construction and wiring for you. If you did then you would have a pluggable application and all the benefits that entails, but you would also miss out on two other benefits of a service-oriented approach: rich metadata and the ability to switch between implementations at execution time.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

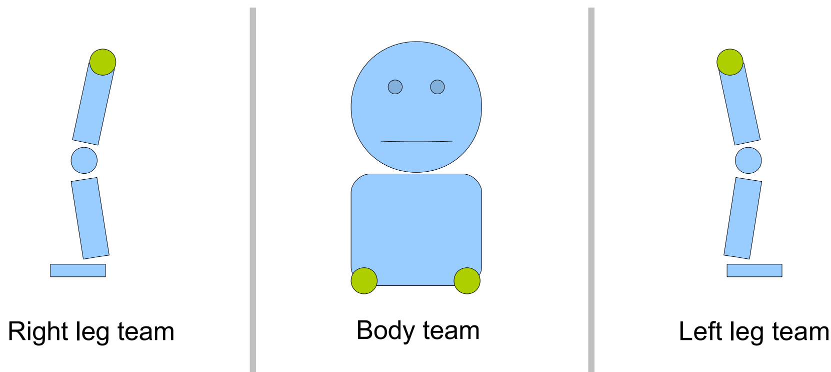


Figure 4.4 Programming to interfaces means teams can work in parallel

Clear descriptions of dependencies

Interfaces alone can't easily capture certain characteristics of a service; like the quality of a particular implementation, or configuration settings like supported locales. Such details are often best recorded as metadata alongside the service interface, and to do this you need some kind of framework. Semantics, which describe "what" a service does, are also hard to capture. Simple semantics like pre- and post-conditions can be recorded using metadata or might even be enforced by the service framework. Other semantics can only be properly described in documentation, but even here metadata can help provide a link to the relevant information. Think about your current application, what characteristics might you want to record outside of classes and interfaces? To get you started, Table 1 describes some characteristics from real-world services that could be recorded as metadata:

Table 1 Example characteristics of real-world services

Characteristic	Why might you be interested?
Supported locales	A price checking service may only be available for certain currencies
Transaction cost	You might want to use the cheapest service, even if it takes longer
Throughput	You may want to use the fastest service regardless of cost
Security	You may only want to use services that are digitally signed by certain providers
Persistence characteristics	You may only want to use a service that guarantees to store your data in such a way that it will not be lost should the JVM be restarted

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

As you can see, metadata can capture fine-grained information about your application in a structured way. This is very helpful to developers when assembling, supporting, and maintaining an application. Recording metadata alongside a service interface also means you can be more exact about what you actually need. The service framework can use this metadata to filter out services you don't want, without having to load and access the service itself. But why might you want to do this? Why not just call a method on the service to ask if it does what you need?

Support for multiple competing implementations

A single Java interface can have many implementations; one might be fast, but use a lot of memory, another could be slow, but conserve memory. How do you know which one to use when they both implement the same interface? You could add a query method to the interface that tells you more about the underlying implementation, but this leads to bloat and reduces maintainability. What happens when you add another implementation that can't be characterized using the existing method? Using a query method also means you have to find and call each service implementation before you even know whether you want to use it or not, which isn't very efficient. Especially when you might have hundreds of potential implementations that could be loaded at execution time.

Because service frameworks help you record metadata alongside services, they can also help you query and filter on this metadata when discovering services. This is different from classic dependency injection frameworks, which look up implementations solely based on the interfaces used at a given dependency point. Figure 4.5 shows how services can help you get exactly what you want.

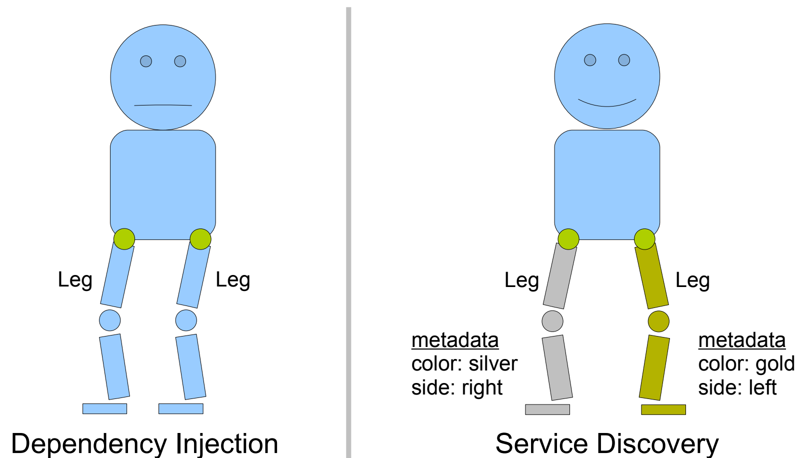


Figure 4.5 Dependency injection vs. service discovery

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Hopefully by now you agree that services are a good thing – but as the saying goes, you can have too much of a good thing! How can you know when you should use a service or when it would be better to use another approach such as a static factory method or simple dependency injection?

4.1.3 When to use services?

The best way to decide when to use a service is to consider the benefits: less coupling, programming to interfaces, additional metadata, and multiple implementations. If you have a situation where any of these make sense or your current design provides similar benefits, then you should use a service. The most obvious place to use a service is between major components, especially if you want to replace or upgrade those components over time without having to rewrite other parts of the application. Similarly, anywhere you lookup and choose between implementations is another candidate for a service, because it means you can replace your custom logic with a standard, recognized approach.

Services can also be used as a substitute for the classic listener pattern [ref2]. The listener pattern is when one object offers to send events to other objects, known as listeners. The event source provides methods to subscribe and unsubscribe listeners and is responsible for maintaining the list of listeners. Each listener implements a known interface to receive events and is responsible for subscribing and unsubscribing to the event source (Figure 4.6).

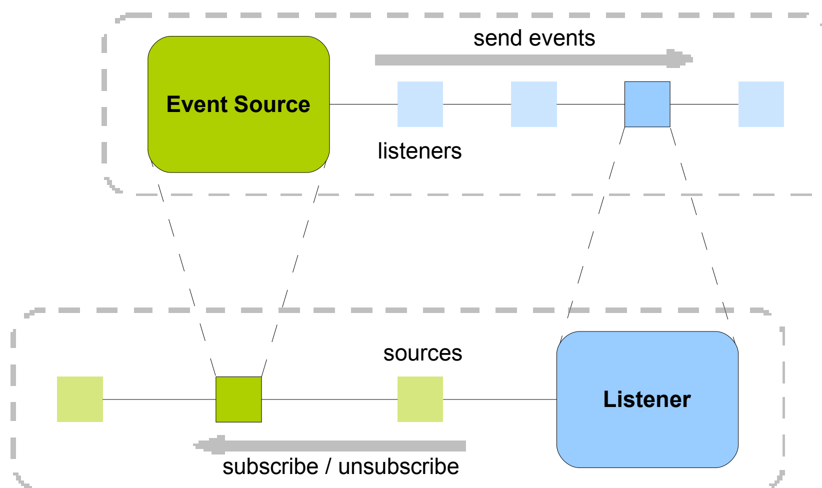


Figure 4.6 Listener pattern

Implementing the listener pattern involves writing a lot of code to manage and register listeners, but how can services help? A service can be seen as a more general form of

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

listener, because it can receive all kinds of requests, not just events. So why not save time and get the service framework to manage listeners for you by registering them as services? To find the current list of listeners, the sender just queries the service framework for matching services (Figure 4.2). Service metadata can be used to further define and filter the interesting events for a listener. In OSGi, this is known as the whiteboard pattern and we'll use this pattern later on when we update the "paint example" to use services in 4.4.

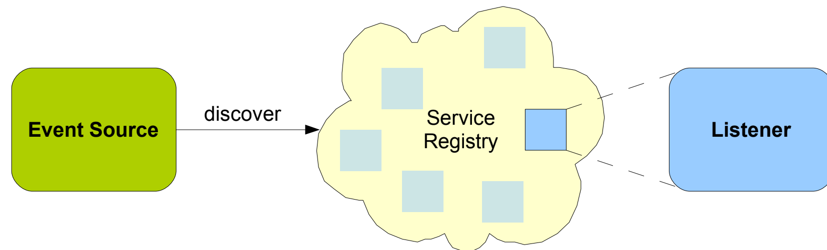


Figure 4.7 Whiteboard pattern

When not to use services?

Another way to decide if you should use a service is to consider when you wouldn't want to use them. Depending on the service framework, there may be overhead involved when calling a service, so you probably don't want to use them in performance critical code. That said, the overhead when calling a service in OSGi can be almost equivalent to making a direct method call when you use the approaches shown in chapter 11. You should also consider the work needed to define and maintain the service contract. There is no point in using a service between two tightly coupled pieces of code that are always developed and updated in tandem (unless of course you need to keep choosing between multiple implementations).

What if you're still not sure whether to use a service or not? Thankfully there is an approach which makes development easier and helps you migrate to services later on: programming to interfaces. If you use interfaces, then you're already more than halfway to using services, especially if you also take advantage of dependency injection. Of course interfaces can be taken to extremes; there's no point in creating an interface for a class if there's only ever going to be one implementation. But for outward-facing interaction between components, it definitely makes sense to use interfaces wherever possible.

So what have we learned? We saw how interfaces reduce coupling and promote faster development, regardless of whether you actually end up using services. We also saw how services help capture and describe dependencies and how they can be used to switch between different implementations. More importantly, we learned how a service-oriented approach makes developers think more about where work should be done, rather than just lumping code all in one place. And finally, we went through a whole section about services without once mentioning remote or distributed systems.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

So is OSGi just another service model? Should we end the chapter here with an overview of the API and move on to other topics? No, because there is one aspect that is unique to the OSGi service model – services are completely dynamic!

4.2 OSGi services in action!

What do we mean by dynamic? After your bundle has discovered and started using a service in OSGi, it can disappear at any time. Perhaps the bundle providing it has been stopped or even uninstalled, perhaps a piece of hardware has failed; whatever the reason, you should be prepared to cope with services coming and going over time. This is different from many other service frameworks, where once you bind to a service it is fixed and never changes – although it might throw a runtime exception to indicate a problem. OSGi doesn't try to hide this dynamism, if a bundle wants to stop providing a service then there is little point in trying to hold it back or pretend the service is still there. This is similar to many of the failure models used in distributed computing. Hardware problems in particular should be acknowledged and dealt with promptly rather than ignored. Thankfully, OSGi provides a number of techniques and utility classes to build robust yet responsive applications on top of such fluidity; we'll look more closely at these in chapter 11. Before we can discuss the best way to handle dynamic services, we need to understand how OSGi services work at the basic level.

The OSGi framework has a centralized service registry that follows a "publish-find-bind" model (Figure 4.7). Or to put this in the perspective of service providers and consumers:

- A providing bundle can publish POJOs as services
- A consuming bundle can find and then bind services

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

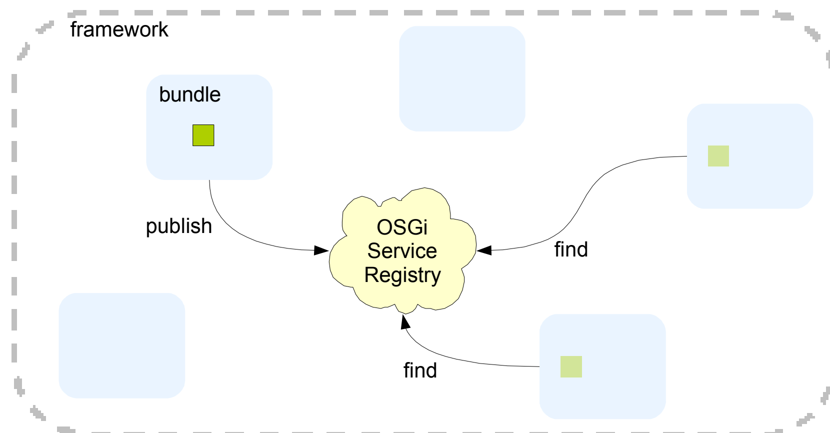


Figure 4.8 OSGi service registry

The OSGi service registry is accessed through the `BundleContext` interface, which we saw back in section 3.2.4; back then we looked at its lifecycle-related methods, now we'll look into its service-related methods as shown in Listing 4.1.

Listing 4.1 `BundleContext` methods related to services

```
public interface BundleContext {
  ...

  void addServiceListener(ServiceListener listener, String filter)
    throws InvalidSyntaxException;

  void addServiceListener(ServiceListener listener);

  void removeServiceListener(ServiceListener listener);

  ServiceRegistration registerService(
    String[] clazzes, Object service, Dictionary properties);

  ServiceRegistration registerService(
    String clazz, Object service, Dictionary properties);

  ServiceReference[] getServiceReferences(String clazz, String filter)
    throws InvalidSyntaxException;

  ServiceReference[] getAllServiceReferences(String clazz, String filter)
    throws InvalidSyntaxException;

  ServiceReference getServiceReference(String clazz);

  Object getService(ServiceReference reference);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
  boolean ungetService(ServiceReference reference);

  ...
}
```

As long as your bundle has a valid context (i.e., when it is active), it can use services. Let's see how easy it is to use a bundle's `BundleContext` to publish a service.

4.2.1 Publishing a service

Before we can publish a service, we need to describe it so others can find it. In other words, we need to take details from the agreed contract and record them in the registry. So what details does OSGi need from the contract?

Defining a service

It's actually very simple, in order to publish a service in OSGi you need to provide an array of interface names along with an optional dictionary of metadata. Here's what you might use for a service that can provide both stock listings and stock charts:

```
String[] interfaces = new String[] {
    StockListing.class.getName(), StockChart.class.getName()};            #A

Dictionary metadata = new Properties();                                    #B
metadata.setProperty("name", "LSE");
metadata.setProperty("currency", Currency.getInstance("GBP"));             #C
metadata.setProperty("country", "GB");
```

#A Class.getName() helps when refactoring
#B metadata must be in Dictionary type
#C metadata can contain any Java type

Once everything is ready, you can publish your service by using your bundle context:

```
ServiceRegistration registration =
    bundleContext.registerService(interfaces, new LSE(), metadata);
```

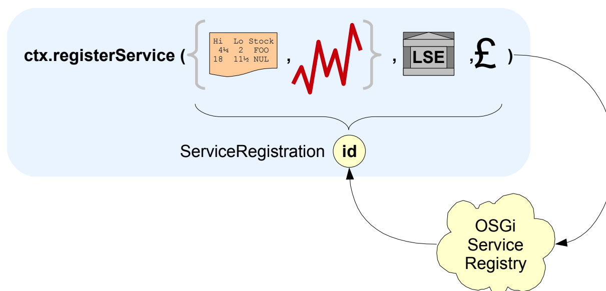


Figure 4.9 Publishing a service that provides both stock listings and stock charts

SERVICE REGISTRATIONS ARE PRIVATE

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The service registration should not be shared with other bundles, because it is tied to the lifecycle of the publishing bundle.

The "LSE" implementation is a Plain Old Java Object (POJO), it doesn't need to extend or implement any specific OSGi types or use any annotations. It just has to match the provided service details. There is no leakage of OSGi types into service implementations. In fact, you don't even have to use interfaces if you don't want to, OSGi will accept services registered under concrete class names.

Updating service metadata

Once you've published a service, you can change its metadata at any time by using its service registration:

```
registration.setProperties(newMetadata);
```

This makes it easy for your service to adapt to circumstances and inform consumers about any such changes by updating its metadata. The only piece of metadata you cannot change is the "service.id" property, which is maintained by the framework. Other properties which have special meaning to the OSGi framework are shown in Table 2:

Table 2 Standard OSGi Service Properties

Key	Type	Description
objectClass	String[]	The class names the service is registered under, defaults to the set of names passed in when registering the service.
service.id	Long	Unique registration sequence number, assigned by the framework when registering the service – cannot be chosen or changed by the developer.
service.pid	String	Persistent (unique) service identifier, chosen by developer.
service.ranking	Integer	Ranking used when discovering services, defaults to 0 – services are sorted by their ranking (highest first) and then by their id (lowest first)
service.description	String	Description of the service, chosen by developer.
service.vendor	String	Name of the vendor providing the service, chosen by developer.

Removing a service

Removing a published service can also be done at any time by the publishing bundle:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
registration.unregister();
```

What happens if your bundle stops before you've removed all your published services? The framework keeps track of what you have registered and any services that have not yet been removed when a bundle stops are automatically removed by the framework. So you don't have to explicitly unregister a service when your bundle is stopped, although it is prudent to unregister before cleaning up required resources. Otherwise, someone could attempt to use the service while you're trying to clean it up.

We've successfully published our service in only a few lines of code and without any use of OSGi types in our service implementation, now let's see if it's just as easy to discover and use the service.

4.2.2 Finding and binding services

As with publishing, you need to take details from the service contract to discover the right services in the registry. The simplest query takes a single interface name, which is the main interface you expect to use as a consumer of the service:

```
ServiceReference reference =
    bundleContext.getServiceReference(StockListing.class.getName());
```

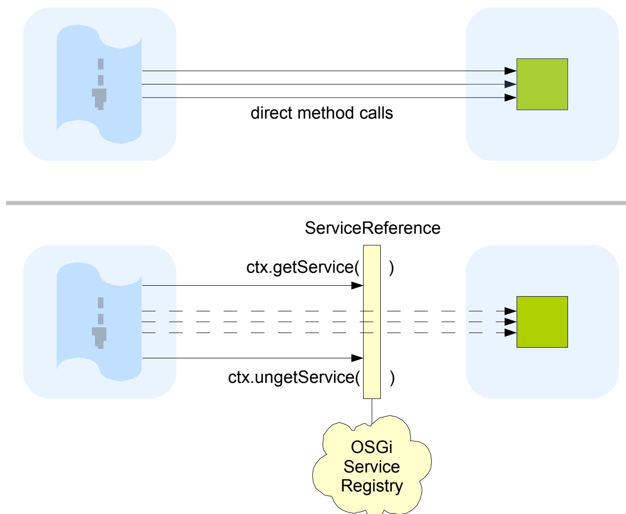


Figure 4.10 Using an OSGi service

This time the registry returns a service reference, which is an indirect reference to the discovered service. This service reference can safely be shared with other bundles, because it

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

is not tied to the lifecycle of the discovering bundle. But why does the registry return an indirect reference and not the actual service implementation?

To make services fully dynamic, the registry must decouple the use of a service from its implementation. By using an indirect reference it can track usage of the service, support laziness, and tell consumers when the service is removed.

Revisiting our magic method

Recall in chapter 3 when we implemented the "refresh" command for our shell, we had to use the magic `getPackageAdminService()` method to acquire the Package Admin Service? Now we have enough knowledge to see what was really happening behind the scenes:

```
private PackageAdmin getPackageAdminService() {
  return (PackageAdmin) m_context.getService(              #1
    m_context.getServiceReference(                          #2
      PackageAdmin.class.getName()));                       #3
}
```

As you see, the method is actually quite simple. At (#2) we use the `BundleContext` to find a service implementing the Package Admin Service interface at (#3). This returns a service reference, which we use to get the service implementation at (#1). No more magic!

Choosing the best service

If multiple services match the given query, the framework will choose what it considers to be the "best" services. It determines the best service using the ranking property mentioned in Table 2, where a larger numeric value denotes a higher ranked service. If multiple services have the same ranking, then the framework will then choose the service with the lowest service identifier, also covered in Table 2. Since the service identifier is an increasing number assigned by the framework, lower identifiers are associated with older services. So if multiple services have equal ranks, the framework effectively chooses the oldest service, which guarantees some stability and provides an affinity to existing services.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

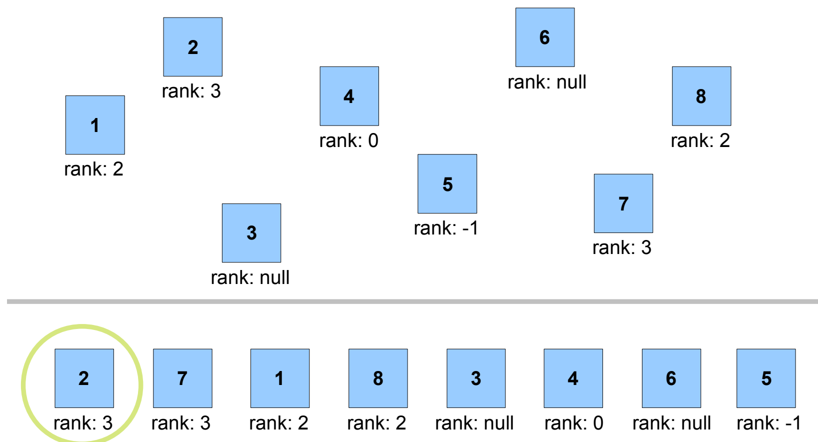


Figure 4.11 OSGi service ordering (by highest service.ranking then lowest service.id)

You've seen how to find services based on the interfaces they provide, but what if you want to discover services with certain properties? For example, in Figure 4.12 if we just ask for any stock listing service we'd get back the first one (NYSE) – but what if we want a UK based listing? The bundle context provides us with another query method that accepts a standard LDAP filter string, described in RFC 1960 [ref3], and returns all services matching the filter.

A quick guide to using LDAP queries

Attribute matching:

```
(name=John Smith)
(age>=20)
(age<=65)
```

Fuzzy matching:

```
(name~=johnsmith)
```

Wild-card matching:

```
(name=Jo*n*Smith*)
```

Does attribute exist:

```
(name=*)
```

Match ALL the contained clauses:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
(&(name=John Smith)(occupation=doctor))
```

Match ONE of the contained clauses:

```
(|(name~=John Smith)(name~=Smith John))
```

Negate the contained clause:

```
(!(name=John Smith))
```

Here's how you might find all stock listing services using the GBP currency:

```
ServiceReference[] references =
    bundleContext.getServiceReferences(StockListing.class.getName(),
        "(currency=GBP)");
```

We would now get back the LSE service instance with service.id 3.

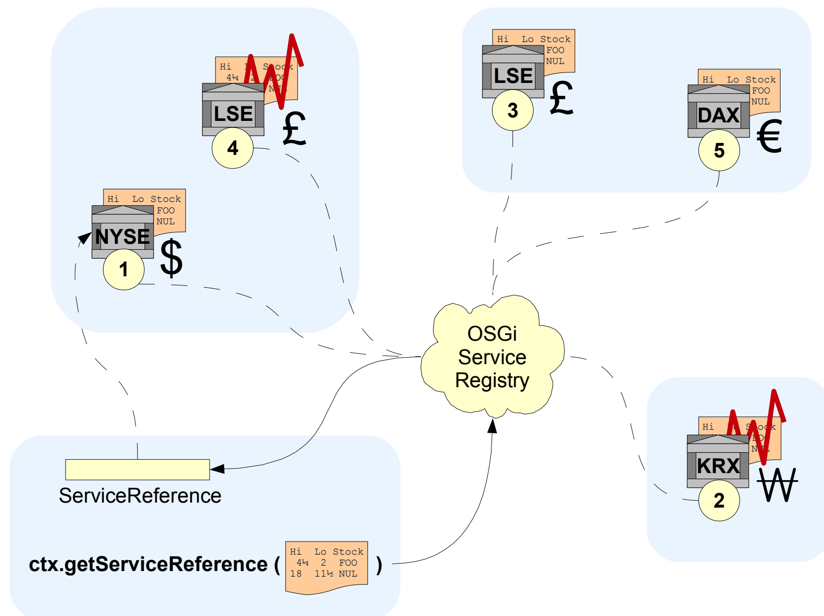


Figure 4.12 Discovering an OSGi service

You can also use the "objectClass" property, mentioned in Table 2, to query for services providing additional interfaces. Here we narrow the search to those stock listing services that use a currency of GBP and also provide a chart service:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
ServiceReference[] references =
    bundleContext.getServiceReferences(StockListing.class.getName(),
        "(&(currency=GBP)(objectClass=org.example.StockChart))");
```

Which means we would now get the LSE service instance with service.id 4. As we've just seen, we can look up all sorts of service references based on our needs, but how do we use them? We need to dereference each service reference to get the actual service object.

Using a service

Before you can use a service you must bind to the actual implementation from the registry:

```
StockListing listing =
    (StockListing) bundleContext.getService(reference);
```

The implementation returned will typically be exactly the same POJO instance previously registered with the registry, although the OSGi specification does not actually prohibit use of proxies or wrappers. Each time you call `getService()` the registry will increment a usage count, so it can keep track of who is using a particular service. To be a good OSGi citizen, we should tell the registry when we have finished with a service:

```
bundleContext.ungetService(reference);
listing = null;
```

SERVICES ARE NOT PROXIES

In general, in OSGi when you are making method calls on a service you are holding a reference to the actual Java object supplied by the providing bundle. For this reason, you should also remember to null variables referring to the service instance when done using it, so it can be safely garbage collected. The actual service implementation should generally never be stored in a long-lived variable such as a field, instead you should try to access it temporarily via the service reference and expect that the service may go away at any time.

We have now seen how to publish simple Java POJOs as OSGi services, how they can be discovered, and how the registry tracks their use. But if you remember one thing from this section it should be that services can disappear at any time. If you want to write a robust OSGi-based application you should not rely on services always being around or even appearing in a particular order when starting your application. Of course, we don't want to scare you with all of this talk of dynamism. It is important to realize dynamism isn't created or generated by OSGi, it just enables it. Services are never arbitrarily removed; either a bundle has decided to remove it or an agent has stopped a bundle. You have control over how much dynamics you need to deal with, but it is always good to code defensively in case things change in the future or your bundles are used in different scenarios.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

So what is the best way to cope with potential dynamism? How can you get the most from dynamic services without continual checking and re-checking? Well the next section discusses potential pitfalls and recommended approaches when programming with dynamic services.

4.3 Dealing with dynamics

In the last section we covered the basics of OSGi services and saw how easy it is to publish and discover services. In this section we shall look more closely at the dynamics of services and techniques to help you write robust OSGi applications. To demonstrate this, we are going to use the OSGi Log Service.

The Log Service is a standard OSGi service, one of the so-called "compendium" or non-core services. Compendium services will be covered more in 4.6.2. For now, all we need to know is that the Log Service provides a simple logging facade, with various flavors of methods accepting a logging level and a message.

Listing 4.2 The OSGi Log Service

```
package org.osgi.service.log;

import org.osgi.framework.ServiceReference;

public interface LogService {

  public static final int LOG_ERROR   = 1;
  public static final int LOG_WARNING = 2;
  public static final int LOG_INFO    = 3;
  public static final int LOG_DEBUG   = 4;

  public void log(int level, String message);
  public void log(int level, String message,
      Throwable exception);

  public void log(ServiceReference sr, int level, String message);
  public void log(ServiceReference sr, int level, String message,
      Throwable exception);
}
```

With OSGi, we could use any number of possible Log Service implementations in our example, such as those written by OSGi framework vendors or others written by third-party bundle vendors. To keep things simple and to help us trace what's happening inside the framework, we will use our own dummy Log Service that implements only one method and outputs a variety of debug information about the bundles using it.

We learned the basics of discovering services in section 4.2.2, in the following section we will take that knowledge and use it to lookup and call the Log Service, pointing out and solving potential problems as we go along.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

EXAMPLES AHEAD, PLEASE BEWARE

The examples in the next section are intended purely to demonstrate the proper use of dynamic OSGi services. In order to keep these explanatory code snippets focused and to the point they will occasionally avoid using proper programming techniques such as encapsulation. You should be able to join the dots between the patterns we show you in these examples and real-world OO design.

4.3.1 Avoiding common pitfalls

When people start using OSGi, they often write code that looks similar to this:

Listing 4.3 Broken lookup example – service instance stored in a field

```
public class Activator implements BundleActivator {

  volatile LogService m_logService;                                    #A

  public void start(BundleContext context) {
    ServiceReference logServiceRef =
        context.getServiceReference(LogService.class.getName());       #B

    m_logService = (LogService) context.getService(logServiceRef);     #C

    startTestThread();                                                 #D
  }

  public void stop(BundleContext context) {
    m_logService = null;                                               #E
    stopTestThread();
  }
}
```

#A volatile shared between threads
#B find single best Log Service
#C store instance in field (bad!)
#D start Log Service test thread
#E clear field when we're done

Because we store the Log Service instance in a field, our test code can be very simple:

```
while (m_logService != null) {
  m_logService.log(LogService.LOG_INFO, "ping");
  pauseTestThread();
}
```

But there is a major problem with our bundle activator! The Log Service implementation is stored directly in a field, which means the consumer won't know when the service is retracted by its providing bundle. It only finds out when the implementation starts throwing exceptions after removal when the implementation becomes unstable. This hard reference to the implementation also keeps it from being garbage collected while our bundle is active,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

even if the providing bundle is uninstalled. To fix this, let's replace the Log Service field with the indirect service reference.

Listing 4.4 Broken lookup example – service is only discovered on startup

```
public class Activator implements BundleActivator {

  volatile ServiceReference m_logServiceRef;
  BundleContext m_context;

  public void start(BundleContext context) {
    m_logServiceRef =
        context.getServiceReference(LogService.class.getName());          #A

    m_context = context;                                                    #B

    startTestThread();
  }

  public void stop(BundleContext context) {
    m_logServiceRef = null;
    stopTestThread();
  }
}
```

#A store indirect service reference instead
#B must remember context for later on

We will also need to change our test method to always dereference the service:

Listing 4.5 Broken lookup example – testing the discovered Log Service

```
while (m_logServiceRef != null) {
  LogService logService =
      (LogService) m_context.getService(m_logServiceRef);                   #A

  if (logService != null) {                                                 #B
    logService.log(LogService.LOG_INFO, "ping");
  } else {
    alternativeLog("LogService has gone");
  }

  pauseTestThread();
}
```

#A need the saved bundle context
#B if null then service was removed

This is slightly better, but there is still a problem with our bundle activator. We only discover the Log Service once in the start method, so if there is no Log Service when our bundle starts then the reference will always be `null`. Similarly, if there is a Log Service at startup, but it subsequently disappears, then the reference will always return `null` from that

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

point onwards. Perhaps you want this one-off check, so you can revert to another (non-OSGi) logging approach based on what's available at startup. But this isn't very flexible, it would be much better if we could react to changes in the Log Service and always use the active one.

A simple way of reacting to potential service changes is to always lookup the service just before you want to use it, like so:

Listing 4.6 Broken lookup example – potential race condition

```
while (m_context != null) {
  ServiceReference logServiceRef =
      m_context.getServiceReference(LogService.class.getName());           #A

  if (logServiceRef != null) {                                              #B
    ((LogService) m_context.getService(logServiceRef)).log(
        LogService.LOG_INFO, "ping");                                       #C
  } else {
    alternativeLog("LogService has gone");
  }

  pauseTestThread();
}
```

#A lookup best Log Service each time
#B if null then service was removed
#C safe to dereference... or is it?

With this change our bundle activator becomes very trivial, just recording the context:

```
public class Activator implements BundleActivator {

  volatile BundleContext m_context;

  public void start(BundleContext context) {
    m_context = context;
    startTestThread();
  }

  public void stop(BundleContext context) {
    m_context = null;
    stopTestThread();
  }
}
```

Unfortunately, we're still not done, because there is a problem in our test method – can you see what it is? Here's a clue: remember that services can disappear at any time and with a multi-threaded application this can even happen between single statements.

The problem is that in between the calls to `getServiceReference()` and `getService()`, the Log Service could disappear. The current code assumes once you have a reference you can safely dereference it immediately afterwards. This is a common mistake

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

made when starting with OSGi and an example of what's known as a “race condition” in computing.

Let's make the lookup more robust by adding a few more checks and a try-catch block:

Listing 4.7 Correct lookup example

```
while (m_context != null) {
  ServiceReference logServiceRef =
      m_context.getServiceReference(LogService.class.getName());          #A

  if (logServiceRef != null) {                                            #B
    try {
      LogService logService =
          (LogService) m_context.getService(logServiceRef);

      if (logService != null) {                                           #C
        logService.log(LogService.LOG_INFO, "ping");
      } else {
        alternativeLog("LogService has gone");
      }

    } catch (RuntimeException re) {
      alternativeLog("error in LogService " + re);                        #D
    } finally {
      m_context.ungetService(logServiceRef);                              #E
    }
  } else {
    alternativeLog("LogService has gone");
  }

  pauseTestThread();
}
```

#A lookup best Log Service each time
#B if null then no service available
#C if null then service was removed
#D report any problems using service
#E unget service when not using it

Our test method is now robust, but not perfect. We react to changes in the Log Service, and fall back to other logging methods when there are problems finding or using a service, but we can still miss Log Service implementations. For example, imagine a Log Service is available when we first call `getServiceReference()`, but it is removed and a different Log Service appears before we can use the original service reference. Our `getService()` call will return `null` and we'll end up not using any Log Service, even though a valid replacement was available. This particular race condition cannot be solved by adding checks or loops because it is an inherent problem with the two stage “find-then-get” discovery process. Instead, we must use another facility provided by the service layer to avoid this problem: service listeners.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

4.3.2 Listening for services

The OSGi framework supports a simple, but flexible listener API for service events. We briefly discussed the listener pattern back in 4.1.3, where one object (in this case the framework) offers to send events to other objects, known as listeners. For services there are currently three different types of event, shown in Figure 4.13:

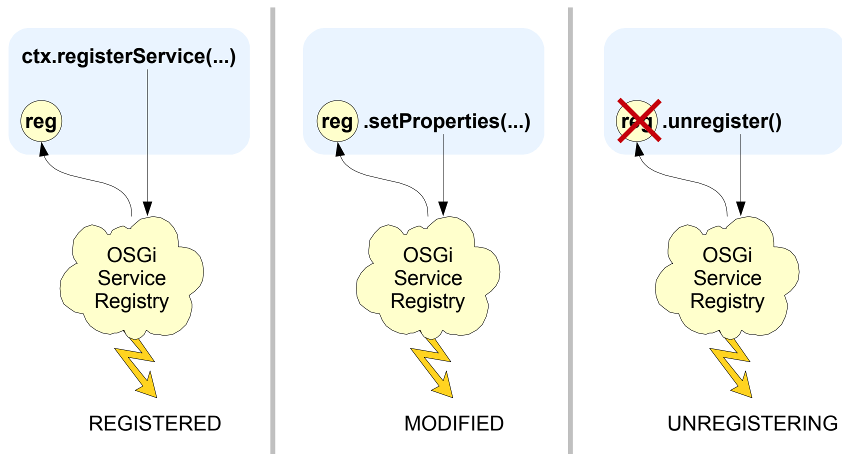


Figure 4.13 OSGi service events

- REGISTERED - a service has been registered and can now be used.
- MODIFIED - some service metadata has been modified.
- UNREGISTERING - a service is in the process of being unregistered.

Every service listener must implement this interface in order to receive service events:

```
public interface ServiceListener extends EventListener {
  public void serviceChanged(ServiceEvent event);
}
```

How might we use such an interface in our current example? We could use it to cache service instances on REGISTERED events and avoid the cost of repeatedly looking up the Log Service, as we were doing in 4.3.1. A simple caching implementation might go something like this:

Listing 4.8 Broken listener example – caching the latest service instance

```
class LogListener implements ServiceListener {
  public void serviceChanged(ServiceEvent event) {
    switch (event.getType()) {

      case ServiceEvent.REGISTERED:
        m_logService = (LogService)
            m_context.getService(event.getServiceReference());           #A
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
          break;

        case ServiceEvent.MODIFIED:                                        #B
          break;

        case ServiceEvent.UNREGISTERING:
          m_logService = null;                                             #C
          break;

        default:
          break;
      }
    }
  }
```

#A valid as events are delivered synchronously
#B nothing to do – only metadata has changed
#C stop using service (see a problem here?)

It is safe to call the `getService()` method during the `REGISTERED` event, because the framework delivers service events synchronously using the same thread. This means we know the service won't disappear, at least from the perspective of the framework, until the listener method returns. Of course the service itself could still throw a runtime exception at any time, but using `getService()` with a `REGISTERED` event will always return a valid service instance. For the same reason, you should make sure the listener method is relatively short and will not block or deadlock, otherwise you would block other service events from being processed.

Registering a service listener

We have our service listener, but how do we tell the framework about it? The answer is as usual via the bundle context, which defines methods to add and remove service listeners. We must also choose an LDAP filter to restrict events to services implementing Log Service; otherwise, we could end up receiving events for hundreds of different services. The final code looks something like this:

Listing 4.9 Broken listener example – existing services are not seen

```
public class Activator implements BundleActivator {

  volatile BundleContext m_context;
  volatile LogService m_logService;

  public void start(BundleContext context) throws Exception {
    m_context = context;

    String filter = "(" + Constants.OBJECTCLASS + "=" +                #A
        LogService.class.getName() + ")";

    context.addServiceListener(new LogListener(), filter);             #B
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    startTestThread();
  }

  public void stop(BundleContext context) {                              #C
    m_logService = null;
    m_context = null;
    stopTestThread();
  }
}
```

#A LDAP filter matches LogService instances
#B add listener for future Log Service events
#C framework automatically removes listeners

Notice that we don't explicitly remove the service listener when we stop the bundle. This is because the framework keeps track of what listeners we have added and automatically clears up any remaining listeners when the bundle stops. We saw something similar in 4.2.1 when the framework removed any leftover service registrations.

Our test method is now very simple, because we're caching the service instance:

```
while (m_context != null) {
  if (m_logService != null) {
    m_logService.log(LogService.LOG_INFO, "ping");
  } else {
    alternativeLog("LogService has gone");
  }
  pauseTestThread();
}
```

This looks much better doesn't it? We don't have to do so much checking, nor polling of the service registry. Instead, we wait for the registry to tell us whenever a Log Service appears or disappears. Unfortunately there are a number of problems in the above code example! First, there are some minor issues with our test method; we don't catch runtime exceptions when using the service and because of the caching we don't "unget" the service when we're not using it. The cached Log Service could also change between the non-null test and when we actually use it.

More importantly there is a significant error in the listener code, because it doesn't check that the `UNREGISTERING` service is the same as the Log Service currently being used. Imagine two Log Services (A and B) are available at the same time, where the test method was using Log Service A. If Log Service B was unregistered, the listener would clear the cached instance even though Log Service A is still available. Similarly as new Log Services are registered, the listener will always choose the newest service regardless of whether it has a better service ranking or not. To make sure we use the highest ranked service and to be able to switch to alternative implementations whenever a service is removed, we must keep track of the current set of active service references – not just a single instance.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

There is another subtle error in the bundle activator in Listing 4.9, which you may not have noticed at first. In fact, this error might never show up in practice, depending on how you start your application. Think back to how listeners work: the event source will send events to the listener as they occur. What about events that happened in the past? What about already published services? In this case, the service listener will not receive events that happened in the dim and distant past and will remain oblivious to existing Log Service implementations.

Fixing the service listener

So we have two problems to fix: we must keep track of the active set of Log Services and take account of already registered Log Services. The first problem requires the use of a sorted set and relies on the natural ordering of service references, as defined in the specification of the `compareTo()` method. We'll also add a helper method to decide which Log Service to pass to the client, based on the cached set of active service references:

Listing 4.10 Correct listener example – keeping track of active Log Services

```
class LogListener implements ServiceListener {

  SortedSet<ServiceReference> m_logServiceRefs =
      new TreeSet<ServiceReference>();                                  #A

  public synchronized void serviceChanged(ServiceEvent event) {         #B
    switch (event.getType()) {
      case ServiceEvent.REGISTERED:
        m_logServiceRefs.add(event.getServiceReference());
        break;
      case ServiceEvent.MODIFIED:
        break;
      case ServiceEvent.UNREGISTERING:
        m_logServiceRefs.remove(event.getServiceReference());
        break;
      default:
        break;
    }
  }

  public synchronized LogService getLogService() {                     #C
    if (m_logServiceRefs.size() > 0) {
      return (LogService) m_context.getService(
          m_logServiceRefs.last());                                     #D
    }
    return null;
  }
}
```

#A service references ordered by ranking
#B must lock listener before changing state
#C must lock listener before querying state
#D last service reference has highest ranking

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The second problem can be fixed in the bundle activator by issuing pseudo-registration events for each existing service, to make it look like the service has only just appeared:

Listing 4.11 Correct listener example – sending pseudo registration events

```
public class Activator implements BundleActivator {

  volatile BundleContext m_context;
  volatile LogListener m_logListener;

  public void start(BundleContext context) throws Exception {
    m_context = context;

    m_logListener = new LogListener();                                       #A

    synchronized (m_logListener) {                                           #B

      String filter = "(" + Constants.OBJECTCLASS + "=" +
          LogService.class.getName() + ")";

      context.addServiceListener(m_logListener, filter);

      ServiceReference[] refs =
          context.getServiceReferences(null, filter);                        #C

      if (refs != null) {
        for (ServiceReference r : refs) {
          m_logListener.serviceChanged(
              new ServiceEvent(ServiceEvent.REGISTERED, r));                 #D
        }
      }
    }

    startTestThread();
  }

  public void stop(BundleContext context) {
    m_context.removeServiceListener(m_logListener);                          #E

    m_logListener = null;
    m_context = null;

    stopTestThread();
  }
}
```

#A store listener for test thread
#B lock listener before adding it
#C check for existing Log Services
#D send pseudo events for existing services
#E example of explicitly removing a listener

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

We deliberately lock the listener before passing it to the framework, so that our pseudo-registration events will be processed first. Otherwise, it would be possible to receive an UNREGISTERING event for a service before its pseudo-registration. Only when the listener has been added do we check for existing services, to make sure we don't miss any intervening registrations. We could potentially end up with duplicate registrations by doing the checks in this order, but that is better than missing services.

The test method now only needs to call the helper method to get the best Log Service:

Listing 4.12 Correct listener example – using the listener to get the best Log Service

```
while (m_context != null) {
  LogService logService = m_logListener.getLogService();

  if (logService != null) {
    try {
      logService.log(LogService.LOG_INFO, "ping");
    } catch (RuntimeException re) {
      alternativeLog("error in LogService " + re);
    }
  } else {
    alternativeLog("LogService has gone");
  }

  pauseTestThread();
}
```

You might have noticed the finished listener example still doesn't "unget" the service after using it, this is left as an exercise for the reader. Here's a hint to get you started; think about moving responsibility for logging into the listener. This would also help you reduce the time between binding the service and actually using it.

Service listeners reduce the need to continually poll the service registry. They let you react to changes in services as soon as they occur and get around the inherent race condition of the "find-then-get" approach. The downside of listeners is the amount of code you need to write. Imagine having to do this for every service you want to use and having to repeatedly test for synchronization issues. Why doesn't OSGi provide a utility class to do all of this for you, a class that has been battle-hardened and tested in many applications, which you can configure and customize as you require. Well it does, and its name is the ServiceTracker.

4.3.3 Tracking services

The OSGi ServiceTracker class provides a safe way for developers to get the benefits of service listeners without the pain. To show how easy it can be, let's take the bundle activator from our last example and adapt it to use the service tracker:

Listing 4.13 Standard tracker example

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
public class Activator implements BundleActivator {

  volatile BundleContext m_context;
  volatile ServiceTracker m_logTracker;

  public void start(BundleContext context) {
    m_context = context;

    m_logTracker = new ServiceTracker(context,
        LogService.class.getName(), null);                          #A

    m_logTracker.open();                                            #B

    startTestThread();
  }

  public void stop(BundleContext context) {

    m_logTracker.close();                                           #C

    m_logTracker = null;
    m_context = null;

    stopTestThread();
  }
}
```

#A create a ServiceTracker for Log Services
#B must open tracker before using it
#C close tracker to ensure proper cleanup

In this example we use the basic `ServiceTracker` constructor that takes a bundle context, the service type you want to track, and a "customizer" object. We'll look at customizers in a moment, for now we don't need any customization so we pass `null`. If you need more control over what services are tracked, there is another constructor that accepts a filter.

OPEN SESAME

Before you can use a tracker you must open it, this registers the underlying service listener and initializes the tracked list of services. This is often the thing people forget to do when they first use a service tracker and wonder why they don't see any services. Similarly, when you're done with the tracker you must close it. While the framework will automatically remove the service listener when the bundle stops, it is best to explicitly call close so that all the tracked resources can be properly cleared.

And that's all we need to do to track instances of the Log Service, we don't need to write our own listener or worry about managing long lists of references. When we need to actually use the Log Service, we just ask the tracker for the current instance:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
LogService logService = (LogService) m_logTracker.getService();
```

There are other tracker methods to get all active instances and access the underlying service references, there's even a method that helps you wait until a service appears. Often you'll find a raw service tracker is all you need, but there are times you'll want to extend it. Perhaps you want to decorate a service with additional behavior or you need to acquire or release resources as services appear and disappear. You could simply extend the `ServiceTracker` class, but you would have to be careful not to break the behavior of any methods you override. Thankfully, there is a way to extend a service tracker without subclassing it: with a "customizer" object. The `ServiceTrackerCustomizer` interface shown below provides a safe way to enhance a tracker by intercepting tracked service instances:

Listing 4.14 ServiceTrackerCustomizer interface

```
public interface ServiceTrackerCustomizer {

  public Object addingService(ServiceReference reference);              #A

  public void modifiedService(ServiceReference reference,               #B
      Object service);

  public void removedService(ServiceReference reference,                #C
      Object service);
}
```

#A matching service is being registered
#B metadata of matching service has changed
#C matching service has been removed

Like a service listener, a customizer is based on the three major events in the life of a service: adding, modifying, and removing. The `addingService()` method is where most of the customization occurs. The associated tracker will call this whenever a matching service is added to the OSGi service registry. You are free to do whatever you want with the incoming service; you could initialize some resources or wrap it in another object, for example. The object you return will be tied to the service by the tracker and returned wherever the tracker would normally return the service instance. If you decide you don't want to track a particular service instance, simply return `null`. The other two methods in the customizer are typically used for housekeeping tasks, like updating or releasing resources.

Suppose we want to decorate the Log Service, such as adding some text around the log messages. Our service tracker customizer might look something like this:

Listing 4.15 Customized tracker example – decorated Log Service

```
class LogServiceDecorator implements ServiceTrackerCustomizer {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
  public Object addingService(ServiceReference ref) {
    return new LogService() {

      public void log(int level, String message) {
        ((LogService) m_context.getService(ref)).log(level,
            "<<" + message + ">>");                              #A
      }

      public void log(int level, String message,
          Throwable exception) {}

      public void log(ServiceReference sr, int level, String message) {}
      public void log(ServiceReference sr, int level, String message,
          Throwable exception) {}
    };
  }

  public void modifiedService(ServiceReference ref, Object service) {}

  public void removedService(ServiceReference ref, Object service) {}
}
```

#A wrap code around original Log Service

All we have to do to decorate the Log Service, is pass the customizer to the tracker:

```
m_logTracker = new ServiceTracker(context, LogService.class.getName(),
    new LogServiceDecorator());
```

Now any Log Service returned by this tracker will add angle brackets to the logged message. This is a trivial example, but I hope you can see how powerful customizers can be. Service tracker customizers are especially useful in separating code from OSGi specific interfaces, since they act as a bridge connecting your application code to the service registry.

We have seen three different ways to access OSGi services; directly through the bundle context, reactively with service listeners, and indirectly using a service tracker. Which way should you choose? Well, if you only need to use a service very intermittently and don't mind using the raw OSGi API, then using the bundle context is probably the best option. At the other end of the spectrum, if you need full control over service dynamics and don't mind the potential complexity, then a service listener would be best. In all other situations you should use a service tracker, as it helps you handle the dynamics of OSGi services with the least amount of effort.

WHAT? NO ABSTRACTIONS?

If none of these options suit you and you prefer to use a higher level abstraction, such as components, this is fine too. As we mentioned at the start of this chapter it is possible to build component models on top of these core APIs. In fact, this is exactly what many people have been doing for the past few years and there are several service-oriented

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

component frameworks based on OSGi which we will discuss in chapter 11. But remember, all of these component frameworks make some subtle, but important, semantic choices when mapping components to the OSGi service model. If you ever need to cut through these abstractions and get to the real deal, you now know how.

Now we know all about OSGi services and their dynamics, let's look again at our ongoing paint program and see where it might make sense to use services.

4.4 Using services in the paint example

The last time we saw the paint example was back in 3.4 where we used an extender pattern to collect shapes. Why don't we try using a service instead? A shape service makes a lot of sense, because we can clearly define what responsibilities belong to a shape and use metadata to describe various non-functional attributes like its name and icon. Remember the first thing to define when creating a new service is the contract, so what should a shape service look like?

4.4.1 Defining a shape service

Let's use the previous interface as the basis of our new service contract, but this time instead of extension names we'll declare service property names. These names will tell the client where to find additional metadata about the shape:

```
public interface SimpleShape {

  public static final String NAME_PROPERTY = "simple.shape.name";         #A
  public static final String ICON_PROPERTY = "simple.shape.icon";         #B

  public void draw(Graphics2D g2, Point p);                               #C
}
```

#A metadata key for the shape name
#B metadata key for the shape icon
#C draw shape onto given canvas

This is not much different from the previous interface defined in 3.4. This shows how easy it is to switch over to services when programming with interfaces. With this contract in hand, we now need to update each shape bundle to publish their implementation as a service, and update the paint frame bundle to track and consume these shape services.

4.4.2 Publishing a shape service

Before we can publish a shape implementation as a service, we need a bundle context. To get the bundle context, we we need to add a bundle activator to each shape bundle, as shown in Listing 4.16.

Listing 4.16 Publishing a shape service

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
public class Activator implements BundleActivator {
  private BundleContext m_context = null;

  public void start(BundleContext context) {
    m_context = context;
    Hashtable dict = new Hashtable();

    dict.put(SimpleShape.NAME_PROPERTY, "Circle");                        #A
    dict.put(SimpleShape.ICON_PROPERTY,
        new ImageIcon(this.getClass().getResource("circle.png")));        #B

    m_context.registerService(SimpleShape.class.getName(),
        new Circle(), dict);                                              #C
  }

  public void stop(BundleContext context) {}
}
```

#A record name under correct metadata key
#B record icon under correct metadata key
#C publish the new shape service

Our shape bundles will now publish their shape services when they start, and remove

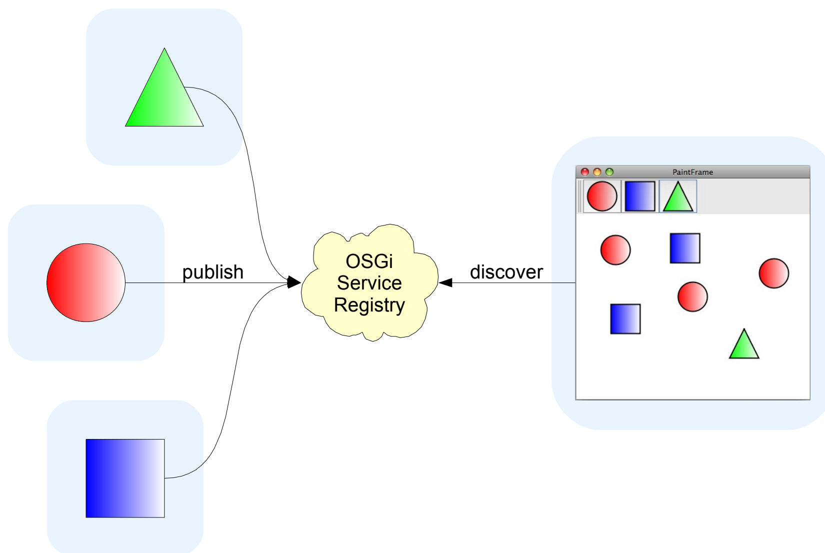


Figure 4.14 Painting with services

them when they stop. To use these shapes when painting we need to update our paint frame bundle so it will use now services instead of bundles, as shown in Figure 4.14.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

4.4.3 Tracking shape services

Remember the DefaultShape class that acted as a simple proxy to an underlying shape bundle in 3.4. When the referenced shape bundle was installed, the DefaultShape used its classes and resources to paint the shape. When the shape bundle was not installed, the DefaultShape drew a placeholder image instead. Well, we can use exactly the same approach with services, except that instead of a bundle identifier we use a service reference:

Listing 4.17 Drawing with a shape service

```
if (m_context != null) {
  try {
    if (m_shape == null) {
      m_shape = (SimpleShape) m_context.getService(m_ref);               #A
    }
    m_shape.draw(g2, p);                                                  #B
    return;
  } catch (Exception ex) {}                                               #C
}
```

#A get referenced shape service
#B draw shape – simple method call
#C use placeholder image if problem

We'll also add a dispose method to tell the framework when we're done with the service:

```
public void dispose() {
  if (m_shape != null) {
    m_context.ungetService(m_ref);
    m_context = null;
    m_ref = null;
    m_shape = null;
  }
}
```

Our new DefaultShape is now based on an underlying service reference, but how do we find these references? Remember the advice from 4.3.3; we want to use several instances of the same service, and react as they appear and disappear, but we don't want detailed control – so what we need is a ServiceTracker!

In the previous chapter, we used a BundleTracker to react as shape bundles came and went. This proved to be a good design choice, because it means our ShapeTracker class can already process shape events and trigger the necessary Swing actions. All we need to change is the source of shape events, which now come from the ServiceTracker methods:

Listing 4.18 Sending shape events from ServiceTracker methods

```
public Object addingService(ServiceReference ref) {
  SimpleShape shape = new DefaultShape(m_context, ref);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    processShapeOnEventThread(ADDED, ref, shape);
    return shape;
  }

  public void modifiedService(ServiceReference ref, Object svc) {
    processShapeOnEventThread(MODIFIED, ref, (SimpleShape) svc);
  }

  public void removedService(ServiceReference ref, Object svc) {
    processShapeOnEventThread(REMOVED, ref, (SimpleShape) svc);
    ((DefaultShape) svc).dispose();                                  #A
  }
```

#A ungets the service and clears fields

The processing code also needs to use service metadata rather than extensions:

```
String name = (String) ref.getProperty(SimpleShape.NAME_PROPERTY);
Icon icon = (Icon) ref.getProperty(SimpleShape.ICON_PROPERTY);
```

And that's all there is to it, we now have a service-based paint example! If you want to see it in action, go into the `code/chapter04/paint-example/` directory of the companion code, type `ant` to build it, and `java -jar launcher.jar bundles` to run it. The fact that we only needed to change a few files is a testament to the non-intrusiveness of OSGi services. Hopefully, you can also see how easy it would be to do this in reverse, and adapt a service-based example to use extensions. Imagine being able to decide when and where to use services in your application, without having to factor it into the initial design. The OSGi service layer gives you that ability, and the previous layers help you manage and control it. But how can the modularity and lifecycle layers help, how do they relate to the service layer?

4.5 Relating services to modularity and lifecycle

The service layer builds on top of the modularity and lifecycle layers. We've already seen one example of this, where the framework automatically unregisters services when their registering bundle stops. But there are other places where the layers interact, such as providing bundle-specific (also known as factory) services, when you should “unget” and unregister services, and how you should bundle up services. But let's start with how modularity affects what services you can see.

4.5.1 Why can't I see my service?

There may be times you ask yourself this question and wonder why even though the OSGi framework shows a particular service as registered, you cannot access it from your bundle. The answer comes back to modularity. You can only see classes from packages that are imported by your bundle. Similarly, the OSGi framework will only return references to services whose versions of registered interfaces are visible to your bundle. The reasoning

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

behind this is that you should be able to cast service instances to any of their registered interfaces without causing a ClassCastException.

But what if you want to query all services, regardless of what interfaces you can actually see? While not common, this is useful in management scenarios where you want to track third-party services you don't yet know about. To support this the OSGi framework provides a so-called All* variant of the getServiceReferences() method to return all matching services, regardless of whether their interfaces are visible to the calling bundle. For example:

```
ServiceReference[] references =
    bundleContext.getAllServiceReferences(null, null);
```

This returns references to all services currently registered in the OSGi service registry. Similarly for service listeners there is an All* extension of the ServiceListener interface, which will let you receive all matching service events. The ServiceTracker is the odd one out; there's no All* variant, to ignore visibility you just start the tracker with open(true).

We've just seen that while one bundle can see a service, another bundle with different imports might not. How about two bundles with the same imports? They would see the same service instances. What if we actually wanted them to see different instances – is it possible to customize services for each consumer?

4.5.2 Can I provide a bundle-specific service?

You may have noticed throughout this chapter we have assumed service instances are created first then published, discovered, and finally used. Or to put it another way, creation of service instances is not related to their use. But there are times when you want to create services lazily or you want to customize a service specifically for each bundle using it. An example of this is our simple Log Service implementation from 4.3. None of the Log Service methods accept a bundle or bundle context, but we might like to record details of the bundle logging the message. How is this possible in OSGi, doesn't the registerService() method expect a fully constructed service instance?

The OSGi framework defines a special interface to use when registering a service. The ServiceFactory interface acts as a marker telling the OSGi framework to treat the provided instance not as a service, but as a factory which can create service instances on-demand. The OSGi service registry uses this factory to create instances just before they are needed, when the consumer first attempts to use the service. A factory could potentially be creating a number of instances at the same time, so it must be thread-safe.

```
public interface ServiceFactory {

  public Object getService(Bundle bundle,
      ServiceRegistration registration);

  public void ungetService(Bundle bundle,
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
        ServiceRegistration registration, Object service);
}
```

The framework caches factory created service instances, so a bundle requesting the same service twice will receive the same instance. This cached instance is only removed when the bundle has completely finished with a service (that is the number of calls to "get" it match the calls to "unget" it), when the bundle has stopped, or the service factory is unregistered.

So should we always "unget" a service after we use it, like closing an I/O stream?

4.5.3 When should I "unget" a service?

We just saw that instances created from service factories are cached until the consuming bundle has finished with the service. This is determined by counting the calls to `getService()` compared to `ungetService()`. Forgetting to call "unget" can lead to instances being kept around until the bundle is stopped. Similarly, agents interrogating the framework will assume the bundle is using the service when it actually isn't. So should we always "unget" after using a service, perhaps something like the following?

```
try {
  Service svc = (Service) m_context.getService(svcRef);
  if (svc != null) {
    svc.dispatch(something);
  } else {
    fallback(somethingElse);
  }
} finally {
  m_context.ungetService(svcRef);
}
```

This would record exactly when we use the service, but what happens if we want to use it again and again in a short space of time? Services backed by factories would end up creating and destroying a new instance on every call, which could be costly. We might also want to keep the instance alive between calls if it contains session-related data. In these circumstances it makes more sense to "get" at the start of the session and "unget" at the end of the session. For long-lived sessions you would still need to track the service in case it was removed, probably using a service tracker customizer to close the session. In all other situations you should "unget" the service when finished with it.

But what about the other side of the equation? Should bundles let the framework unregister their services when they stop or should they be more pro-active and unregister services as soon as they don't want to or can't support them?

4.5.4 When should I unregister my service?

The OSGi framework does a lot of tidying up when a bundle stops – it removes listeners, releases used services, and unregisters published services. It can often feel like you don't need to do anything yourself, indeed many bundle activators have empty stop() methods.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

But there are times when it is prudent to unregister a service yourself. Perhaps you've just received a hardware notification and need to tell bundles not to use your service. Perhaps you need to perform some processing before shutting down and don't want bundles using your service while this is going on. At times like this you should remember you are in control and it is often better to be explicit than rely on the framework to clean up after you.

After that salutary message, let's finish off this section with a modularity topic that has caused a lot of heated discussion on OSGi mailing lists: where to put service interfaces.

4.5.5 Should I bundle interfaces separately?

Service interfaces are by definition decoupled from their implementations. So should they be bundled separately in their own bundle or duplicated inside each implementation bundle? OSGi supports both options, because as long as the metadata is correct it can wire the various bundles together to share the same interface. But why might you want to copy the interface inside each implementation bundle, surely that would lead to duplicated content?

Think about deploying a set of services into a framework, if each service has both an API and implementation bundle, then that doubles the number of bundles to manage. Putting the interface inside the implementation bundle means you only need to provide one JAR file. Similarly, users don't have to remember to install the API – if they have the implementation they automatically get the API for free. This sounds good, so why doesn't everyone do this?

It comes down to managing updates. Putting interfaces inside an implementation bundle means the OSGi framework could decide to use that bundle as the provider of the API package. If you then want to upgrade and refresh the implementation bundle, then all the consuming bundles will end up being refreshed, causing a wave of restarting bundles. Similarly, if you decide to uninstall the implementation, the implementation classes will only be unloaded by the garbage collector when the interface classes are no longer being used (because they share the same class loader).

In the end, there is no single right answer. Each choice has consequences of which you should be aware. Just as with other topics we've discussed: service visibility, service factories, using "unget", and unregister; you need to know the possibilities to make an informed choice. Whatever you decide, we can all agree that services are an important feature of OSGi.

4.6 Framework services

Services are such an important feature that they are actually used throughout the OSGi specification itself. By using services to extend the framework, the core API can be kept very lean and clean. When reading the specification you will realize a lot of services are actually optional, and we'll discuss these "compendium" services in a moment, but first we'll take a quick look at the set of core services (most of which) every OSGi framework must implement and provide.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Table 3 Framework services covered in this section

Service	Type	Description
Package Admin	Core	Manage bundle updates and discover who exports what
Start Level	Core	Query and control framework and bundle start levels
URL Handlers	Core	Dynamic URL stream handling
Permission Admin	Core	Manage bundle and service permissions
HTTP	Compendium	Put simple servlets and resources onto the web
Event Admin	Compendium	Topic-based "pub-sub" event model
Configuration Admin	Compendium	Manage and persist configuration data
User Admin	Compendium	Role-based authentication and authorization

4.6.1 Core services

Package Admin Service

The OSGi Package Admin Service, which we discussed in chapter 3, provides a selection of methods to discover details about exported packages and the bundles that export and/or import them. You can use this service to trace dependencies between bundles at execution time, which can help when upgrading because you can see what bundles might be affected by the update. The Package Admin Service also provides methods to refresh exported packages, which may have been removed or updated since the last refresh, and to explicitly resolve specific bundles.

Start Level Service

The OSGi Start Level Service lets you programmatically query and set the start level for individual bundles as well as the framework itself. You can use start levels to deploy an application, or roll-out a significant update, in controlled stages.

URL Handlers Service

The OSGi URL Handlers Service adds a level of dynamism to the standard Java URL process. The Java specification unfortunately only allows one `URLStreamHandlerFactory` to be set during the lifetime of a JVM, so the framework will attempt to set its own implementation at startup. If this is successful then this factory will dynamically provide URL stream handlers and content handlers, based on implementations registered with the OSGi service registry.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

(Conditional) Permission Admin Service

There are actually two services dealing with permissions in OSGi: the Permission Admin Service, which deals with permissions granted to specific bundles, and the Conditional Permission Admin Service, which provides a more general purpose and fine-grained permission model based on conditions. Both of these services build on the standard Java2 security architecture. While these two are core services, they are actually optional, since all security-related features are optional in the OSGi specification.

So we now know which core services we can expect to see in an OSGi framework, but what about the optional "compendium" services? What sort of capabilities do they cover?

4.6.2 Compendium services

In addition to the core services, the OSGi Alliance defines a set of non-core standard services called the "compendium" services. While the core services are typically available by default in a running OSGi framework, the compendium services will not. In fact, keeping with our desire for modularity, we wouldn't want them to be included by default since this would lead to bloated systems. Instead, these services are provided as separate bundles by framework implementers or other third-party parties and typically work on all frameworks.

We have already seen one example of a compendium service, the Log Service from 4.3 which provided a simple logging API. This is one of the more well-known compendium services. Let's take a brief look at other examples which we'll be using later on in the book:

HTTP Service

The OSGi HTTP Service supports registration of servlets and resources under named aliases. These aliases are matched against incoming URI requests and the relevant servlet or resource is used to construct the reply. You can authenticate incoming requests using either standard HTTP/HTTPS, the OSGi User Admin service, or your own custom approach. The current HTTP Service is based on version 2.1 of the servlet specification [ref4], which means it doesn't cover servlet filters, event listeners, or JSPs. Later versions of the HTTP Service specification should address this and there are some implementations that already support these additional features [ref5].

Event Admin Service

The OSGi Event Admin Service provides a basic "publish-subscribe" event model. Each event consists of a topic, which is basically a semi-structured string, and set of properties. Event handlers are registered as services and can use metadata to describe which topics and properties they are interested in. Events can be sent synchronously or asynchronously and are delivered to matching event handlers by using the whiteboard pattern, which we discussed earlier in 4.1.3. Other types of OSGi events (like framework, bundle, service, and log events) are mapped and re-published by the Event Admin Service implementation.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Configuration Admin Service

The OSGi Configuration Admin Service delivers configuration data to those services with persistent identifiers ("service.pid") that implement the `ManagedService` interface – or `ManagedServiceFactory`, if they want to create a new service instance per configuration. These so-called "configuration targets" accept configuration data in the form of a dictionary of properties. Management bundles, which have been granted permission to configure services, can use the Configuration Admin Service to initialize and update configurations for other bundles. Non-management bundles can only update their own configurations. The Config Admin Service is pluggable, and can be extended by registering implementations of the `ConfigurationPlugin` interface with the OSGi service registry.

User Admin Service

The OSGi User Admin Service provides a role-based model for authentication (checking credentials) and authorization (checking access rights). An authenticating bundle would use the User Admin Service to pre-populate the required roles, groups, and users along with identifying properties and credentials. This bundle can then query the User Admin Service at a later date to find users, check their credentials, and confirm their authorized roles. It can then decide how to proceed based on the results of these checks.

This is just a short sample of the most popular compendium services, a complete table can be found in Appendix [ref6]. You can also read detailed specifications of each service in the OSGi Service Compendium book [ref7].

4.7 Summary

That was a lot of information to digest, so don't worry if you got a bit woozy. Let's try and summarize this chapter together:

- A service is "work done for another".
- Service contracts define responsibilities, and match consumers with providers.
- Services encourage a relaxed, pluggable, interface-based approach to programming
- You don't mind where a service comes from as long as it meets the contract.
- The best place to use services is between replaceable components.
- Think carefully before using services in tightly-coupled or high performance code.
- Services can replace the listener pattern with a much simpler whiteboard pattern.
- OSGi services use a "publish-find-bind" model.
- OSGi services are truly dynamic and can disappear at any time.
- The easiest and safest approach is to use the OSGi `ServiceTracker` utility class.
- OSGi services build on and interact with the previous module and lifecycle layers.
- OSGi defines "core" framework services and additional "compendium" services.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Finally, as we close the chapter on services, we can all now agree that they are not just limited to distributed or remote applications. There is a huge benefit to applying a service-oriented design to a purely local, single-JVM application and we honestly hope you get the opportunity to experience this in your next project.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

5
Delving Deeper into Modularity

In the preceding chapters we covered a myriad of details about the three layers of the OSGi framework. Believe it or not, we didn't cover everything; instead, our focus was on explaining the common features, best practices, and framework behavior necessary to help you understand and start employing OSGi in your own projects. Depending on the project, the aforementioned features and best practices may not be sufficient. This can be especially true when it comes to legacy situations, where you are not able to make sweeping changes to your existing code base. Sometimes the clean theory of modularity conflicts with the messiness of reality, so occasionally compromises are needed to simply get things moving or to meet objectives.

In this chapter, we investigate more aspects of OSGi's modularity layer. We will look into simple features, such as making imported packages a little more flexible, and into more complicated ones, such as splitting Java packages across multiple bundles or breaking a single bundle into pieces. You will likely not need to use most of the features described in this chapter as often as the preceding ones and, in fact, if you are you should likely review your design since it is likely missing the benefits brought by more streamlined modularization approach. With that said, it is worthwhile to be aware of these advanced features of OSGi modularization and the circumstances under which they are useful. To assist in this endeavor, we will introduce use cases and examples to help you understand when and how to apply them.

This chapter is not strictly necessary for understanding subsequent chapters, so feel free to postpone reading it until later. Otherwise, let's dig in.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

5.1 Managing your exports

From what we've learned so far, exporting a package from a bundle is fairly simple: just include it in the `Export-Package` header and potentially include some attributes. This does not cover all the details of exporting packages. In the following subsections we discuss other aspects, like importing exported packages, implicit attributes, mandatory attributes, class filtering, and duplicate exports.

5.1.1 Importing your exports

In chapter 2, we learned how `Export-Package` exposes internal bundle classes and how `Import-Package` makes external classes visible to internal bundle classes. This seems to be a nice division of labor between the two. We might even assume the two are mutually exclusive. In other words, we might assume a bundle exporting a given package cannot import it also and vice versa. In many module systems this would be a reasonable assumption, but for OSGi it is incorrect. A bundle that imports a package it exports is actually a special case in OSGi, but what exactly does it mean? The answer to this question is both philosophical and technical.

The original vision of the OSGi service platform was to create a lightweight execution environment where dynamically downloaded bundles collaborate. In an effort to meet the "lightweight" aspect of this vision, these bundles collaborated by sharing direct references to service instances. Using direct references means bundles collaborate via normal method calls, which is very lightweight. As a byproduct of using direct references, bundles must share the Java class associated with shared service instances. As we have learned, OSGi has code sharing covered in spades with `Export-Package` and `Import-Package`. Still, there is an issue lurking in here, so let's examine a collaboration scenario more closely.

Imagine bundle A wants to use an instance of class `javax.servlet.Servlet` from another bundle B. As we now understand, in their respective metadata bundle A would import package `javax.servlet` and bundle B would export it. Makes sense. Now imagine another bundle C also wants to share an instance of class `javax.servlet.Servlet` with bundle A. It has two choices at this point, it can:

1. Not include a copy of package `javax.servlet` in its bundle JAR file and import it instead.
2. Include a copy of package `javax.servlet` in its bundle JAR file and also export it.

If the option 1 approach is taken (see Figure 5.1), then bundle C cannot be used unless bundle B is present, since it has a dependency on package `javax.servlet` and only bundle B provides the package (i.e., it is not self contained).

On the other hand, if the option 2 approach is taken (see Figure 5.2), then bundle C is self contained and both B and C can be used independently. But what happens if you want bundle A to interact with the `javax.servlet.Servlet` objects from bundles B and C at the same time? It cannot do so. Why?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

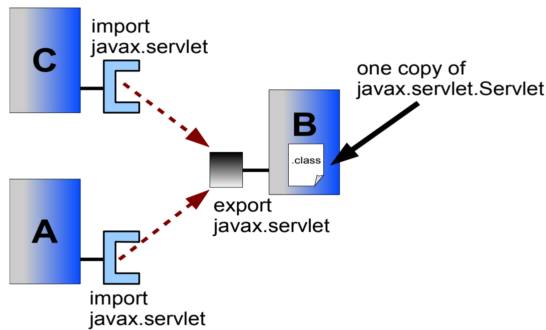


Figure 5.1 If bundle B imports from C, then both can share Servlet instances with A since there is only one copy of the Servlet class, but this creates a dependency for C on B

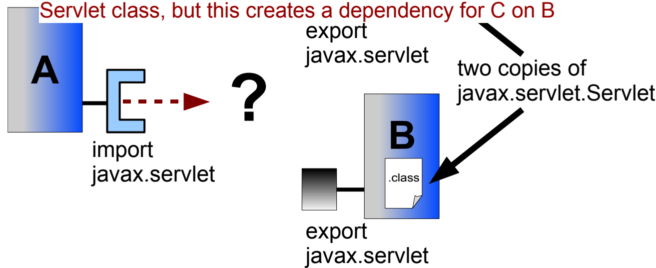


Figure 5.2 If B and C each have a copy of the Servlet class, then A can only share Servlet instances with either B or C since it can only see one definition of a class

If bundles B and C both include and export a copy of the `javax.servlet` package, then there will actually be two copies of the `javax.servlet.Servlet` class. Bundle A would not be able to use both instances, since they would come from different copies of the `javax.servlet.Servlet` class and would be incompatible. Due to this incompatibility, the OSGi framework only allows bundle A to see one copy, which means A would not be able to collaborate with both B and C at the same time.

It is not important for you to completely understand these arcane details of Java class loaders, especially since OSGi technology tries to relieve you from having to worry about class loaders in the first place. The important point is to understand the issues surrounding the two options above: option 1 results in bundle C requiring B to be present, while option 2 results in bundle A not being able to see the shared object instances from bundles B and C at the same time. This gets us to the crux of OSGi's special case for allowing a bundle to import a package it also exports.

Neither choice is very satisfactory. The solution devised by the OSGi specification is to allow a bundle to both import and export the same package (see Figure 5.3). In this case, the bundle contains the given package and its associated classes, but it may not actually end

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

up using them. A bundle importing and exporting the same package is actually offering the OSGi framework a choice; it allows the framework to treat it as either an importer or an exporter of the package, whichever seems appropriate at the time the framework makes the decision. Another way to think about this is it defines a substitutable export, where the framework is free to substitute the bundle's export with an exported package from another bundle. Returning to our example, both bundles B and C can include a copy of package `javax.servlet` with both importing and exporting it, knowing they will work independently or together.

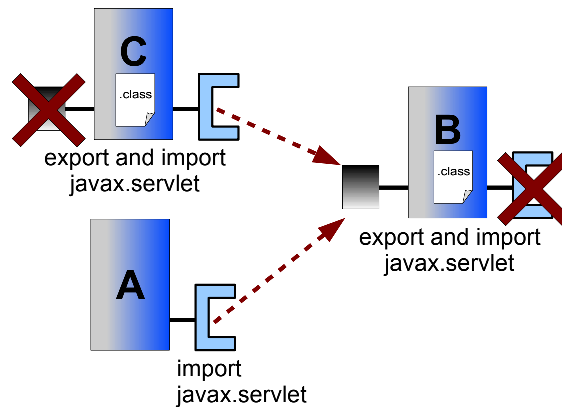


Figure 5.3 B and C can both export and import the Servlet package, which makes it possible for the framework to choose to substitute packages so all bundles use a single class definition

Admittedly, this may seem odd, but as our discussion here illustrates, to simplify the OSGi vision of a collaborative environment it is necessary to make sure bundles use the same class definitions. In fact, up until the OSGi R4 specification, `Export-Package` implicitly meant `Import-Package` too. The R4 specification removed this implicitness, making it a requirement to have a separate `Import-Package` to get a substitutable export, but this did not lessen the importance of doing so in cases where collaboration is desired. An interesting side effect of this is the possibility of metadata like this:

```
Export-Package: javax.servlet; version="2.4.0"
Import-Package: javax.servlet; version="2.3.0"
```

This is not a mistake. A bundle may include a copy of a given package at a specific version level, but may work with a lower range. This could make the bundle useful in a wider range of scenarios, since it can still work in an environment where an older version of the package must be used.

When to import our exports?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

So the question on your mind now probably is, “With all of these benefits, shouldn't I just make all of my exports substitutable?” Not necessarily. If your bundle is purely a library bundle providing packages and nothing more, then it is not necessary to import its packages too. Another rule of thumb to consider is whether another bundle could meaningfully provide the packages; if the packages are specific to your application, then only exporting them is probably sufficient. A common situation when importing and exporting a package is useful is when using an interface-based development approach.

In interface-based programming, which is the foundation of the OSGi service approach, you assume there are potentially multiple implementations of well-known interfaces. You may want to package the interfaces into their implementations to keep them self contained. In this case, to ensure interoperability, the bundles should import and export the interface packages. Since the whole point of OSGi services is to foster collaboration among bundles using interface-based programming, the choice between importing only or exporting and importing interface packages is fairly common.

You do have an alternative approach, which is to always package your collaborative interface packages into a separate bundle. By never packaging your interfaces in a bundle providing implementations, you can also be sure all implementations can be used together because all implementations will import the shared packages. If you follow this approach, then none of your implementations will be self contained, since they will all have external dependencies on the shared packages. The trade off is deciding whether you want more bundles with dependencies among them or fewer self-contained bundles with some content overlap.

Next we will look at implicit export attributes. Unlike importing exported packages, which give the framework more flexibility when resolving imports, implicit export attributes limit how the framework resolves an import.

5.1.2 Implicit export attributes

Generally speaking, OSGi regards the same package exported by multiple bundles as being completely equivalent if the package versions are the same. This is beneficial when it comes to dependency resolution, since it is possible for the framework to satisfy an import for a given package from any available matching exporter. In certain situations, you may not wish to have your bundle's imports satisfied by an arbitrary bundle; instead, you may want to import from a specific bundle. For example, perhaps you patched a bug in a common open source library and you do not want to risk using a non-patched version of the packages. OSGi supports this through implicit export attributes. Consider the following bundle manifest snippet:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: my.javax.servlet
Bundle-Version: 1.2.0
Export-Package: javax.servlet; javax.servlet.http; version="2.4.0"
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

As we learned in chapter 2, this metadata exports the packages `javax.servlet` and `javax.servlet.http` with a `version` attribute of the specified value. Additionally, the framework implicitly attaches the bundle's symbolic name and version to all packages exported by a bundle. Therefore, the previous metadata conceptually looks like this (also depicted in Figure 5.4):

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: my.javax.servlet
Bundle-Version: 1.2.0
Export-Package: javax.servlet; javax.servlet.http; version="2.4.0";
 bundle-symbolic-name="my.javax.servlet"; bundle-version="1.2.0"
```

Although this is conceptually what is happening, do not try to explicitly specify the

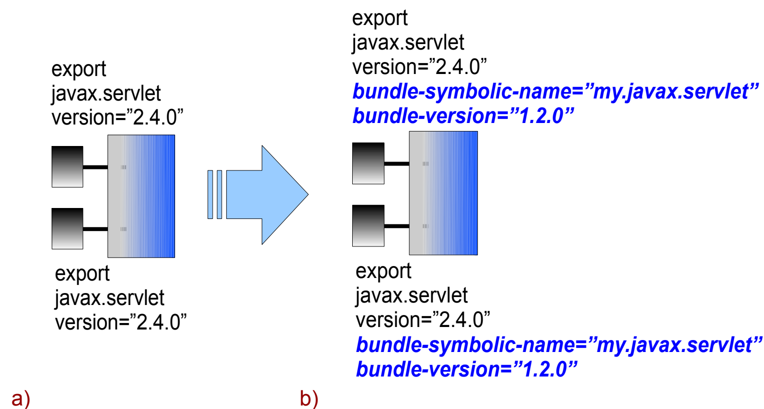


Figure 5.4 a) Our metadata declares explicit attributes which are attached to our bundle's exported packages, but b) the framework also implicitly attaches attributes explicitly identifying from which bundle the exports come.

`bundle-symbolic-name` and `bundle-version` attributes on your exports. These attributes can only be specified by the framework; the framework considers any bundle explicitly specifying these attributes as invalid and throws an exception during installation. With these implicit attributes, it is possible for us limit the dependency resolution of an imported package to specific bundles. For example, another bundle may contain the following snippet of metadata:

```
Import-Package: javax.servlet; bundle-symbolic-name="my.javax.servlet";
 bundle-version="[1.2.0, 1.2.0]"
```

In this case, the importer limits its dependency resolution to a specific bundle by specifying its symbolic name with a precise version range. As you can imagine, this makes the dependency a lot more brittle, but under certain circumstances this may be desired.

You might be thinking implicit export attributes aren't completely necessary to control how import dependencies are resolved. You'd be correct. Good old arbitrary attributes can

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

also be used to achieve the same effect, just make sure you make your attribute name and/or value sufficiently unique. For example, we could modify our exporting manifest like this:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: javax.servlet
Bundle-Version: 1.2.0
Export-Package: javax.servlet; javax.servlet.http; version="2.4.0";
 my-provider-attribute="my.value.scheme"
```

In this case, the importer simply needs to specifying the corresponding attribute name and value on its `Import-Package` declaration. There is actually an advantage to using this approach if you are in a situation where you must have brittle dependencies: it is not as brittle as implicit attributes. The reason is you are able to refactor your exporting bundle without impacting importing bundles, since these attribute values are not tied to the containing bundle. On the downside side, arbitrary attributes are easier for other bundles to imitate, even though there are no guarantees either way.

In short, it is best to avoid brittle dependencies, but at least now we understand how both implicit and arbitrary export attributes allow importing bundles to have a say in how their dependencies are resolved. Thinking about the flip side, it may also occasionally be necessary for exporting bundles to have some control over how importing bundles are resolved. Mandatory attributes can help us here.

5.1.3 Mandatory export attributes

The OSGi framework promotes arbitrary package sharing among bundles. As we discussed in the last subsection, there are some situations where this isn't always desired. Up until now, the importing bundle appears to be completely in control of this situation, since it declares the matching constraints for dependency resolution. For example, consider the following metadata snippet for importing a package:

```
Import-Package: javax.servlet; version="[2.4.0,2.5.0)"
```

Such an import declaration will match any provider of `javax.servlet` as long as it is in the specified version range. Now consider the following metadata snippet for exporting a package in another bundle:

```
Export-Package: javax.servlet; version="2.4.1"; private="true"
```

Will the imported package match this exported package? Yes, it will, as depicted in Figure 5.5. The name of the attribute, `private`, may have tricked you into thinking otherwise, but it is just an arbitrary attribute and has no meaning (if it did have meaning to the framework, it would likely be a directive not an attribute). When it comes to matching an import to an export, only the attributes mentioned on the import declaration are compared against the attributes on the export declaration. So in this case, the import mentions the package name and a version range, which happen to match the exports package name and version. The `private` attributes is not even considered.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

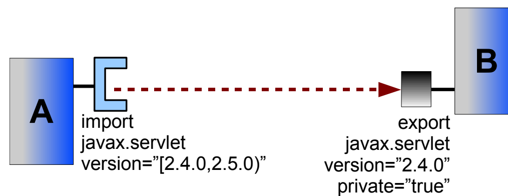


Figure 5.5 Only attributes mentioned in the imported package declaration impact dependency resolution matching, any attributes mentioned only in the exported package declaration are ignored.

In some situations, you may wish to have a little more control in your exporting bundle. For example, maybe you are exposing a package containing non-public API or you have modified a common open source library in an incompatible way so you don't want unaware bundles to inadvertently match your exported packages. The OSGi specification provides this capability through the notion of a mandatory export attribute and specified with the `mandatory` export package directive.

MANDATORY DIRECTIVE

The `mandatory` export package directive specifies a comma-delimited list of attribute names which any importer must match in order to be wired to the exported package.

To see how mandatory attributes work, let's modify the previous snippet to export its package using one:

```
Export-Package: javax.servlet; version="2.4.1"; private="true";
 mandatory:="private"
```

We have added the `mandatory` directive to the exported package. In this example we declared the `private` attribute as mandatory. Any export attribute declared as mandatory places a constraint on importers. If the importers do not specify a matching value for the attribute, then they do not match. In other words, the export attribute cannot be ignored, as depicted in Figure 5.6. The need for mandatory attributes does not arise often; we will see some other use cases in the coming sections. Until then, let's look into another more fine-grained mechanism bundles have to control what is exposed from their exported packages.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

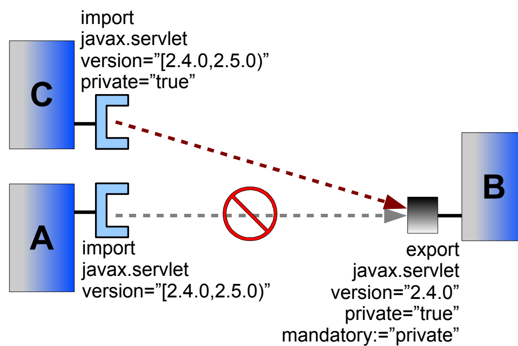


Figure 5.6 If an export attribute is declared as mandatory, then importing bundles must declare the attribute and matching value or else it will not match when the framework resolves dependencies.

5.1.4 Export filtering

In chapter 1, we discussed the limitations of Java's rules for code visibility. There is no way to declare "module" visibility, so `public` visibility must be used for classes accessed across packages. This is not necessarily problematic if you can keep your public and private API in separate packages, since bundles have the ability to hide packages by simply not exporting them. Unfortunately, this is not always possible and in some cases you end up with a `public` implementation class inside of a package exported by the bundle. To cope with this situation, OSGi provides include and exclude export filtering directives for fine-grained control over the classes exposed from your bundle's exported packages.

EXCLUDE/INCLUDE DIRECTIVES

The `exclude` and `include` export package directives specify a comma-delimited list of class names to exclude or include from the exported package, respectively.

To see how we can use these directives, consider a hypothetical bundle containing a package (e.g., `org.foo.service`) with a service interface (`public class Service`), an implementation of the service (package private `class ServiceImpl`), and a utility class (`public class Util`). In this hypothetical example, the utility class is part of the private API. It is included in this package due to dependencies on the service implementation and is public because it is used by other packages in the bundle. We need to export the `org.foo.service` package, but we do not want to expose the `Util` class. In general, we should avoid such scenarios, but the following metadata snippet illustrates how we can do this with export filtering:

```
Export-Package: org.foo.service; version="1.0.0"; exclude:="Util"
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

This exported package behaves like any normal exported package as far as dependency resolution is concerned, but at execution time it filters the `Util` class from the package so importers cannot access it as shown in Figure 5.7. A bundle attempting to load the `Util` class will receive class not found exception. The value of the directive specifies only class names; the package name must not be specified, nor should the `.class` portion of the class file name. The "`*`" character is also supported as a wild card, so it is possible to exclude all classes with names matching "`*Impl`", for example.

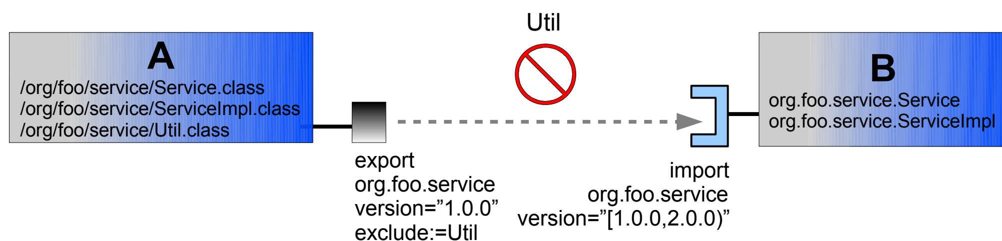


Figure 5.7 Bundle A exports the org.foo.service package but excludes the "Util" class, when bundle B imports the org.foo.service package from bundle A it can only access the "Service" and "ServiceImpl" classes.

In some cases it might be easier to specify which classes are allowed instead of which ones are disallowed. For those situations, the `include` directive is available. It specifies which classes the exported package should expose. The default value for the `include` directive is "`*`", while the default value for the `exclude` directive is an empty string. It is possible to specify both the `include` and `exclude` directive for a given exported package. A class is only visible if it is matched by an entry in the `include` directive and not matched by any entry in the `exclude` directive.

You should definitely strive to separate your public and private API into different packages so these mechanisms aren't needed, but they are here to get you out of a tough spot when you need them. Next, we will move onto another mechanism to help us manage our API.

5.1.5 Duplicate exports

A given bundle can only see one version of a given class while it executes. Knowing this, it is not surprising to learn that bundles are not allowed to import the same package more than once. What you may find surprising is OSGi allows a bundle to export the same package more than once. For example, the following snippet of metadata is perfectly valid:

```
Export-Package: javax.servlet; version="2.3.0",
 javax.servlet; version="2.4.0"
```

How is it possible you ask? The trick is the bundle doesn't actually contain to separate sets of classes for the two exported packages. In reality, the bundle is masquerading the same set of classes as different packages. Why would it do this? If we expound on the above

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

snippet, maybe we have unmodifiable third-party bundles with explicit dependencies on `javax.servlet` version `2.3.0` in our application. Since version `2.4.0` is backwards compatible with version `2.3.0`, we can use duplicate exports to allow our bundle to stake a backwards compatibility claim. In the end, all bundles requiring either version of `javax.servlet` can resolve, but they all use the same set of classes at execution time as depicted in Figure 5.8.

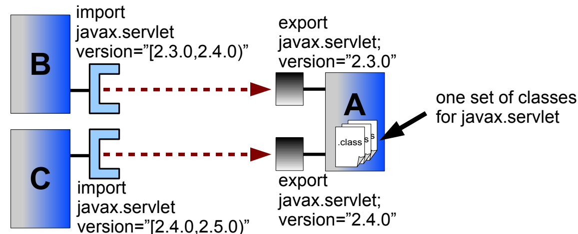


Figure 5.8 A bundle can export the same package multiple times, but this is only a form of masquerading and only one set of classes exist for the package in the bundle

As with exporting filtering, this is another mechanism to manage our API. We can take this further and combine it with some of the other mechanisms we've learned about in this section for additional API management techniques. Generally, we don't want to expose our bundle's implementation details, but sometimes we do want to expose implementation details to some bundles; this is similar to the "friend" concept in C++. A friend is allowed to see implementation details while non-friends cannot. To achieve something like this in OSGi, we need to combine mandatory export attributes, export filtering, and duplicate exports. To illustrate, let's return to our example from export filtering:

```
Export-Package: org.foo.service; version="1.0.0"; exclude:="Util"
```

In this example, we excluded our `Util` class, since it was an implementation detail. This is the exported package our non-friends should use. For our friends we need to export the package without filtering, such as:

```
Export-Package: org.foo.service; version="1.0.0"; exclude:="Util",
 org.foo.service; version="1.0.0"
```

Now we have one export hiding the `Util` class and one exposing it. So how do we control who gets access to what? That's right, mandatory export attributes. We need the following:

```
Export-Package: org.foo.service; version="1.0.0"; exclude:="Util",
 org.foo.service; version="1.0.0"; friend="true"; mandatory:="friend"
```

Only bundles explicitly importing our package with the `friend` attribute and matching value will see the `Util` class. Clearly, this is not a very strong sense of friendship, since any bundle can specify the `friend` attribute, but at least it requires an opt-in strategy to using implementation details signaling that the importer is willing to live with the consequences of potential breaking changes in the future. Best practice dictates avoiding the friendship

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

concept, since it weakens modularity. If API is valuable enough to export it, then you should consider making it public API.

We have now explored all of the extra export capabilities provided by the OSGi framework. These extra capabilities represent advanced use cases which allow us to handle some of the more thorny edge cases of operating in a modular environment. So this covers exports, but what about imports? Surely if there are all these extra capabilities of exports there must be more to learn about imports? Yes, there is. In the next section, we will learn how to make importing packages a little more flexible which again can provide us with some wiggle room when trying to get legacy Java applications to work in an OSGi environment.

5.2 Loosening your imports

Explicitly declared imports are great, since they allow us to more easily reuse code because we know its requirements and can automate dependency resolution. This gives us the benefit of being able to detect misconfigurations early, rather than receiving various class loading and class cast exceptions at execution time. On the other hand, explicitly declared imports are somewhat constraining, since the framework uses them to strictly control whether or not our code can be used; if an imported package cannot be satisfied, the framework does not allow the associated bundle to transition into the `RESOLVED` state. Additionally, to import a package we have to know the name of a package in advance, but this is not always possible.

What can we do in these situations? The OSGi framework provides two different mechanisms for dealing with such situations: optional and dynamic imports. Let's look into how each of these can help as well as compare and contrast them.

5.2.1 Optional imports

Sometimes a given package imported by a bundle is not strictly necessary for it to function properly. Consider an imported package for a non-functional purpose, like logging. If our bundle uses logging, even if no logging package is available in the framework, it can continue to function properly, it just won't be able to do any logging. To express this, OSGi provides the `resolution` directive to mark imported packages as optional.

RESOLUTION DIRECTIVE

The `resolution` import package directive can have a value of `mandatory` or `optional` to indicate whether the imported package is required to successfully resolve the bundle.

Consider the following metadata snippet:

```
Import-Package: javax.servlet; version="2.4.0",
 org.osgi.service.log; version="1.3.0"; resolution:="optional"
```

This import statement declares dependencies on two packages, `javax.servlet` and `org.osgi.service.log`. The dependency on the logging package is optional, as indicated by the use of the `resolution` directive with `optional` value. This means the bundle can

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

be successfully resolved even if there is no `org.osgi.service.log` package available. Attempts to use the class at execution time will result in a `ClassNotFoundException`. All imported packages have a resolution associated with them, but the default value is `mandatory`. We'll look at how this is used in practice in section 5.2.4, but for now let's consider the other tool in the box, called dynamic imports.

5.2.2 Dynamic imports

Certain Java programming practices make it difficult to know all the packages a bundle may need at execution time. A prime example is locating a JDBC driver. Often, the name of the class implementing the JDBC driver is a configuration property or is supplied by the user at execution time. Since our bundle can only see classes in packages it imports, how is it possible for it to import an unknown package name? This sort of situation arises when dealing with service provider interface (SPI) approaches, where a common interface is known in advance, but not the name of the class implementing it. To capture this, OSGi allows a bundle to dynamically import packages.

> **DYNAMICIMPORT-PACKAGE**
>
> A comma-separated list of packages needed at execution time by internal bundle code from other bundles, but not needed at resolve time. The package names can be a "*" wildcard or can include a trailing "`.*`" as a recursive sub-package wildcard.

You may have expected from the previous examples that dynamic imports would be handled by an import package directive as well, but they are sufficiently different to warrant their own metadata header. In the most general sense, a dynamic import is expressed in the bundle metadata like this:

```
DynamicImport-Package: *
```

Since the intended use case is for gaining access to unknown packages, the use of wildcards is necessary to match any package. The above snippet will dynamically import any package needed by the bundle. When the wildcard is used at the end of a package name (e.g., "`org.foo.*`"), it matches all sub-packages recursively, but does not match the specified root package. Given the dynamic and open-ended nature of dynamic imports, it is important to understand precisely when in the class search order of a bundle they are searched. They appear in the class search order as follows:

1. Requests for classes in "`java.`" packages are delegated to the parent class loader; searching stops with either a success or failure (section 2.5.4?).
2. Requests for classes in an imported package are delegated to the exporting bundle; searching stops with either a success or failure (section 2.5.4?).
3. The bundle class path is searched for the class; searching stops if found, but continues to the next step with a failure (section 2.5.4?).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

4. Requests for classes matching a dynamically imported package are delegated to an exporting bundle if one is found; searching stops with either a success or failure.

As you can see, dynamic imports are only attempted as a last resort, but once a dynamically imported package is resolved and associated with the importing bundle, then it behaves just like a statically imported package. Future requests for classes in the same package will be serviced in step 2.

Dynamic imports are alluring to new OSGi programmers, because they provide similar behavior to standard Java programming where everything available on the class path is visible to the bundle. Unfortunately, this approach is not modular and does not allow the OSGi framework to verify whether dependencies are satisfied in advance of using your bundle. As a result, dynamically importing packages should be seen as bad practice, except for explicitly dealing with SPI approaches.

It is also possible to use dynamically imported packages in a fashion more similar to optionally imported packages by specifying additional attributes like normal imported packages:

```
DynamicImport-Package: javax.servlet.*; version="2.4.0"
```

Or even:

```
DynamicImport-Package: javax.servlet; javax.servlet.http; version="2.4.0"
```

In the first case, all sub-packages of `javax.servlet` of version `2.4.0` are dynamically imported, while in the second only the explicitly mentioned packages are dynamically imported. More precise declarations such as this often make less sense when using dynamic imports, since the general use case is for unknown package names.

So, we apparently have two different ways of loosening bundle imports, let's compare and contrast them a little more closely.

5.2.3 Optional versus dynamic imports

As mentioned in the previous subsections, there are intended use cases for both optional and dynamic imports, but the functionality they provide overlaps. To better understand each, we will look into the similarities and differences; let's start with the similarities.

Both are used to express dependencies on packages which may or may not be available at execution time. While this is the specific use case for optional imports, it is actually a byproduct of dynamic imports. Either way, this has the following impact:

1. Optional/dynamic imports never cause a bundle to be unable to resolve.
2. Your bundle code must be prepared to catch `ClassNotFoundException`s for the optionally/dynamically imported packages.

Only normally imported packages (i.e., mandatory imported packages) impact bundle dependency resolution. If a mandatory imported package cannot be satisfied, then the bundle cannot be resolved and used. Neither optionally or dynamically imported packages are required to be present when resolving dependencies. For optional imports, this is the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

whole point; they are optional. For dynamic imports, they are not necessarily optional, but since they are unknown a priori, it is not possible for the framework to enforce that they exist.

Since the packages may not exist in either case, the logical consequence is the code in any bundle employing either mechanism must be prepared to catch `ClassNotFoundException`s when attempting to access classes in the optionally or dynamically imported packages. This is typically the sort of issue the OSGi framework tries to help us avoid with explicit dependencies; we shouldn't be dealing with class loading issues as developers. By now you must be wonder what's the difference between the two? It has to do with when the framework tries to resolve the dependencies.

The framework attempts to resolve an optionally imported package once when the associated bundle is resolved. If the import is satisfied, then the bundle has access to the package. If not, then the bundle does not and will not ever have access to the package unless it is re-resolved. For a dynamically imported package, the framework attempts to resolve it at execution time when the bundle's executing code tries to use a class from the package. Further, the framework keeps trying to resolve the package each time the bundle's executing code tries to use classes from it until it is successful. Once it is successful, then the bundle is wired to the provider of the package and it behaves like a normal import from that point forward. Let's look at how we could use these mechanisms in an example for logging, which is often an optional activity for bundles.

5.2.4 Logging example

Logging is typically a good example of optional functionality. The OSGi specification defines a simple logging service, which we might want to use in our bundles, but we cannot be certain it will always be available. One way to deal with this uncertainty is to create a simple "proxy" logger which uses the logging service if available or prints to standard output if not. Our first example will use an optional import for the `org.osgi.service.log` package. The simple proxy logger code is depicted in Listing 5.1.

Listing 5.1 Simple proxy logger using optional import

```
public class Logger {
  private final BundleContext m_context;
  private final ServiceTracker m_tracker;
  public Logger(BundleContext context) {                                  #1
    m_context = context;
    m_tracker = init(m_context);
  }
  private ServiceTracker init(BundleContext context) {                    #2
    ServiceTracker tracker = null;
    try {
      tracker = new ServiceTracker(                                       #3
        context, org.osgi.service.log.LogService.class.getName(), null);
      tracker.open();
    } catch (NoClassDefFoundError error) { }                              #4
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
      return tracker;
    }
    public void close() {                                                    #5
      if (m_tracker != null) {
        m_tracker.close();
      }
    }
    public void log(int level, String msg) {                                 #6
      boolean logged = false;
      if (m_tracker != null) {                                               #7
        LogService logger = (LogService) m_tracker.getService();
        if (logger != null) {                                                #8
          logger.log(level, msg);                                            #9
          logged = true;
        }
      }
      if (!logged) {
        System.out.println("[" + level + "] " + msg);                       #10
      }
    }
  }
```

The proxy logger has a constructor (#1) which takes the `BundleContext` object to track log services, an `init()` method (#2) to create a `ServiceTracker` for log services, a `close()` method (#5) to stop tracking log services, and a `log()` method (#6) for logging messages. Looking more closely at the `init()` method, at (#3) we try to use the logging package to create a `ServiceTracker`. Since we are optionally importing the logging package, we surround it in a try-catch block at (#4). If an exception is thrown we set our tracker to `null`, otherwise we end up with a valid tracker. When a message is logged, we check if we have a valid tracker at (#7). If so, we try to log to a log service. Even if we have a valid tracker, it doesn't mean we actually have a log service, which is why we verify it at (#8). If we have a log service we use it at (#9), otherwise we log to standard output at (#10). The important point is we only attempt to probe for the log package once with a single call to `init()` from the constructor, since an optional import will never be satisfied later if it is not satisfied already. Our bundle activator is depicted in Listing 5.2.

Listing 5.2 Bundle activator creating the proxy logger

```
  public class Activator implements BundleActivator {
    private volatile Logger m_logger = null;
    public void start(BundleContext context) throws Exception {
      m_logger = new Logger(context);                                    #1
      m_logger.log(4, "Started");
      …
    }
    public void stop(BundleContext context) {
      m_logger.close();                                                      #2
    }
  }
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

When the bundle is started, we create an instance of our proxy logger at (#1) which will be used throughout our bundle for logging. Although not depicted here, the bundle passes a reference or somehow provides access to the logger instance to any internal code needing a logger at execution time. When the bundle is stopped, we invoke `close()` on the proxy logger at (#2), which stops its internal service tracker, if necessary. The manifest for our logging bundle is:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: example.logger
Bundle-Activator: example.logger.Activator
Import-Package: org.osgi.framework, org.osgi.util.tracker,
 org.osgi.service.log; resolution:=optional
```

How would this example change if we wanted to treat the logging package as a dynamic import? The impact to the `Logger` class is depicted in Listing 5.3.

Listing 5.3 Simple proxy logger using dynamic import

```
public class Logger {
  private final BundleContext m_context;
  private ServiceTracker m_tracker;                                        #1
  public LoggerImpl(BundleContext context) {
    m_context = context;
  }
  private ServiceTracker init(BundleContext context) {
    ServiceTracker tracker = null;
    try {
      tracker = new ServiceTracker(
        context, org.osgi.service.log.LogService.class.getName(), null);
      tracker.open();
    } catch (NoClassDefFoundError error) { }
    return tracker;
  }
  public synchronized void close() {                                       #2
    if (m_tracker != null) {
      m_tracker.close();
    }
  }
  public synchronized void log(int level, String msg) {                    #3
    boolean logged = false;
    if (m_tracker == null) {
      m_tracker = init(m_context);                                         #4
    }
    if (m_tracker != null) {
      LogService logger = (LogService) m_tracker.getService();
      if (logger != null) {
        logger.log(level, msg);
        logged = true;
      }
    }
    if (!logged) {
      System.out.println("[" + level + "] " + msg);
    }
  }
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
}
```

We can no longer make our ServiceTracker member variable final at (#1), since we don't know when it will actually be created. To make our proxy logger thread safe and avoid creating more than one ServiceTracker instance, we need to synchronize our entry methods at (#2) and (#3). Since the logging package could appear at any time during execution, we try to create the ServiceTracker instance each time we log a message at (#4) until successful. As before, it all else fails, we log to standard output. The manifest metadata is pretty much the same as before, except for the logging package:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: example.logger
Bundle-Activator: example.logger.Activator
Import-Package: org.osgi.framework, org.osgi.util.tracker
DynamicImport-Package: org.osgi.service.log
```

These two example should illustrate the differences between these two mechanisms. As you can see, if you really plan to take advantage of the fully dynamic nature of dynamic imported packages, then there is added complexity with respect to threading and concurrency. There is also potential overhead associated with dynamic imports too, not only because of the synchronization, but also because it can be costly for the framework to try to find a matching package at execution time. For logging, which happens frequently, this cost could be great.

Optional imports are optional

We should point out that you can use dynamic imports in a fashion similar to optional imports. So in this sense, the use of the optional import package mechanism is itself optional. For example, we could just modify the metadata of the optional logger example to be a dynamic import instead, but keep the code exactly the same. If we did this, then the two examples would be largely equivalent. If this is the case, then why choose one over the other? There is no real reason or recommendation for doing so. These two concepts overlap for historical reasons. Dynamic imports have existed since the R2 release of the OSGi specification, whereas optional imports have only existed since the R4 release. Even though optional imports overlapped dynamic imports, they were added for consistency with bundle dependencies, which were also added in R4 and could also be declared as optional. We will look at them next.

5.3 Requiring bundles

In section 5.1.2, we discussed how implicit export attributes allow bundles to import packages from a specific bundle. The OSGi specification also supports a module-level dependency concept, called a required bundle, providing a similar capability. In [ref section ch2], we discussed a host of reasons why package-level dependencies are preferred over module-level dependencies, such as them being less brittle and coarse grained. We won't

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

rehash those general issues. However, there is one particular use case where requiring bundles may be necessary in OSGi – if you must deal with split packages.

SPLIT PACKAGE

A split package is a Java package whose classes are not contained in a single JAR, but are split across multiple JAR files; in OSGi terms, it is a package split across multiple bundles.

In standard Java programming, packages are generally treated as split; the Java class path approach merges all packages from different JAR files on the class path into one big soup. This is the anathema to OSGi's modularization model where packages are treated as atomic (i.e., they cannot be split).

When migrating to OSGi from a world where split packages are common, we are often forced to confront ugly situations. But even in the OSGi world, over time a package may grow too large and reach a point where you can logically divide it into disjoint functionality for different clients. Unfortunately, if you simply break up the existing package and assign new disjoint package names, you'd break all existing clients. Splitting the package allows its disjoint functionality to be used independently, but for existing clients we need an aggregated view of the package.

This gives you an idea of what a split package is, but how does this relate to requiring bundles? This will become clearer after we discuss what it actually means to require a bundle and introduce a use case for doing so, which we'll do next.

5.3.1 Declaring bundle dependencies

The big difference between importing a package and requiring a bundle is the scope of the dependency. While an imported package defines a dependency from a bundle to a specific package, a required bundle defines a dependency from a bundle to every package exported by a specific bundle. To require a bundle, we declare it in the bundle's manifest file using the bundle symbolic name and version of the required bundle.

REQUIRE-BUNDLE

A comma-separated list of target bundle symbolic names on which a bundle depends, indicating the need to access all packages exported by the specifically mentioned target bundles.

We use the `Require-Bundle` header to specify a bundle dependency in a manifest, like this:

```
Require-Bundle: A; bundle-version="[1.0.0,2.0.0)"
```

Resolving required bundles is similar to imported packages. The framework tries to satisfy each required bundle; if it is unable to do so, then the bundle cannot be used. The framework resolves the dependency by searching the installed bundles for ones matching the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

specified symbolic name and version range. Figure 5.9 depicts a resolved bundle dependency.

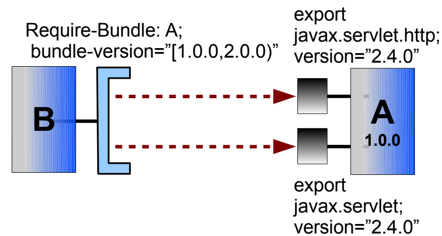


Figure 5.9 Requiring a bundle is similar to explicitly importing every package exported by the target bundle

When resolving a bundle dependency, the framework must still obey any "uses" constraints associated with the exported packages of the required bundle. To a large degree, requiring bundles is just a brittle way to import packages, since it specifies "who" instead of "what". The significant difference is how it fits into the overall class search order for the bundle, which is:

1. Requests for classes in "`java.`" packages are delegated to the parent class loader; searching stops with either a success or failure (from section 2.5.4).
2. Requests for classes in an imported package are delegated to the exporting bundle; searching stops with either a success or failure (from section 2.5.4?).
3. Requests for classes in a package from a required bundle are delegated to the exporting bundle; searching stops if found, but continues with the next required bundle or the next step with a failure.
4. The bundle class path is searched for the class; searching stops if found, but continues to the next step with a failure (from section 2.5.4).
5. Requests for classes matching a dynamically imported package are delegated to an exporting bundle if one is found; searching stops with either a success or failure (from section 5.2.2).

Packages from required bundles are searched only if the class was not found in an imported package, which means imported packages override packages from required bundles. Did you notice another important difference between imported packages and packages from required bundles in the search order? If a class in a package from a required bundle cannot be found, the search continues to the next required bundle in declared order or the bundle's local class path. This is because `Require-Bundle` supports split packages, which we will discuss in more detail in the next subsection, but first let's look at some of the remaining details of requiring bundles.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

As we briefly mentioned in section 5.2.4, it is possible to optionally require a bundle using the `resolution` directive; for example:

```
Require-Bundle: A; bundle-version="[1.0.0,2.0.0)"; resolution:="optional"
```

The meaning is the same as when we optionally import packages, such as not impacting dependency resolution and the need to catch `ClassNotFoundException`s when your bundle attempts to use potentially missing classes. It is also possible to control downstream visibility of packages from a required bundle using the `visibility` directive, which can be specified as either `private` by default or `reexport`. For example:

```
Require-Bundle: A; bundle-version="[1.0.0,2.0.0)"; visibility:="reexport"
```

This makes the required bundle dependency transitive. If a bundle contains this, then any bundle requiring it will also see the packages from bundle A (i.e., they are re-exported). Figure 5.10 provide a pictorial example.

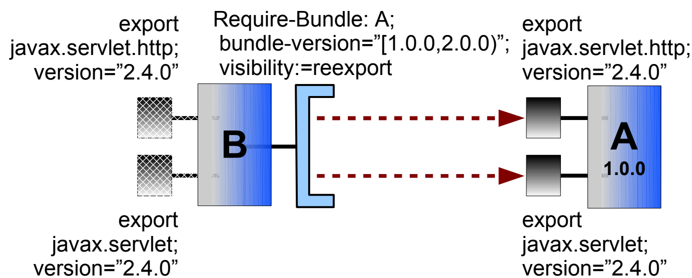


Figure 5.10 When bundle B requires bundle A with `reexport` visibility, any packages exported from A will become visible to any bundles requiring B

CAUTION!

There are very few, if any, good reasons to use `Require-Bundle` with `reexport` visibility. This mechanism is not very modular and excessive use results in very brittle systems with high coupling.

Now let's return our attention to how `Require-Bundle` supports split packages.

5.3.2 Aggregating split packages

Avoiding split packages is the recommended approach in OSGi, but occasionally you may run into a situation where you need to split a package across bundles. `Require-Bundle` makes such situations possible. Since class searching does not stop when a class is not found for required bundles, it is possible to use `Require-Bundle` to search for a class across a split package by requiring multiple bundles containing its different parts. For example, assume we have a package `org.foo.bar` split across bundles A and B. Here is a manifest snippet from bundle A:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: A
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Bundle-Version: 2.0.0
Export-Package: org.foo.bar; version="2.0.0"
```

Here is a manifest snippet from bundle B:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: B
Bundle-Version: 2.0.0
Export-Package: org.foo.bar; version="2.0.0"
```

Both bundles claim to export `org.foo.bar`, even though they each only offer half of it; yes, this is problematic, but we will ignore it for now and come back to it shortly. Now if we have another bundle wanting to use the entire `org.foo.bar` package, it has to require both bundles. The bundle metadata might look something like this:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: C
Bundle-Version: 1.0.0
Require-Bundle: A; version="[2.0.0,2.1.0)", B; version="[2.0.0,2.1.0)"
```

When code from bundle C attempts to load a class from the `org.foo.bar` package it follows these steps:

1. It delegates to bundle A. If the request succeeds the class is returned, but if it fails it goes to the next step.
2. It delegates to bundle B. If the request succeeds the class is returned, but if it fails it goes to the next step.
3. It tries to load the class from bundle C's local class path.

The last step actually allows `org.foo.bar` to be split across the required bundles as well as the requiring bundle. Since searching continues across all require bundles, bundle C is able to use the whole package. What about a bundle wanting to use only one half of the package? Instead of requiring both bundles, it could require just the bundle containing the portion it needs. Sounds reasonable, but does this mean once you split a package you are stuck with using bundle-level dependencies and can no longer use package-level dependencies? No, it doesn't, but it does require some best practice recommendations.

Handling split packages with Import-Package

If another bundle wanted to use `Import-Package` to access the portion of the package contained in bundle B, it could do something like this:

```
Import-Package: org.foo.bar; version="2.0.0"; bundle-symbolic-name="B"
```

This is actually quite similar to using `Require-Bundle` for the specific bundle. If we added an arbitrary attribute to each exported, called `split` for example, we could use it to indicate a part name. Assume we set `split` equal to `part1` for bundle A and `part2` for bundle B. Then we could import the portion from B as follows:

```
Import-Package: org.foo.bar; version="2.0.0"; split="part2"
```

This has the benefit of being a little more flexible, since if we later change which bundle contains which portion of the split package it won't break existing clients. What about existing clients which were using `Import-Package` to access to the entire `org.foo.bar`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

package, is it still possible? It is very likely there are existing client bundles simply doing the following:

```
Import-Package: org.foo.bar; version="2.0.0"
```

Will they see the entire package if it is now split across multiple bundles? No. How would the framework resolve this dependency? The framework has no understanding of split packages as far as dependency resolution is concerned. If bundles A and B were installed and another bundle came along with the above import declaration, the framework would treat A and B as both being candidates to resolve dependency. It would simply choose one following the normal rules of priority for multiple matching candidates. Clearly, no matter which candidate it chose, the resulting solution would be incorrect. To avoid such situations, we need to ensure our split package portions aren't accidentally used by the framework to resolve an import for the entire package, but how? Mandatory attributes can help us here, we could rewrite bundle A's metadata like so:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: A
Bundle-Version: 2.0.0
Export-Package: org.foo.bar; version="2.0.0"; split="part1";
 mandatory:="split"
```

And likewise for bundle B, but with `split` equal to `part2`. Now for a bundle to import either part of the split package, they must explicitly mention the part they wish to use. But what about an existing client bundle wanting to import the whole package? Since its import doesn't specify the mandatory attribute, it cannot be resolved. We need some way to reconstruct the whole package and make it available for importing; OSGi allows us to create a façade bundle for such a purpose. To make bundle C a façade bundle, we change its metadata to be:

```
Bundle-ManifestVersion: 2
Bundle-SymbolicName: C
Bundle-Version: 1.0.0
Require-Bundle: A; version="[2.0.0,2.1.0)", B; version="[2.0.0,2.1.0)"
Export-Package: org.foo.bar; version="2.0.0"
```

The only change is the last line where bundle C exports `org.foo.bar`, which is another form of re-exporting a package. In this case, it aggregates the split package by requiring the bundles containing the different parts and it re-exports the package without the mandatory attribute. Now any client simply importing `org.foo.bar` will be able to resolve to bundle C and have access to the whole package. This outlines best practices when it comes to split packages:

1. Always export split packages with a mandatory attribute to avoid unsuspecting bundles from using it.
2. Use either `Require-Bundle` or `Import-Package` plus the mandatory attribute to access split packages.
3. To provide access to the whole package, create a façade bundle that requires the bundles containing the package parts and exports the package in question.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

This is not necessarily the most intuitive or straightforward way to deal with split packages, but they are anathema to OSGi. This approach was not intended to make them easy to use, since they are best avoided, but it does make it possible for those situations where you have no choice. Despite these dire sounding warnings, OSGi actually provides another way of dealing with split packages, called bundle fragments. We will talk about those shortly, but first we will discuss some of the issues surrounding bundle dependencies and split packages.

5.3.3 Issues with bundle dependencies

Using `Import-Package` and `Export-Package` is the preferred way to share code since they couple the importer and exporter to a lesser degree. Using `Require-Bundle` entails much tighter coupling between the importer and the exporter and suffers from other issues, such as the following.

Mutable exports

Requiring bundles are impacted by changes to the exports of the required bundle, which introduce another form of breaking change needing to be considered. Such changes are not always easily apparent since the use of `reexport` visibility can result in chains of required bundles where removal of an export in upstream required bundles breaks all downstream requiring bundles.

Shadowing

Since class searching continues across all required bundles and the bundle class path like normal class path searching, it is possible for content in required bundles to shadow other required bundles and the bundle itself. The implications of this are not always obvious, especially if some bundles are optionally required.

Ordering

If a package is split across multiple bundles, but there are overlapping classes in them, then the declared ordering of `Require-Bundle` declaration is significant. All bundles requiring the bundles with overlapping content must declare them in the same order or else their view of the package will be different. This is similar to traditional class path ordering issues.

Completeness

Even though it is possible to aggregate split packages using a façade bundle, the framework has no way of verifying whether an aggregated package is complete. This becomes the responsibility of the bundle developer.

Restricted access

An aggregated split package is not completely equivalent to the unsplit package. Each portion of the split package is loaded by its containing bundle's class loader. In Java, classes loaded by different class loaders cannot access package private members and types, even if they are in the same package.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

This is by no means an exhaustive list, but it gives you some ideas of what to look out for when using `Require-Bundle` and hopefully dissuades you from using it too much. Enough of the scare tactics, so far we have introduced you to some of the more advanced features of managing OSGi dependencies, including:

- Importing exports
- Implicit export attributes
- Mandatory export attributes
- Export filtering
- Duplicate exports
- Optional and dynamic imports
- Requiring bundles

These tools allow you to solve some of the more complex edge cases found when migrating a classic Java application to an OSGi environment. That must be it right? We must have covered every possible mechanism of specifying dependencies? Well not quite, there is one more curve ball to be thrown into the mix, that of bundle fragments. Fragments are another way to deal with split packages by allowing the content of a bundle to be split across multiple bundle JAR files. We will learn about these and the patterns and that arise from them in the next section.

5.4 Dividing bundles into fragments

Although splitting packages is not a good idea, occasionally it does make sense, such as with Java localization. Java handles localization by using `java.util.ResourceBundle`s (which have nothing to do with OSGi bundles), as a container to help you turn locale-neutral keys into local-specific objects. When a program wants to convert information into the user's preferred locale, it uses a resource bundle to do so. A `ResourceBundle` is created by loading a class or resource from a class loader using a base name, which ultimately defines the package containing the class or resource for the `ResourceBundle`. This approach means you typically package many localization for different locales into the same Java package.

If you have lots of localizations or lots of information to localize, then packaging all of your localizations into the same OSGi bundle could result in a large deployment unit. Additionally, you wouldn't be able to introduce new localizations or fix mistakes in existing ones without releasing a new version of the bundle. It would be nice if we could keep localizations separate, but unlike the split package support of `Require-Bundle`, these split packages are generally not useful without the bundle to which they belong. To this end OSGi, provides another approach to managing these sorts of dependencies through fragments. We

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

will come back to localization shortly when we present a more in-depth example, but first we will discuss what fragments are and what we can do with them.

5.4.1 Understanding fragments

If we recall the modularity discussion of Chapter 2, we know there is a difference between logical modularity and physical modularity. Normally in OSGi, a logical module and a physical module are treated as the same thing; a bundle is a physical module as a JAR file, but it is also the logical module at run-time forming an explicit encapsulation boundary. Through fragments, OSGi allows us to break a single logical module across multiple physical modules. This means we can split a single logical bundle across multiple JAR files.

Breaking a bundle into pieces does not result in a handful of peer bundles; instead, we define one host bundle and one or more fragment bundles. A host bundle is technically usable without fragments, but the fragments are not usable without a host. Another way to think about it is fragments are treated as optional by the OSGi framework. The host bundle is not necessarily aware of its fragments, since it is the fragments who declare a dependency on the host using the `Fragment-Host` manifest header, like this:

```
Fragment-Host: org.foo.hostbundle; bundle-version="[1.0.0,1.1.0)"
```

The `Fragment-Host` header is somewhat confusingly named, since it seems to be declaring the bundle as a fragment host; it should be read as "require fragment host". The target is the symbolic name and bundle version range of the host bundle. Although this header value follows the common OSGi syntax, you cannot specify multiple symbolic names. A fragment is limited to belonging to one host bundle, although it may be applicable to a range of host versions. Note that we do not need to do any special to define a bundle as a host, any bundle without a `Fragment-Host` header is a potential host bundle. Likewise, any bundle with a `Fragment-Host` header is a fragment. So, we understand the relationship between a host and its fragments, but how do they actually work together?

When the framework resolves a bundle, it searches the installed bundles to see if there are any fragments for it. If so, it merges the fragments into the host bundle. This merging happens in two different ways:

1. Physical – the content and metadata from fragments conceptually become part of the host bundle.
2. Logical – unlike normal bundles, fragments do not have their own class loader and instead are accessed via their host's class loader.

The first form of merging recombines the split physical pieces of the logical bundle, while the second form creates a single logical bundle since OSGi uses a single class loader per logical bundle to achieve encapsulation.

Fragments and package private access

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

As a technical side note, Java only allows package private access to classes loaded by the same class loader; two classes in the same package, but loaded by two different class loaders cannot access each others' package private members. By using the host class loader, fragment bundles are properly recombined to avoid this issue in the case where a package is split across fragments and/or the host. This is not the case for split packages accessed through `Require-Bundle`. This is not always important, but the distinction between these two forms of support for split packages is worth understanding.

Returning to our discussion about resolving a bundle, if it has fragments then the framework merges their metadata with the host's and resolves the bundle like normal. Figure 5.11 illustrates the before and after effects of the merging process.

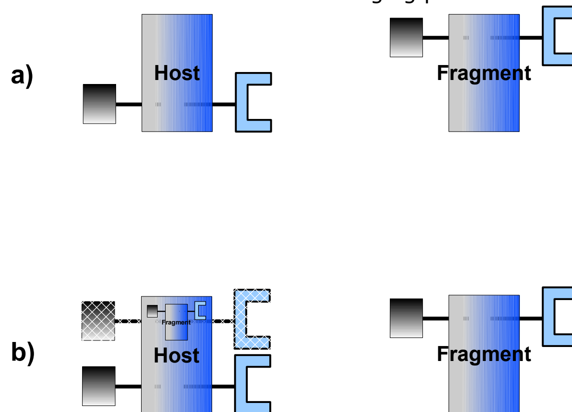


Figure 5.11 a) The host and fragment bundles are deployed as independent bundles in the framework; b) when the framework resolves the host, it effectively merges the fragment's content and metadata into the host bundle

1. Requests for classes in "`java.`" packages are delegated to the parent class loader; searching stops (section 2.5.4?).
2. Requests for classes in an imported package are delegated to the exporting bundle; searching stops (section 2.5.4?).
3. Requests for classes in a package from a required bundle are delegated to the exporting bundle; searching continues with a failure (section 5.3.1).
4. The host bundle class path is searched for the class; searching continues with a failure (section 2.5.4?).
5. Fragment bundle class paths are searched for the class in the order in which the fragments were installed; searching stops if found, but continues through all fragment class paths and then to the next step with a failure.
6. Requests for classes matching a dynamically imported package are delegated to an exporting bundle if one is found; searching stops (section 5.2.2).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

This is the complete bundle class search order, as such you may want to mark this page for future reference! This search order makes it clear how fragments support split packages, since the host and all fragment class paths are searched until the class is found.

Fragments and the `Bundle-ClassPath`

Bundle class path search order seems fairly linear, but fragments do introduce one exception. When specifying a bundle class path, we can embed JAR files, such as:

```
Bundle-ClassPath: .,specialized.jar,default.jar
```

Typically, you would expect both of these JAR files to be contained in the JAR file of the bundle declaring it, but this need not be the case. If fragments aren't involved, then the framework ignores a non-existent JAR file on the bundle class path. However, if the bundle has fragments attached to it, then the framework will search the fragments for the specified JAR files if they do not exist in the host bundle. In the above example, imagine the host contains `default.jar`, but does not contain `specialized.jar`, which is actually contained in an attached fragment. The impact this has on the class search order is that the specified fragment content will be searched before some of the host bundle content.

Sometimes this is useful if you want to provide some default functionality in the host bundle, but be able to override it on platforms where you have specialized classes (e.g., using native code). This approach can also be used to provide a means for issuing patches to bundles after the fact, but in general it is better to replace the whole bundle.

Some final issues regarding fragments. Fragments are allowed to have any metadata a normal bundle may have, except `Bundle-Activator`. This makes sense since fragments cannot be started or stopped and can only be used when combined with the host bundle. Attaching a fragment to a host creates a dependency between the two, which is similar to the dependencies created between two bundles via Import-Package or Require-Bundle. This means if either bundle is updated or uninstalled, then the other bundle is impacted and any refreshing of the one will likely lead to refreshing of the other.

We started our foray into fragments discussing localization since it was the main use case for them. Next, we will look at an example of how to use fragments for just that purpose.

5.4.2 Using fragments for localization

To see how we can use fragments to localize an application, let's return to the service-based paint program from chapter 4. The main application window was implemented by the `PaintFrame` class. Recall its design, `PaintFrame` did not have any direct dependencies on OSGi API; instead, it used a `ShapeTracker` class to track `SimpleShape` services in the OSGi service registry and inject them into the `PaintFrame` instance. `ShapeTracker`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

injected service into the PaintFrame using its addShape() method as depicted in Listing 5.4.

Listing 5.4 Method used to inject shapes into PaintFrame object

```
public void addShape(String name, Icon icon, SimpleShape shape) {        #1
  m_shapes.put(name, new ShapeInfo(name, icon, shape));                  #2
  JButton button = new JButton(icon);                                    #3
  button.setActionCommand(name);
  button.setToolTipText(name);                                           #4
  button.addActionListener(m_reusableActionListener);

  if (m_selected == null) {
    button.doClick();
  }

  m_toolbar.add(button);                                                 #5
  m_toolbar.validate();
  repaint();
}
```

We see that addShape() is invoked with the name, icon, and service object of the SimpleShape implementation at (#1). The exact details are not so import, but the shape is recorded in a data structure at (#2), a button is created for it at (#3), its name is set as the button's tool tip at (#4), and after a few other steps the associated button is added to the toolbar at (#5). Focusing on (#4), the tool tip is textual information displayed to the user when he or she hovers the mouse over the shape's toolbar icon. It would be nice if this information could be localized.

There are different approaches we could take to localize the shape name. One approach is to define a new service property which defines the ResourceBundle base name. This way shape implementations could define their localization base name, much like they use service properties to indicate name and icon. In such an approach, the PaintFrame.addShape() would need to be injected with the base name property so it could perform the localization lookup. This is probably not ideal in this situation. Another approach is to focus on where the shape's name is set in the first place, which occurs in the shape implementation's bundle activator. Listing 5.5 depicts the activator of the circle implementation.

Listing 5.5 Original circle bundle activator with hard-coded name

```
public class Activator implements BundleActivator {
  public void start(BundleContext context) {
    Hashtable dict = new Hashtable();
    dict.put(SimpleShape.NAME_PROPERTY, "Circle");                       #1
    dict.put(SimpleShape.ICON_PROPERTY,
      new ImageIcon(this.getClass().getResource("circle.png")));
    context.registerService(
      SimpleShape.class.getName(), new Circle(), dict);                  #2
  }
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
  public void stop(BundleContext context) {}
}
```

We can see at (#1) the hard-coded shape name is assigned to the service property dictionary and the shape service is ultimately registered at (#2). So the first thing we need to do is to change the hard-coded name into a lookup from a `ResourceBundle`. Listing shows the necessary changes.

Listing 5.6 Modified circle bundle activator with `ResourceBundle` name lookup

```
public class Activator implements BundleActivator {
  public static final String CIRCLE_NAME = "CIRCLE_NAME";                    #1

  public void start(BundleContext context) {
    ResourceBundle rb = ResourceBundle.getBundle(
      "org.foo.shape.circle.resource.Localize");                             #2
    Hashtable dict = new Hashtable();
    dict.put(SimpleShape.NAME_PROPERTY, rb.getString(CIRCLE_NAME));          #3
    dict.put(SimpleShape.ICON_PROPERTY,
      new ImageIcon(this.getClass().getResource("circle.png")));
    context.registerService(
      SimpleShape.class.getName(), new Circle(), dict);
  }

  public void stop(BundleContext context) {}
}
```

We've modified the activator to use look up the shape name using the key constant defined at (#1) from a `ResourceBundle` we create at (#2), which is assigned to the service properties at (#3). Even though we won't go into the complete details of using `ResourceBundle` objects, the important part in this example is when we define it at (#2). Here we specify the base name of `org.foo.shape.circle.resource.Localize`. By default, this refers to the `Localize.properties` in the `org.foo.shape.circle.resource` package, which contains a default mapping for our name key. We need to modify our circle implementation to have this additional package and we add the `Localize.properties` file to it with the following content:

```
CIRCLE_NAME=Circle
```

This is the default mapping for our shape name; if we had a more complicated example, we would have many more default mappings for other terms. If we want to provide other mappings to other languages, we need to include them in this same package, but in separate property files named after the locale's country code. For example, the country code for Germany is “DE”, so for its localization we create a file called Localize_de.properties with the following content:

```
CIRCLE_NAME=Kreis
```

We do this for each locale we want to support, then at runtime when we create our `ResourceBundle`, the correct property file will be selected based on locale of the user's computer. This all sounds pretty nice, but if we have a lot of information to localize, then we need to include all of this information in our bundle, which could greatly increase its size.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Further, we have no way of adding support for new locales without releasing a new version of our bundle. This is where fragments can help, since we can split the resource package into different fragments. We keep the default localization in our circle implementation, but all other localizations are put into separate fragments. We didn't need to change the metadata of our circle bundle, since it is unaware of fragments; the contents of our circle bundle is:

```
META-INF/MANIFEST.MF
META-INF/
org/
org/foo/
org/foo/shape/
org/foo/shape/circle/
org/foo/shape/circle/Activator.class
org/foo/shape/circle/Circle.class
org/foo/shape/circle/circle.png
org/foo/shape/circle/resource/
org/foo/shape/circle/resource/Localize.properties
```

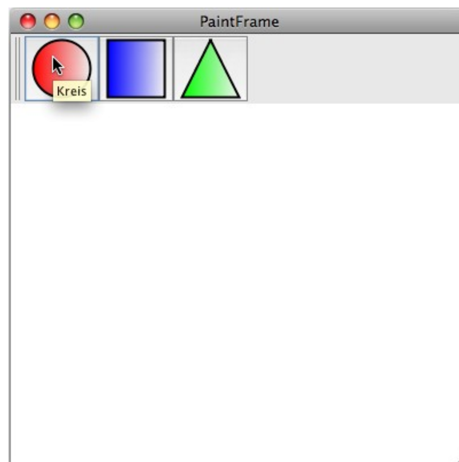
For this example we will create a localization fragment bundle for German using the properties file shown above; the metadata for this bundle is:

```
Bundle-ManifestVersion: 2
Bundle-Name: circle.resource-de
Bundle-SymbolicName: org.foo.shape.circle.resource-de
Bundle-Version: 5.0
Fragment-Host: org.foo.shape.circle; bundle-version="[5.0,6.0)"
```

The important part of this metadata is the last line, which declares it as a fragment of the circle bundle. The contents of the fragment bundle is quite simple:

```
META-INF/MANIFEST.MF
META-INF/
org/
org/foo/
org/foo/shape/
org/foo/shape/circle/
org/foo/shape/circle/resource/
org/foo/shape/circle/resource/Localize_de.properties
```

It only contains a resource file for the German translation, which we can see is a split package with the host bundle. We can create any number of localization fragments following this same pattern and for our other shapes (e.g., square and triangle). Figure 5.12 shows the paint program with the German localization fragments installed.



©Manning Publications ... the Author Online forum:
http:// ... mID=507

Figure 5.12 Paint program with installed German localization fragments

To run this example, go into the `code/chapter05/paint-example/` directory of the companion code and type `ant` to build and `java -jar launcher.jar bundles/` to run. With this approach, we only need to deploy the needed localization fragments along with our shape implementations and we can create new localizations or update existing ones without releasing new versions of our shape bundles.

We've now covered all aspects of the OSGi modularity specification! As you can see there is a comprehensive toolkit available to help you deal with virtually any scenario the Java language can throw at you. But there is one more trick up our sleeves, the OSGi specification actually does a pretty good job at dealing with native code that runs outside of the Java environment. We'll look at this and how to deal with general factors relating to the JVM environment in the next section.

5.5 Dealing with your environment

Although Java has been fairly successful at attaining its original vision of "write once, run everywhere," there are still situations where it is not entirely able to achieve this vision. One such situation is the myriad of Java platforms, such as Java ME as well as different versions of Java SE. If you develop a bundle with requirements for a specific Java platform, for example if you use classes from the `java.util.concurrent` package, then you will need a Java 5.0 JVM or above. Another situation is if you need to natively integrate with the underlying operating system, such as might be necessary if you need to communicate directly with underlying hardware. As you might expect, in both these situations the OSGi specification provides mechanisms to explicitly declare these scenarios in your bundles to allow an OSGi framework to do the necessary work at execution time. In this section, we will cover both of these topics, starting with the former.

5.5.1 Requiring execution environments

If you develop a bundle with dependency on specific Java execution environments, what will happen if this bundle executes in an unintended environment? Most likely you will get various exceptions for missing classes/methods and/or just faulty results. The "write once, run everywhere" approach for creating bundles is certainly the best, if you can achieve it. But when you can't, you really need to declare the bundle's execution environment constraints in the bundle metadata to avoid people unknowingly deploying your bundle to invalid environments.

The OSGi specification defines an execution environment concept for just this purpose. Like all bundle metadata, we use a manifest header to define it; in this case the header has the rather long name of `Bundle-RequireExecutionEnvironment`. The value of the header is a comma-delimited list of execution environments. Table 5.1 lists the execution environments names defined by the OSGi specification.

Table 5.1 OSGi defined standard execution environment names

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Name	Description
CDC-1.1/Foundation-1.1	Equivalent to J2ME Foundation Profile
CDC-1.1/PersonalBasis-1.1	J2ME Personal Basis Profile
CDC-1.1/PersonalJava-1.1	J2ME Personal Java Profile
J2SE-1.2	Java 2 SE 1.2.x
J2SE-1.3	Java 2 SE 1.3.x
J2SE-1.4	Java 2 SE 1.4.x
J2SE-1.5	Java 2 SE 1.5.x
JavaSE-1.6	Java SE 1.6.x
OSGi/Minimum-1.2	Minimal required set of Java API that allows an OSGi framework implementation

Bundles should list all known execution environments on which they can run, which might look something like this:

```
Bundle-RequiredExecutionEnvironment: J2SE-1.4,J2SE-1.5,JavaSE-1.6
```

This indicates the bundle only runs on modern Java platforms. If a bundle lists a limited execution environment, such as `CDC-1.1/Foundation-1.1`, then it should not use any classes or methods that do not exist in the execution environment. The framework does not verify this claim, it only ensures the bundle is not resolvable on incompatible execution environments.

Resolve-time, not install-time enforcement

Pay special attention to the last sentence. It is possible to install bundles on a given execution environment even if they are not compatible with it, but you will not be able to resolve a bundle unless its execution environment matches the current one. The reason this is tied to the bundle's resolved state is it possible the execution environment changes over time; for example, you may switch between different versions of Java on subsequent executions of the framework. This way, any cached bundles not matching the current execution environment will essentially just be ignored.

A given framework implementation can claim to provide more than one execution environment, since it most case the Java platform versions are backwards compatible. It is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

possible to determine the framework's supported execution environments by retrieving the `org.osgi.framework.executionenvironment` property from `BundleContext.getProperty()`. Now that we understand how to declare our bundles' required execution environments, let's look at how to handle bundles with native code.

5.5.2 Bundling native libraries

Java provides a nice platform and language combination, but it is not always possible to stay purely in Java. In some situations you need to create native code for the platform on which Java is running, such as when communicating directly with hardware or for special performance considerations. Java defined JNI (Java Native Interface) precisely for this purpose; JNI is how Java code calls code written in other programming languages (e.g., C or C++) for specific operating systems (e.g., Windows or Linux). A complete discussion of how JNI works is outside the scope of this book, but the following bullet points provide the highlights:

- Native code is integrated into Java as a special type of method implementation; a Java class can declare a method as having native implementation using the `native` method modifier.
- Classes with native code are compiled normally, but after compilation the `javah` command is used generate C header and stub files, which are used to create the native method implementations.
- The native code is compiled into a library in a platform-specific way for its target operating system.
- The original Java class with the native method will include code to invoke `System.loadLibrary()`, typically in a static initializer, to load the above native library when the class is loaded in the Java VM.
- Other classes can simply invoke the native method as if it were a normal method and the Java platform takes care of the native invocation details.

Although it is fairly straightforward to use native code in Java, it is best to avoid it if possible. Native code does not benefit from the garbage collector and suffers from the typical pointer issues associated with all native code. Additionally, it hinders Java's “write one, run everywhere” goal, since it ties the class to a specific platform. Still, in those cases where it is absolutely necessary, it is nice to know that OSGi supports it. Actually, OSGi even simplifies it a little.

One of the downsides of native code is you end up with an additional artifact to deploy along with your classes. To make matters worse, what you need to do with the native library differs among operating systems; for example, you typically need to put native libraries in specific locations in the file system so they can be found at execution time (e.g., somewhere on the binary search path). OSGi native code support simplifies these issues by:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

- Allowing you to embed your native library directly into your bundle JAR file.
- Allowing you to embed multiple native libraries for different target platforms.
- Automatically handling execution time discovery of native code libraries.

When you embed a native library into your bundle, you must tell the OSGi framework about it. As with all other modularity aspects, we do so in the bundle metadata using the `Bundle-NativeCode` manifest header. With this header we can specify the set of contained native libraries for each platform our bundle supports; the grammar is as follows:

```
Bundle-NativeCode ::= nativecode (',' nativecode)* (',' optional)?
nativecode        ::= path (';' path)* (';' parameter)+
optional          ::= '*'
```

Where a parameter is one of the following:

- `osname` – Name of the operating system.
- `osversion` – The operating system version range.
- `processor` – The processor architecture.
- `language` – The ISO code for a language.
- `selection-filter` – An LDAP selection filter.

For example, if we have a bundle with native libraries for Windows XP, then we might have a native code declaration like:

```
Bundle-NativeCode: lib/math.dll; lib/md5.dll; osname=WindowsXP;
 processor=x86
```

This is a semicolon-delimited list, where the first entries not containing an '=' character are interpreted as file entries in the bundle JAR file and the remaining entries with an '=' character are used to determine if the native library clause matches the current platform. In this case, we state the bundle has two native libraries for Windows XP on the x86 architecture. If `Bundle-NativeCode` is specified, then there must be a matching header for the platform on which the bundle is executing or else the framework will not allow the bundle to resolve. In other words, if a bundle with the above native code header was installed on a Linux box, the framework wouldn't allow the bundle to be used. Makes sense. If these same libraries also worked on Vista, we could specify this as follows:

```
Bundle-NativeCode: lib/math.dll; lib/md5.dll; osname=WindowsXP;
 osname=WindowsVista; processor=x86
```

In cases where the parameter is repeated, then the framework treats this like a logical OR when matching, so these native libraries match Windows XP or Windows Vista. If our bundle also had native libraries for Linux, we could specify this as follows:

```
Bundle-NativeCode: lib/math.dll; lib/md5.dll; osname=WindowsXP;
 osname=WindowsVista; processor=x86, lib/libmath.so; osname=Linux;
 osprocessor=x86
```

We separate different platforms using a comma instead of a semicolon. Notice also that the native libraries do not need to be parallel. In this example, we have two native libraries for the Windows platform, but only one for Linux. This bundle would now be usable on

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Windows XP, Windows Vista, and Linux on the x86 architecture, but on any other platform the framework would not resolve it. In some cases, we may either have optional native libraries or a non-optimized Java implementation for unsupported platforms. We can denote this using the optional clause, like this:

```
Bundle-NativeCode: lib/math.dll; lib/md5.dll; osname=WindowsXP;
 osname=WindowsVista; processor=x86, lib/libmath.so; osname=Linux;
 osprocessor=x86, *
```

The `*` at the end acts as a separate platform clause that can match any platform, so this bundle would be usable on any platform. The actual process of how native libraries are made available when the classes containing native methods perform `System.loadLibrary()` is handled automatically by the framework, so we don't need to worry about it. Even though it isn't often necessary, it is fairly easy to use this mechanism to create bundles with native code.

So now we know how to package our bundles using all the tools of the OSGi modularity layer and also how to deal with issues related to execution environment and native code. The final topic we are going to look at in this chapter is dealing with a particular use case related to bundle usage of physical resources at execution time. In truth, this topic is really related to lifecycle, but we include it in this modularity chapter since it is connected to class loading, which is a modularity concern. Read on and all will be explained.

5.6 Starting lazy bundles

From chapter 3, we know starting a bundle involves invoking the `Bundle.start()` method on its corresponding `Bundle` object. If the bundle has a BundleActivator and it is resolvable, this causes the framework to create an instance of the bundle's activator and invoke `start()` on it, allowing the bundle to initialize itself. The act of starting a bundle and of it being activated are actually two independent concepts, although typically they occur together. We will also look at this independent nature in chapter 7 when discussing start levels; a bundle can be started, but it won't be activated until some other event has occurred to tip the bundle into the active state. Sometimes you might want to leverage this independent nature when developing a bundle. Why? There are two main reasons:

1. Your bundle's exported packages are not able to function properly without a `BundleContext` (e.g., perhaps they require a service from the registry).
2. Your bundle's initialization is costly and you want to defer it until it is actually needed.

The OSGi specification allows bundles to declare a lazy activation policy for such purposes. Of course, there are alternative approaches for dealing with these situations. For the first issue, your classes could always throw exception until activated. For the second, you could minimize initialization in your activator and use threads to do work in the background. Sometimes these alternative approaches are feasible, but sometimes throwing exceptions isn't so clean, nor is it possible to completely reduce all startup overhead, especially if you are starting lots of bundles. In these cases, we can use the lazy activation policy.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

5.6.1 Understanding activation policies

Although the OSGi specification defines activation policies in an open-ended way, there is currently only one activation policy, which is lazy. The main gist of the lazy activation policy is this:

1. A bundle declares itself to be lazy.
2. When a lazy bundle is started, the framework marks the bundle as started, but defers activating it.
3. The lazy bundle's activation is deferred until a class is loaded from it.
4. Once a class is loaded from a lazy bundle, the framework completes its activation like normal.

This is fairly straightforward, but there are some small details lurking in there. Let's revisit the bundle life cycle diagram in Figure 5.13 to get a better understanding of the impact.

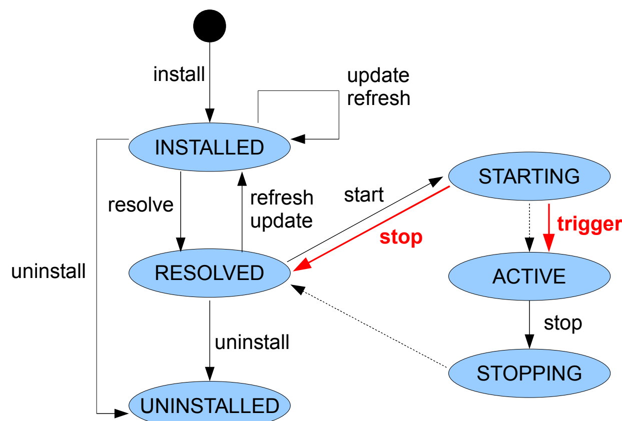


Figure 5.13 The lazy activation policy causes bundles to defer activation and linger in the STARTING state until a class is loaded from it, at which point the framework completes its activation

The red arrows in Figure 5.13 depict additional transitions in the bundle life cycle state diagram. When a bundle is started lazily, it transitions to the `STARTING` state, which is denoted by the framework by firing a `BundleEvent` of type `LAZY_ACTIVTION`, instead of the normal `STARTING` event. The bundle stays in this state until it is stopped or a class is loaded from it. Stopping a lazy bundle in the `STARTING` state returns it to the `RESOLVED` state and results in `STOPPING` and `STOPPED` bundle events. When a class is loaded from a lazy bundle in the STARTING state, this acts as a trigger for the framework automatically activate the bundle, which completes the normal process of creating the bundle activator and calling its `start()` method, resulting in the normal `STARTING` and `STARTED` bundle events.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Since loading a class from one lazy bundle may require other classes to be loaded from other lazy bundles, the framework may end up activating chains of lazy bundles. The framework does not the bundles as it loads classes from them, since this can lead to arcane class loading errors. Instead, the framework delays the activation of each lazy bundle it discovers in a class loading chain until it finishes loading the instigating class. At that point, the framework will activate the detected lazy bundles in reverse order. For example, assume `ClassA` is loaded from bundle A, which requires `ClassB` from bundle B, which in turn requires `ClassC` from bundle C. If all bundles are lazy and in the `STARTING` state, then the framework will activate bundle C, B, and then A before returning `ClassA` to the requester.

Attention!

Be aware that loading resources from a bundle does not trigger lazy activation, only classes. Also, the specification does not unambiguously define how the framework should treat the class loading trigger, so the precise behavior may vary. In particular, some frameworks may scope the trigger to the lifetime of the bundle's class loader (i.e., it only needs to be re-triggered if the bundle is refreshed) while others may scope the trigger to the bundle's `ACTIVE` lifecycle state (i.e., it needs to be re-triggered after the bundle is stopped and restarted). Finally, the lazy activation policy still must obey start levels. This means if the lazy bundle's start level is not satisfied, then it will not be activated even if a class is loaded from the bundle.

Now we know how the lazy activation policy works, let's look into the details of using it.

5.6.2 Using activation policies

As you might expect, we use manifest metadata to declare the activation policy for a bundle; we use the `Bundle-ActivationPolicy` header for this. Since lazy activation is currently the only supported policy, this head can only have one value, which is `lazy`. In our manifest, it would look like this:

```
Bundle-ActivationPolicy: lazy
```

It is only possible for a bundle to declare itself as lazy in its manifest. It is not possible, however, to lazily activate some arbitrary bundle; only bundles declaring the `lazy` policy can be lazily activated. This may seem a little odd, but the thinking behind this goes back to one of the main use cases motivating deferred activation, which is a bundle that requires a `BundleContext` for its exported packages to function properly. In this use case, only the bundle itself knows if this is the case; thus, only the bundle itself can declare the policy.

We know this may sound a little odd, since deferring activation sounds like a good thing to do all the time. Why pay the cost of actually activating a bundle before it is needed? We could start all bundles lazily and then they would only activate when another bundle actually used them, right? There is a fly in the ointment with this approach. Suppose your bundle provides a service. If your bundle is not actually activated, then it won't ever get a chance to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

publish its service in its bundle activator. Thus, no other bundles will be able to use it and it will never get activated lazily. So, even if it were possible to apply an activation policy to a bundle externally, we wouldn't always end up working the way we intended.

All control is not lost, though. When we start a bundle, we at least get to determine whether or not the bundle uses its declared policy. In chapter, we learned about using `Bundle.start()` to start and eagerly activate a bundle. If we call `Bundle.start()` with no arguments on a bundle with lazy activation policy, it will still be activated eagerly, like normal. To start the bundle with lazy activation, we must call `Bundle.start()` with the `Bundle.START_ACTIVATION_POLICY` flag. When we use this flag, we are saying we want to start the bundle using its declared activation policy. A bundle with no activation policy, will be started eagerly like normal, while one with the lazy policy will have its activation deferred as described in the previous section. There is no requirement to start lazy bundles lazily; eagerly starting a bundle is always acceptable, it just means you are willing to pay for the start up cost immediately.

One final lazy activation detail, the specification offers us fine-grained control over which precise packages trigger lazy activation. The specification defines `include` and `exclude` directives, which declare a comma-separated list of included and excluded packages, respectively. For example:

```
Bundle-ActivationPolicy: lazy; include:="com.acme.service"
```

A bundle with this metadata will only be lazily activated if a class is loaded from its `com.acme.service` package. Now let's move on to how bundles can deal with their underlying execution environments.

5.7 Summary

We learned that the OSGi module layer provides many additional mechanisms to deal with collaborative, dynamic, and legacy situations, such as:

- A bundle may import a package it exports to make its export substitutable for purposes of broadening collaboration among bundles.
- Exported packages have bundle symbolic name and version attributes implicitly attached to them, which can be useful if you need to import a package from a specific bundle.
- Exported packages may have mandatory attributes associated with them, which must be specified by an importer to get wired to the exported package.
- It is possible to export the same package more than once with different attributes, which is sometimes helpful if a bundle wishes to masquerade as more than one version of a package.
- The framework ignores optionally imported packages if no exporters are present when the importing bundle is resolved.
- The framework only attempts to resolve dynamically imported packages when the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

importing bundle actually tries to use a class in the imported package; repeated attempts to use a class in the imported package will result in repeated attempts to resolve the package until successful.

- It is possible to require a bundle, rather than importing specific packages, which wires you to everything the target bundle exports; this is typically useful when aggregating split packages.
- Bundle fragments support splitting a bundle into multiple optional JAR files, which is helpful in such situations as localization.
- Bundles with dependencies on specific Java platforms can declare these dependencies with required execution environments.
- Bundles can include native libraries to integrate platform-specific functionality.
- The lazy activation policy defers bundle activation until a class is actually loaded from the lazily started bundle.

Most of these mechanisms are intended for specific use cases and should not be overused to avoid less modular solutions. We've now finished with the first part of the book describing the details of the OSGi framework layers! From here, we will move onto higher level topics describing how we put all of these features to good use.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

6

Moving Towards Bundles

The first part of this book introduced the three layers of OSGi: module, lifecycle, and service. We will now take a more practical look at how you can migrate existing code to OSGi by using one or more of these layers, beginning with examples of turning real-world JAR files into bundles. After that we will examine different ways of migrating a complete application to OSGi and finish up with a short discussion of situations where you might decide not to bundle.

By the end of this chapter you will know how to take your current application and all of its third-party libraries and turn them into bundles, step-by-step. You'll be able to move existing projects to OSGi, plan new projects with OSGi in mind, and understand when it might not be the right solution for you. In other words, you should be able to explain in detail how OSGi will affect your project to your manager and co-workers. But before we reach that stage, we first need to consider a simple question that often comes up on the OSGi mailing lists: how can I turn my JAR file into a bundle?

6.1 Turning JARs into bundles

As we saw in chapter 2, a bundle is a JAR file with additional metadata. So to turn a JAR file into a bundle we just need to add metadata giving it a unique identity and describing what it imports and exports. Simple, right? For most business domain JAR files it will be, but for others (such as third-party GUI or database libraries) you'll need to think carefully about their design. Where is the line between what's public and what's private, which imports are required and which are optional, and what versions are compatible with one another? In this section we'll help you come up with this metadata by taking a series of common library JAR files and turning them into working bundles. We shall also consider some advanced bundling

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

techniques, such as embedding dependencies inside the bundle, as well as how to manage external resources and background threads.

Before we can even load a bundle into an OSGi framework it must have an identity. This identity should be unique, at least among the set of bundles loaded into the framework. But how should we choose such an identity? If we pick names at random we could clash with other projects or other developers, just like in Figure 6.1:

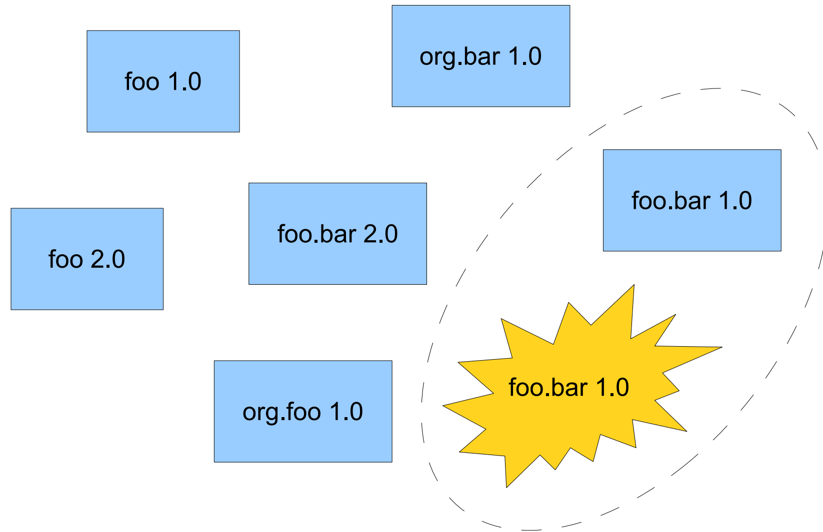


Figure 6.1 Bundles must have a unique identity

6.1.1 Choosing an identity

Each bundle installed into an OSGi framework must have a unique identity, made up of the `Bundle-SymbolicName` and `Bundle-Version`. One of the first steps in turning a JAR file into a bundle is to decide what symbolic name to give it. The OSGi specification doesn't mandate a naming policy, but recommends a reverse domain naming convention. This is the same as Java package naming, so if the bundle is the primary source of a particular package then it makes sense to use it as the `Bundle-SymbolicName`.

Let's look at a real-world example, the kXML parser [ref]. This small JAR file provides two distinct “top-level” packages: the XmlPull API `org.xmlpull.v1` and the kXML implementation `org.kxml2`. If this JAR was the only one expected to provide the `org.xmlpull.v1` API or it only contained this package, then it would be reasonable to use this as the symbolic name. However, this JAR file also provides a particular implementation of the XmlPull API, so it makes more sense to use the name of the implementation as the symbolic name because it captures the essence of what the bundle provides:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Bundle-SymbolicName: org.kxml2
```

Alternatively, you could use the domain of the project that distributes the JAR file:

```
Bundle-SymbolicName: net.sourceforge.kxml.kxml2                         #A
```
#A domain = http://kxml.sourceforge.net/kxml2/

Or if Maven [ref] project metadata is available, you could use this to identify the JAR file:

```
Bundle-SymbolicName: net.sf.kxml.kxml2                                  #A
```
#A Maven groupId + artifactId

Sometimes you might decide on a name that doesn't correspond to a particular package or distribution. For example, consider two implementations of the same service API provided by two different bundles. OSGi lets you hide non-exported packages so these bundles could have an identical package layout, but at the same time provide different implementations. You can still base the symbolic name on the main top-level package, or the distribution domain, but you must add a suffix to ensure each implementation has a unique identity. This is the approach that the SLF4J project [ref] used when naming their various logging implementation bundles:

```
Bundle-SymbolicName: slf4j.juli                                         #A
Bundle-SymbolicName: slf4j.log4j
Bundle-SymbolicName: slf4j.jcl
```
#A all these bundles export org.slf4j.impl

If you are wrapping a third-party library, you might want to prefix your own domain in front of the symbolic name. This makes it clear you are responsible for the bundle metadata rather than the original third-party. For example, the symbolic name for the SLF4J API bundle in the SpringSource Enterprise Bundle Repository [ref] clearly shows it was modified by SpringSource and is not an official SLF4J JAR:

```
Bundle-SymbolicName: com.springsource.slf4j.api
```

Don't worry too much about naming bundles, in the end you just need to give each bundle a unique enough name for your target deployment. You are free to rename your bundle later on if you wish, because by default the framework wires import packages to export packages regardless of bundle symbolic names. It is only when someone uses `Require-Bundle` (section 5.3.1) that consistent names become important. That's another reason why package dependencies are preferred over module dependencies, because they don't tie you down to a particular symbolic name forever.

Once you have decided on a name, the next step is to version your bundle. Determining the `Bundle-Version` is more straightforward than choosing the symbolic name, because pretty much every JAR file distribution is already identified by some sort of build version or release tag. On the other hand, version numbering schemes that don't match the recognized

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

OSGi format of `major.minor.micro.qualifier` will have to be converted before you can use them. Table 6.1 shows some actual project versions and attempts to map them to OSGi:

Table 6.1 Mapping real-world project versions to OSGi

Project version	**Suggested OSGi equivalent**
2.1-alpha-1	2.1.0.alpha-1
1.4-m3	1.4.0.m3
1.0_01-ea	1.0.1.ea
1.0-2	1.0.2
1.0.b2	1.0.0.b2
1.0a1	1.0.0.a1
2.1.7c	2.1.7.c
1.12-SNAPSHOT	1.12.0.SNAPSHOT
0.9.0-incubator-SNAPSHOT	0.9.0.incubator-SNAPSHOT
3.3.0-v20070604	3.3.0.v20070604
4aug2000r7-dev	0.0.0.4aug2000r7-dev

Not every version can be automatically converted to the OSGi version format. Look at the last example in the table; it starts with a number, but this is actually part of the date rather than the major version. This is the problem with free-form version strings, there is no standard way of comparing them or breaking them into component parts. OSGi versions on the other hand have standardized structure and well-defined ordering.

Once you have uniquely identified your bundle by name and version, you can add more information; a human-friendly `Bundle-Name`, a more detailed `Bundle-Description`, license details, vendor details, a link to online documentation, and so on. Most if not all of these details can be taken from existing project information, such as the following example from the second release of Google-Guice [ref]:

Listing 6.1 Example OSGi manifest from Guice 2

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Bundle-SymbolicName: com.google.inject
Bundle-Version: 2.0
Bundle-Name: guice
Bundle-Copyright: Copyright (C) 2006 Google Inc.
Bundle-Vendor: Google Inc.
Bundle-License: http://www.apache.org/licenses/LICENSE-2.0
Bundle-DocURL: http://code.google.com/p/google-guice/
Bundle-Description: Guice is a lightweight dependency injection
 framework for Java 5 and above
```

Remember that new OSGi bundles should also have this header:

```
Bundle-ManifestVersion: 2
```

Which tells the OSGi framework to process your bundle according to the latest specification. While this is not mandatory it is strongly recommended, because it enables additional checks and support for advanced modularity features offered by OSGi R4 specifications and beyond. Once you have captured enough bundle details to satisfactorily describe your JAR file, the next thing to decide is which packages it should export to other bundles in the framework.

6.1.2 Selecting what to export

Most bundles export at least one package, but a bundle doesn't actually have to export anything at all. Bundles providing service implementations via the service registry don't have to export any packages if they import their service API from another bundle. This is because their implementation is shared indirectly via the service registry and accessed using the shared API, as in Figure 6.2. But what about the package containing the `Bundle-Activator` class, doesn't that need to be exported? No, you don't need to export the bundle activator package unless you want to share it with other bundles. Indeed it is best practice to keep it private. As long as the activator class has a public modifier the framework can load it; even if it belongs to an internal, non-exported package.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

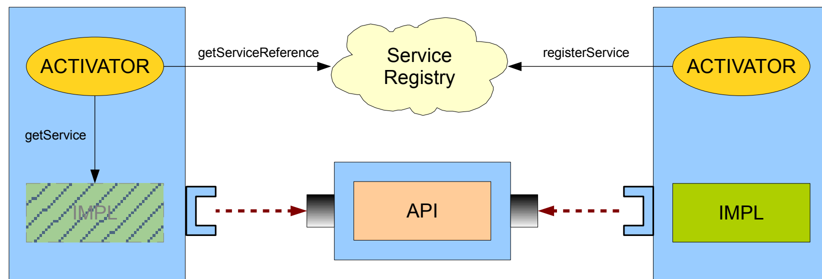


Figure 6.2 Sharing implementations without exporting their packages

So the question remains: when is it necessary for you to export packages and which packages in your JAR file do you actually need to export? The classic, non-OSGi approach is to export everything and make the entire contents of the JAR file visible. For API-only JAR files this is fine, but for implementation JAR files you don't want to expose internal details. Clients might then use and rely on these internal classes by mistake. As we'll see in a moment, exporting everything also increases the chance of conflicts among bundles containing the same package, particularly when they provide a different set of classes in those packages. When you're new to OSGi, exporting everything can look like a reasonable choice to begin with, especially if you don't know precisely where the public API begins or ends. On the contrary, you should really try to trim down the list of exported packages as soon as you have a working bundle.

Let's use a real-world example to demonstrate how to select your exports. Take a look at some of the packages containing classes and resources inside the core BeanUtils 1.8.0 library [ref] from Apache Commons:

```
org.apache.commons.beanutils
org.apache.commons.beanutils.converters
org.apache.commons.beanutils.locale
org.apache.commons.beanutils.locale.converters
org.apache.commons.collections
```

None of these packages seem private; there isn't an 'impl' or 'internal' package in the list, but the `org.apache.commons.collections` package is in fact an implementation detail. If you look closely at the BeanUtils Javadoc [ref], you will see this package actually contains a subset of the original Apache Commons Collections API [ref]. BeanUtils only uses a few of the Collections classes, and rather than have an execution-time dependency on the entire JAR file, the project embeds a copy of what it needs instead. So what happens when your application requires both the BeanUtils and Collections JAR files?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

This is typically not a problem in a non-OSGi environment because the application class loader exhaustively searches the entire class path to find a class. If both BeanUtils and Collections were on the same class path they would be merged together, with classes in BeanUtils overriding those from Collections, or vice versa depending on their ordering on the class path. Figure 6.3 (taken from chapter 2) shows an example of this:

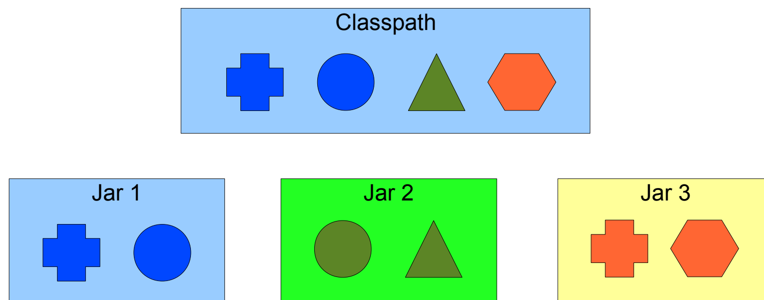


Figure 6.3 The classic application classloader merges JAR files into a single class space

One important caveat is that this only works if the BeanUtils and Collections versions are compatible. If you have incompatible versions on your class path then you would get runtime exceptions as the merged set of classes are inconsistent.

OSGi tries to avoid this by isolating bundles; bundles only see what they import and only if another bundle exports it. Unfortunately for our current example, this means if we exported `org.apache.commons.collections` from the BeanUtils bundle and the framework wires another bundle to it, it would only see the handful of collection classes from BeanUtils. They would not see the complete set of collection classes sitting in the Commons Collections bundle. To make sure this doesn't happen, we must exclude the partial package from our BeanUtils bundle's exports:

```
Export-Package: org.apache.commons.beanutils,
 org.apache.commons.beanutils.converters,
 org.apache.commons.beanutils.locale,
 org.apache.commons.beanutils.locale.converters
```

We can do this because the collections package does not belong to the main BeanUtils API. Now if it was purely an implementation detail never exposed to clients, our job would be complete. But there's a hitch: a class from the collections package is indirectly exposed to BeanUtils clients via a return type on some deprecated methods. What can we do? We could optionally import it, since the package like this:

```
Import-Package: org.apache.commons.collections;resolution:=optional
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

In the case, if the full package is available we will import it instead, but if it is not available then we can use the private copy. Would this work? It is better, but still not entirely accurate. Unfortunately, the only way to resolve this situation is to refactor the BeanUtils bundle to not contain the partial private copy of `org.apache.commons.collections`; see the sidebar if you want more details as to why an optional import won't work.

Revisiting "uses" constraints

So, we hypothesized about modifying our example BeanUtils bundle to optionally import `org.apache.commons.collections`. The idea was our bundle would import it if an exporter was available, but would use its private copy if not. This doesn't work, but why not? It is all about "uses" constraints as discussed in section 2.7.2.

As we mentioned, BeanUtils exposes a type from the collections package in a return type of a method in its exported types; this is a "uses" constraint by definition. To deal with this situation, we must express it somehow. For the sake of it, let's assume we follow the optional import case and we try to model the "uses" constraint correctly, like this:

```
Export-Package:
 org.apache.commons.beanutils; uses:="org.apache.commons.collections",
 org.apache.commons.beanutils.converters,
 org.apache.commons.beanutils.locale,
 org.apache.commons.beanutils.locale.converters
Import-Package: org.apache.commons.collections;resolution:=optional
```

This may actually work in some situations; for example, if our BeanUtils bundle, a BeanUtils and collections importing bundle, and a bundle exporting the collections package. In this case, all the bundles would get wired up to each other and everyone would be using the correct version of the collections packages. Great! But what would happen if the BeanUtils bundle was installed and resolved by itself first. In that case it wouldn't import the collections package (since there isn't one) and would use its private partial copy instead. Now if the the other bundles were installed and resolved, we'd end up with the wiring depicted in Figure 6.4.

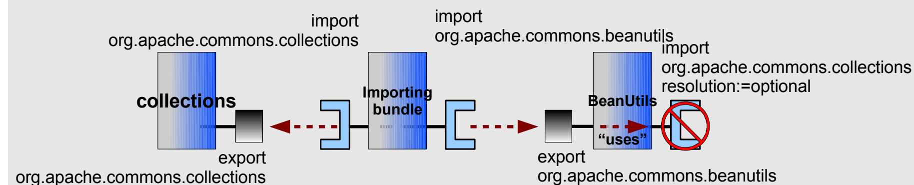


Figure 6.4 A "uses" constraint on an optionally imported package is ignored if the optionally imported package is not actually wired to an exporter

So, what does this mean? It means the BeanUtils bundle is using it own private copy of the collections types, while the importing bundle is using its imported collections types, so it will receive a `ClassCastException` if it uses any methods from BeanUtils that

expose collections types. In the end, there is no way to have a private copy of a package, if its types are exposed via exported packages. As we've concluded already, we must refactor our bundle to export preferably the whole package or to import the package.

Unfortunately, a surprising number of third-party libraries include partial packages, which can lead to similar situations. Some want to re-use code from another large library, but don't want to bloat their own JAR file. Some prefer to ship a single self-contained JAR file that clients can add to their class path without worrying about conflicting dependencies. Some libraries even use tools, such as JarJar [ref], to repackage internal dependencies under different namespaces to avoid potential conflicts. This leads to multiple copies of the same class all over the place, just because Java doesn't provide modularity out of the box! Renamed packages also make debugging harder and confuse developers. OSGi removes the need for renaming and helps you safely share packages while still supporting embedding.

At this point, you might decide it is a good time to refactor the API to make it more modular. Separating interfaces from their implementations can avoid the need for partial (or so-called "split") packages. This will help you reduce the set of packages you need to export and make your bundle more manageable. While this might not be an option for externally developed libraries, it is often worth taking time to contact the original developers to explain the situation. This happened a few years ago with the SLF4J project [ref] and they refactored their API to great effect.

Once you have your list of exported packages, you should consider versioning them. So which version should you use? The common choice is to use the bundle version, which implies the packages change at the same rate as the bundle, but some packages will inevitably change faster than others. You may also want to increment the bundle version because of an implementation fix while the exported API remains at the same level. Although everything starts out aligned you will probably find you need a separate version for each package (or at least each group of tightly-coupled packages). An example of this is the OSGi framework itself, which provides service APIs that have changed at different rates over time:

```
Export-Package: org.osgi.framework;version="1.4",
 org.osgi.service.packageadmin;version="1.2",
 org.osgi.service.startlevel;version="1.1",
 org.osgi.service.url;version="1.0",
 org.osgi.util.tracker;version="1.3.3"
```

Knowing which packages to export is only half of the puzzle of turning a JAR into a bundle, you also need to find out what should be imported. This is often the hardest piece of metadata to define, and causes the most problems when people migrate to OSGi.

6.1.3 Discovering what to import

Do you know what packages a given JAR file needs at execution time? Many developers have tacit or hidden knowledge of what JAR files to put on the class path. Such knowledge is often

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

gained from years of experience getting applications to run, where you reflexively add JAR files to the class path until any `ClassNotFoundExceptions` disappear. This leads to situations where an abundance of JAR files is loaded at execution time, not because they are all required, but because a developer feels they may be necessary based on past experience. Listing 6.2 shows an example class path for a J2EE client. Can you tell how these JAR files relate to one another, what packages they provide and use, and their individual versions?

Listing 6.2 Example class path for a J2EE client

```
concurrent.jar:getopt.jar:gnu-regexp.jar:jacorb.jar:\
jbossall-client.jar:jboss-client.jar:jboss-common-client.jar:\
jbosscx-client.jar:jbossha-client.jar:jboss-iiop-client.jar:\
jboss-j2ee.jar:jboss-jaas.jar:jbossjmx-ant.jar:jboss-jsr77-client.jar:\
jbossmq-client.jar:jboss-net-client.jar:jbosssx-client.jar:\
jboss-system-client.jar:jboss-transaction-client.jar:jcert.jar:\
jmx-connector-client-factory.jar:jmx-ejb-connector-client.jar:\
jmx-invoker-adaptor-client.jar:jmx-rmi-connector-client.jar:jnet.jar:\
jnp-client.jar:jsse.jar:log4j.jar:xdoclet-module-jboss-net.jar
```

With OSGi you explicitly define what packages your bundle needs and this knowledge is then available to any developer who wants it. They no longer have to guess how to compose their class path, the information is readily available in the metadata! It can also be used by tools such as the OSGi Bundle Repository (OBR) [ref] to automatically select and validate collections of bundles for deployment. This means any developer turning a JAR file into a bundle has a great responsibility in defining the correct set of imported packages. If this list is incomplete or too excessive it affects all users of the bundle. Unfortunately standard Java tools do not provide an easy way to determine what packages a JAR file might use at execution time. Manually skimming the source for package names is time-consuming and unreliable. Byte-code analysis is more reliable and repeatable, which is especially important for distributed teams, but it can miss classes that are dynamically loaded by name. For instance:

```
String name = someDynamicNameConstruction(someSortOfContext);
Class<?> clazz = someClassLoader.loadClass(name);                    #A
```

#A this could load a class from any package!

The ideal solution is to use a byte-code analysis tool like “bnd” [ref] followed by a manual review of the generated metadata by project developers. You can then decide whether to keep generating the list of imported packages for every build, or generate the list once and save it to a version controlled file somewhere so it can be pulled into later builds. Most tools for generating OSGi manifests also let you supplement or override the generated list, in case the manual review finds missing or incorrect packages. Once you are happy with the metadata you should run integration tests on an OSGi framework to verify the bundle has the necessary imported packages. You don't want to get a `ClassNotFoundException` in

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

production when an obscure but important piece of code runs for the very first time, and attempts to access a package that hasn't been imported!

We'll show you how to test OSGi applications later on in chapter 8. For now let's continue with our BeanUtils example and use bnd to discover what imports we need. The bnd tool was developed by one of the founders of OSGi, Peter Kriens, and provides a number of Ant tasks and command-line commands specifically designed for OSGi. Bnd uses a "pull" approach to divide a single class path into separate bundles based on a set of instructions. This means we have to tell bnd what packages we want to pull in and export, as well as those we want to pull in and keep private. Bnd instructions take the form of "name:value" properties, which means you can mix normal manifest entries along with bnd instructions. The following instructions select the exported and non-exported (or so-called private) packages that should be contained in our final bundle, along with the optional Collections import we discussed back in 6.1.2. Let's put them in a file named "beanutils.bnd", which you can find in the example code under "chapter06/BeanUtils-example":

```
Export-Package: org.apache.commons.beanutils.*                           #A
Private-Package: org.apache.commons.collections                          #B
Import-Package: org.apache.commons.collections.*;resolution:=optional,*   #C
```

#A public BeanUtils API
#B internal collection classes
#C collections is optional

Bnd accepts package wildcards, which it expands according to what it actually finds in the project byte code. In addition to accepting OSGi manifest headers as instructions it also adds some of its own, such as `Include-Resource` and `Private-Package`, to give you more control over exactly what goes into the bundle. These are not used by the OSGi framework at execution time. You will also notice that the import instruction contains both our optional collections import followed by a global wildcard. This tells bnd to add all the calculated imports and not filter any out. Once we have chosen our exported and internal packages, we invoke the bnd wrap task by passing it the original BeanUtils JAR file along with our custom bnd instructions:

```
$ cd chapter06/BeanUtils-example

$ java -jar ../../lib/bnd-0.0.355.jar \
    wrap -properties beanutils.bnd commons-beanutils-1.8.0.jar           #A
```

#A wrap as bundle

Bnd processes the JAR file using our instructions and generates a new file alongside the original, called "commons-beanutils-1.8.0.bar". We can extract the OSGi enhanced manifest from the newly created BeanUtils bundle like so:

```
$ jar xvf commons-beanutils-1.8.0.bar META-INF/MANIFEST.MF
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

As you can see it contains the following generated list of imported packages:

```
Import-Package: org.apache.commons.beanutils,org.apache.commons.beanut
 ils.converters,org.apache.commons.beanutils.expression,org.apache.com
 mons.beanutils.locale,org.apache.commons.beanutils.locale.converters,
 org.apache.commons.collections.comparators;resolution:=optional,org.a
 pache.commons.collections.keyvalue;resolution:=optional,org.apache.co
 mmons.collections.list;resolution:=optional,org.apache.commons.collec
 tions.set;resolution:=optional,org.apache.commons.logging
```

There are a couple of interesting points about this list. First, bnd has added imports for all the BeanUtils packages that we want to export. As we discussed in 5.1.1, this is good practice when exporting an API because it means that if (for whatever reason) an existing bundle already exports these packages, then we will share the same class space for the API. Without these imports our bundle would sit on its own little island, isolated from any bundles already wired to the previous package exporter. Second, bnd has found byte code references to the Apache Commons logging package, which isn't contained in the BeanUtils JAR file and must therefore be imported. Just think, we can now tell what packages a JAR file needs at execution time by checking the imported package list in the manifest. This is extremely useful for automated deployment of applications. Such a system would know when deploying BeanUtils that it should also deploy Commons Logging (or another bundle that provides the same package, like SLF4J). But which particular version of Logging should it deploy?

Just as with exported packages you should consider versioning your imports. Chapter 2 explained how versioning helps ensure binary compatibility with other bundles. You should try to use ranges rather than leave versions open-ended, because it protects you against potentially breaking API changes in the future. For example, consider:

```
Import-Package: org.slf4j;version="1.5.3”
```

This matches any version of the SLF4J API from 1.5.3 onwards, even unforeseen future releases which could be incompatible with our code. One recommended practice is to use a range starting from the minimum acceptable version up to, but not including, the next major version. (This assumes a change in major version indicates the API is not binary compatible.) For example, if we tested against the 1.5.3 SLF4J API we might use the following range:

```
Import-Package: org.slf4j;version="[1.5.3,2)"
```

This ensures only versions from the tested level to just before the next major release will be used. Not all projects follow this particular versioning scheme, you may need to tweak the range to narrow or widen the set of compatible versions. The width of the import range also depends on how you're using the package. Consider a simple change like adding a method to an interface, which typically occurs during a point release (such as 1.1 to 1.2). If you are just calling the interface this change would not affect you. If on the other hand you are implementing the interface then this would definitely break you, as you now need to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

implement a new method. You can imagine adding the right version ranges to imported packages takes time and patience, but this is often a one-time investment that pays off many times over during the life of a project. Tools such as bnd can help by detecting existing version metadata from dependencies on the class path and by automatically applying version ranges according to a given policy.

Unfortunately tools aren't perfect. While you're reviewing the generated list of imported packages you might notice a few that aren't actually used at execution time. Some code may only be executed in certain scenarios, like an Ant build task that's shipped with a library JAR file for convenience. Other JAR files might dynamically test for available packages and adapt their behavior at execution time to match what's actually installed. In such cases it is useful to mark these imports as optional to tell the OSGi framework the bundle can still work even when these packages are not available. Table 6.2 shows some real-world packages often considered as optional:

Table 6.2 Common optional imported packages found in third-party libraries

Package	**Used for**
javax.swing.*	GUI classes (could be interactive tests)
org.apache.tools.ant.*	ANT taskdefs (build time)
antlr.*	Parsing (maybe build / test related)
sun.misc.*	Sun implementation classes (like BASE64)
com.sun.tools.*	Sun tool support (javac, debugging, etc.)

As we saw back in section 5.2.1, OSGi provides two ways to mark a package as optional. You can either mark packages with the `resolution:=optional` directive or list them as dynamically imported packages. For packages you never expect to be used at execution time, like the Ant packages, we suggest you either use the optional attribute or even remove them from the list of imported packages. Use `resolution:=optional` when you know the bundle will always be used in the same way, once installed. If you want a more adaptive bundle that reacts to the latest set of available packages then you should list them as dynamic imports.

If you are new to OSGi and unsure exactly what packages your JAR file uses, consider using:

```
DynamicImport-Package: *
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

This makes things similar to the classic model where requests to load a new class always result in a query to the complete class path. It also allows your bundle to successfully resolve regardless of what packages are actually available. The downside is that you are pushing the responsibility of finding the right set of bundles onto users, because you don't provide any metadata defining what you really need! So this approach should only be considered as a stopgap measure to get you started.

We've now chosen the exports and imports for our new bundle. Every non-optional, non-dynamic package we import (but don't export) must be provided by another bundle. Does this mean for every JAR file we convert into a bundle, we also need to convert each of its dependencies into bundles? Not necessarily, because unlike standard JAR files OSGi supports embedding JAR files inside bundles.

6.1.4 Embedding vs. importing

Sometimes a JAR file has a close dependency on another JAR file. It might be they only work together, the dependency may be an implementation detail you want to keep private, or you may not want to share the statics in the JAR file with other bundles. In these situations it makes more sense to embed these dependencies inside the primary JAR file when you turn it into a bundle. Embedding the JAR file is easier than converting both JAR files to bundles because you can ignore packages that would otherwise need to be exported and imported between them. The downside of embedding is that it adds unnecessary weight for non-OSGi users, who can't use the embedded JAR file unless the bundle is first unpacked. Figure 6.5a shows how a CGLIB bundle might embed ASM, a small utility for processing byte code.

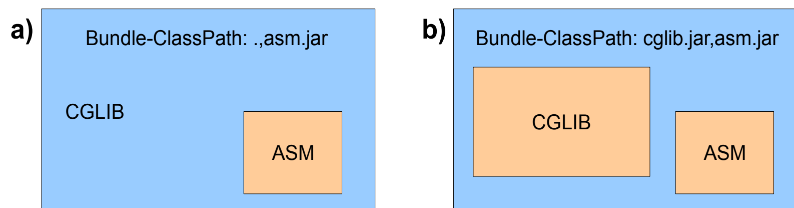


Figure 6.5 Embedding tightly-coupled dependencies in a bundle

Alternatively you might consider creating a new bundle artifact that embeds all the related JAR files together instead of turning the primary JAR file into a bundle. This aggregate bundle could then be provided separately to OSGi users without affecting users of the original JAR files. Figure 6.5b shows how you could use this approach for the CGLIB library. While this does mean you have an extra deliverable to support, it also gives you an opportunity to override or add classes for better interoperability with OSGi. We'll see an example of this in a moment and also later on in section 6.2.1. This often happens when

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

libraries use external connections or background threads, which ideally should be managed by the OSGi lifecycle layer. Such libraries are said to have “state”.

6.1.5 Adding lifecycle support

You might not realize it when you use a third-party library, but a number of them actually have a form of state. This state could take the form of a background thread, a filesystem cache, or a pool of database connections. Libraries usually provide methods to manage this state, such as cleaning up resources and shutting down threads. Often you don't bother calling these methods because the life of the library is the same as the life of your application. In OSGi this is not necessarily the case; your application could still be running after the library has been stopped, updated, and restarted many times. On the other hand, the library could still be available in the framework long after your application has come and gone. You need to tie the library state to its bundle lifecycle and to do that you need to add a bundle activator (see section 3.4.1).

The original HttpClient library [ref] from Apache Commons manages a pool of threads for multi-threaded connections. These threads are started lazily so there is no need to explicitly initialize the pool, but the library provides a method to shut down and clean everything up:

```
MultiThreadedHttpConnectionManager.shutdownAll();
```

If we wanted to wrap the HttpClient library JAR file up as a bundle, we might decide to add an activator that shuts down the thread pool whenever the HttpClient bundle was stopped:

Listing 6.3 Bundle-Activator for HttpClient library

```
package org.apache.commons.httpclient.internal;

import org.apache.commons.httpclient.MultiThreadedHttpConnectionManager;
import org.osgi.framework.*;

public class Activator implements BundleActivator {
  public void start(BundleContext ctx) {}

  public void stop(BundleContext ctx) {
    MultiThreadedHttpConnectionManager.shutdownAll();
  }
}
```

We have to tell OSGi about this activator by adding metadata to the manifest:

```
Bundle-Activator: org.apache.commons.httpclient.internal.Activator
```

You can see this in action by building and running the following example:

```
$ cd chapter06/HttpClient-example
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
$ ant dist

$ java -jar launcher.jar bundles
```

You should see it start and attempt to connect to the internet (ignore Log4J warnings):

```
GET http://www.google.com/
GOT 5500 bytes
->
```

If you use `jstack` to see what threads are running in the JVM, one of them should be:

```
"MultiThreadedHttpConnectionManager cleanup" daemon
```

Stop the HttpClient bundle, which should cleanup the thread pool, and check again:

```
-> stop 5
```

The `MultiThreadedHttpConnectionManager` thread should now be gone. Unfortunately this is not a complete solution, because if you stop and re-start the test bundle the thread pool manager will re-appear – even though the HttpClient bundle is still stopped! Restricting use of the HttpClient library to the bundle active state would require all calls to go through some sort of delegating proxy, or ideally the OSGi service registry. Thankfully the latest 4.0 release of the HttpClient library makes it much easier to manage connection threads inside a container such as OSGi and removes the need for this single static shutdown method.

Bundle activators are mostly harmless because they don't interfere with non-OSGi users of the JAR file. They are only referenced via the bundle metadata and aren't considered part of the public API. They just sit there unnoticed and unused in classic Java applications, until the bundle is loaded into an OSGi framework and started. Whenever you have a JAR file with implicit state or background resources, consider adding an activator to help OSGi users.

We've now covered most aspects of turning a JAR file into a bundle: identity, exports, imports, embedding, and lifecycle management. How many best practices can you remember? Wouldn't it be great to have them summarized as a one-page cheatsheet?

Well look no further than the following page...

6.1.6 JAR file to bundle cheatsheet

The following cheatsheet provides a handy summary for converting JAR files into bundles:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Figure 6.6 JAR file to bundle cheatsheet

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

OK, we know how to take a single JAR file and turn it into a bundle, but what about a complete application? We could simply take our existing JAR, EAR, and WAR files and turn them all into bundles or we could choose to wrap everything up as a single application bundle. Surely we can do better than that. What techniques can we use to bundle up an application and what are the pros and cons? For the answers to this and more, read on!

6.2 Splitting an application into bundles

Most applications are usually made up of one or more JAR files. One way to migrate an application to OSGi is to take these individual JAR files and convert each of them into a bundle using the techniques discussed in the previous section. Converting lots of JAR files is time consuming (especially for beginners), so a simpler approach is to take your complete application and wrap it up as a single bundle. In this section we'll show how to start from such a single application bundle and suggest ways of dividing it further into multiple bundles. Along the way we'll look at how you can introduce other OSGi features, such as services, to make your application more flexible. Finally we'll suggest places where it doesn't make sense to introduce a bundle.

Let's start with the single application bundle or so-called "mega" bundle.

6.2.1 Making a mega bundle

A mega bundle comprises a complete application along with its dependencies. Anything the application needs on top of the standard JDK is embedded inside this bundle and made available to the application by extending the `Bundle-ClassPath` (2.5.3). This is very similar to how Java Enterprise applications are constructed. In fact, you can take an existing web application archive (also known as a WAR file) and easily turn it into a bundle by adding an identity along with a `Bundle-ClassPath` entries for the various classes and libraries contained within it, as shown in Figure 6.7.

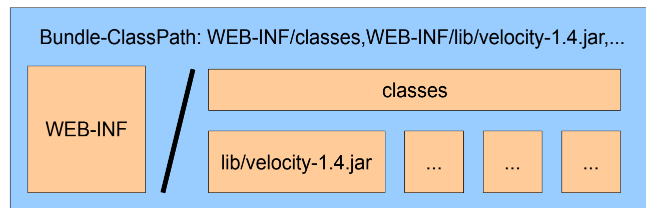


Figure 6.7 Turning a WAR file into a bundle

The key benefit of a mega bundle is that it drastically reduces the number of packages you need to import, sometimes down to no packages at all. The only packages you might need to import are non-`java.*` packages from the JDK (such as `javax.*` packages) or any packages provided by the container itself. Even then you could choose to access them via

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

OSGi “boot delegation” by setting the `org.osgi.framework.bootdelegation` framework property to the list of packages you want to inherit from the container class path. Boot delegation can also avoid certain legacy problems (see section 9.2 for the gory details). The downside is that it reduces modularity, since you cannot override boot-delegated packages in OSGi. A mega bundle with boot delegation enabled is very close to the classic Java application model, the only difference is each application has its own class loader instead of sharing the single JDK application class loader.

JEDIT MEGA-BUNDLE EXAMPLE

This is an action book, so let's shelve the theoretical discussion for the moment and create our own mega bundle based on jEdit, a pluggable Java text editor [ref]. The sample code for this book comes with a copy of the jEdit 4.2 source, which you can unpack like so:

```
$ cd chapter06/jEdit-example

$ ant jEdit.unpack

$ cd jEdit
```

The jEdit build uses Apache Ant [ref], which is good news because it means we can use bnd's Ant tasks to generate OSGi manifests. Maven users should not feel left out though, as they can use the maven-bundle-plugin [ref] which also uses bnd under the covers. So how exactly do we add bnd to the build? Listing 6.4 shows the main target from the original (non-OSGi) jEdit `build.xml`:

Listing 6.4 default jEdit build target

```
<target name="dist" depends="compile,compile14"
  description="Compile and package jEdit.">

  <jar jarfile="jedit.jar"
    manifest="org/gjt/sp/jedit/jedit.manifest"
    compress="false">

    <fileset dir="${build.directory}">
      <include name="bsh/**/*.class"/>
      <include name="com/**/*.class"/>
      <include name="gnu/**/*.class"/>
      <include name="org/**/*.class"/>
    </fileset>

    <fileset dir=".">
      <include name="bsh/commands/*.bsh"/>
      <include name="gnu/regexp/MessagesBundle.properties"/>
      <include name="org/gjt/sp/jedit/**/*.dtd"/>
      <include name="org/gjt/sp/jedit/icons/*.gif"/>
      <include name="org/gjt/sp/jedit/icons/*.jpg"/>
      <include name="org/gjt/sp/jedit/icons/*.png"/>
      <include name="org/gjt/sp/jedit/*.props"/>
```

```
        <include name="org/gjt/sp/jedit/actions.xml"/>
        <include name="org/gjt/sp/jedit/browser.actions.xml"/>
        <include name="org/gjt/sp/jedit/dockables.xml"/>
        <include name="org/gjt/sp/jedit/services.xml"/>
        <include name="org/gjt/sp/jedit/default.abbrevs"/>
      </fileset>
    </jar>
  </target>
```

The `jar` task is configured to take a static manifest file, "org/gjt/sp/jedit/jedit.manifest". If we didn't want to change the build process, but still wanted an OSGi enabled manifest then we could take the jEdit binary, run it through an analyzer like bnd, and add the generated OSGi headers to this static manifest. As mentioned back in 6.1.3, this approach is fine for existing releases or projects that don't change much. On the other hand, integrating a tool such as bnd with your build means you get immediate feedback about the modularity of your application rather than when you actually try to deploy it.

Replacing the jar task with bnd

We're going to make things more dynamic and generate OSGi metadata during the build. This is the recommended approach because we don't have to remember to check and regenerate the metadata after significant changes to the project source. This is especially useful in the early stages of a project when responsibilities are still being allocated.

There are several ways to integrate bnd with a build:

5. use bnd to generate metadata from classes before creating the JAR file
6. create the JAR file as normal and then post-process it with bnd
7. use bnd to generate the JAR file instead of using the Ant `jar` task

If you really need certain features of the `jar` task like indexing, you should use the first or second option. If you are post-processing classes or need to filter resources then choose either the second or third option. We're going to go with the third option to demonstrate how easy it is to switch your build over to bnd. It will also help us later on in 6.2.2 when we start partitioning the application into separate bundles.

First we comment out the `jar` task:

```
<!-- jar jarfile="jedit.jar"                                       #A
  manifest="org/gjt/sp/jedit/jedit.manifest"                       #B
  compress="false">
...
</jar -->
```

#A where to put the JAR
#B fixed manifest entries

Next we add the `bnd` task below it:

```
<taskdef resource="aQute/bnd/ant/taskdef.properties"
  classpath="../../../lib/bnd-0.0.355.jar" />                      #A
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
<bnd classpath="${build.directory}"                                      #B
  files="jedit-mega.bnd" />                                              #C
```

#A location of bnd tasks
#B project class path
#C bnd instructions

There is one key difference between the `jar` and `bnd` tasks that you must remember:

- The `jar` task takes a list of files and directories and copies them all into a single JAR file.
- The `bnd` task takes a class path and a list of instruction files (one file per bundle) that tell it which classes and/or resources to copy from the class path into each bundle.

So if you don't tell bnd to pull a certain package into the bundle, don't be surprised if it's not there. We're building a single mega bundle so we only need one instruction file, which we'll call "jedit-mega.bnd". The first thing we must add is an instruction to tell bnd where to put the generated bundle:

```
-output: jedit.jar
```

The `bnd` task can also copy additional manifest headers into the final manifest, so let's ask bnd to include the original jEdit manifest rather than duplicate its content in our new file:

```
-include: org/gjt/sp/jedit/jedit.manifest
```

We could have left the manifest file where it was, added our instructions to it, and passed that into bnd, but this would make it harder for people to separate out the new build process from the original. It is also better to have the bnd instructions at the project root where they are more visible. We can now try to rebuild the project from inside the "jEdit" directory:

```
$ ant dist
...
[bnd] Warnings
[bnd] Neither Export-Package, Private-Package, -testpackages is set,
      therefore no packages will be included
[bnd] Did not find matching referal for *
[bnd] Errors
[bnd] The JAR is empty
```

Adding bnd instructions

What went wrong? Well we forgot to tell bnd what packages to pull into our new bundle! Using the JAR to bundle cheatsheet from 6.1.6 let's add the following bundle headers to "jedit-mega.bnd", along with a bnd specific instruction to pull in all classes and resources from the build class path and keep them private:

```
Bundle-Name: jEdit
Bundle-SymbolicName: org.gjt.sp.jedit
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Bundle-Version: 4.2

Private-Package: *                                                      #A
```

#A pull in everything as private

Remember that bnd supports wildcard package names, so we can use * to represent the entire project. While this is useful when creating mega bundles, you should be careful using wildcards when separating a class path into multiple, separate bundles or when already bundled dependencies appear on the class path. Always check the content of your bundles to make sure you aren't pulling in additional packages by mistake. Getting back to the task at hand, when we rebuild the jEdit project we now see:

```
$ ant dist
...
[bnd] # org.gjt.sp.jedit (jedit.jar) 849
```

Success! Let's try to run our new JAR file:

```
$ java -jar jedit.jar
```

Whoops, something else went wrong:

```
Uncaught error fetching image:
java.lang.NullPointerException
 at sun.awt.image.URLImageSource.getConnection(Unknown Source)
 at sun.awt.image.URLImageSource.getDecoder(Unknown Source)
 at sun.awt.image.InputStreamImageSource.doFetch(Unknown Source)
 at sun.awt.image.ImageFetcher.fetchloop(Unknown Source)
 at sun.awt.image.ImageFetcher.run(Unknown Source)
```

Adding resource files

It seems our JAR file is missing some resources, can you see why? Look closely at the `jar` task in Listing 6.4; notice how classes come from `${build.directory}`, but the resource files come from "." (the project root). We could write a lengthy `Include-Resource` instruction to tell bnd to pull in these resources, but there is a much easier solution. Simply put the existing resource fileset inside a `copy` task to copy matching resources to the build directory before the `bnd` task runs:

```
<copy todir="${build.directory}">
  <fileset dir=".">
    <include name="bsh/commands/*.bsh"/>
    <!-- and so on... -->
  </fileset>
</copy>
```

The resource files can now be found on the build class path. Rebuild and run jEdit again:

```
$ ant dist
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
...
[bnd] # org.gjt.sp.jedit (jedit.jar) 1003

$ java -jar jedit.jar
```

Bingo! You should see the main jEdit window appear, as shown in Figure 6.8:

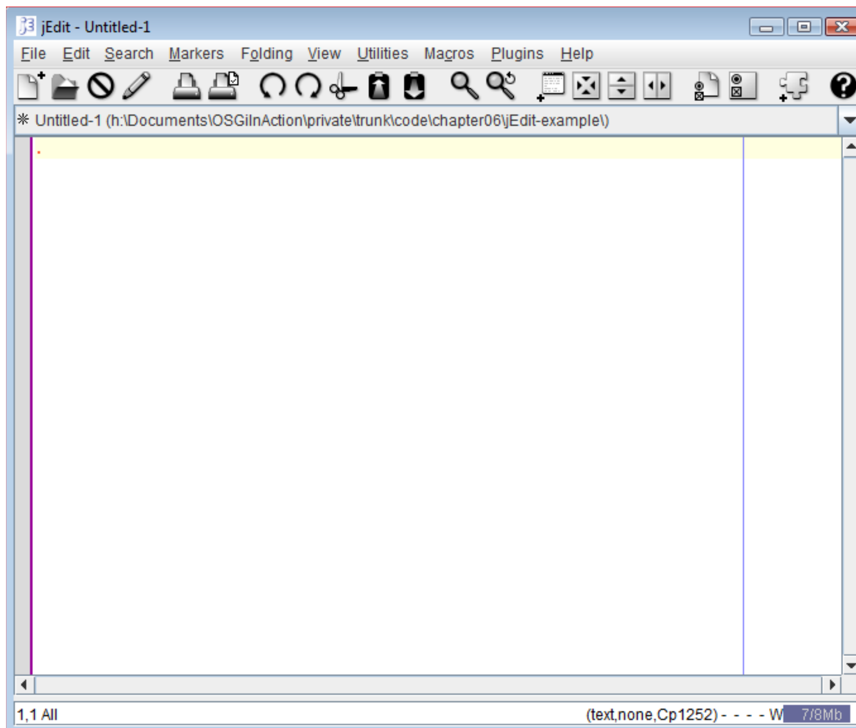


Figure 6.8 Main jEdit window

Our bundle works as a classic JAR file, but will it work as a bundle? Let's review the manifest:

Listing 6.5 jEdit mega bundle manifest

```
Manifest-Version: 1.0
Created-By: 1.6.0_13 (Sun Microsystems Inc.)
Bnd-LastModified: 1250524748304
Tool: Bnd-0.0.355
Main-Class: org.gjt.sp.jedit.jEdit
Bundle-ManifestVersion: 2
Bundle-Name: jEdit
Bundle-SymbolicName: org.gjt.sp.jedit
Bundle-Version: 4.2
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Private-Package: org.gjt.sp.jedit.icons,org.gjt.sp.jedit.help,org.obje
 ctweb.asm,org.gjt.sp.util,org.gjt.sp.jedit,org.gjt.sp.jedit.syntax,bs
 h.reflect,org.gjt.sp.jedit.pluginmgr,bsh.commands,org.gjt.sp.jedit.pr
 int,org.gjt.sp.jedit.menu,org.gjt.sp.jedit.browser,org.gjt.sp.jedit.p
 roto.jeditresource,org.gjt.sp.jedit.io,org.gjt.sp.jedit.options,com.m
 icrostar.xml,gnu.regexp,bsh.collection,org.gjt.sp.jedit.search,org.gj
 t.sp.jedit.gui,org.gjt.sp.jedit.buffer,org.gjt.sp.jedit.msg,installer
 ,org.gjt.sp.jedit.textarea,bsh
Import-Package: javax.print.attribute,javax.print.attribute.standard,j
 avax.swing,javax.swing.border,javax.swing.event,javax.swing.filechoos
 er,javax.swing.plaf,javax.swing.plaf.basic,javax.swing.plaf.metal,jav
 ax.swing.table,javax.swing.text,javax.swing.text.html,javax.swing.tre
 e
```

Our jEdit bundle doesn't export any packages, but it does use packages from Swing. These should come from the system bundle, which is typically setup to export JDK packages (although this can be overridden). You might be wondering if we should add version ranges to the packages imported from the JDK. This is not required as most system bundles don't version their JDK packages. You only need to version these imports if you wanted to use another implementation that's different from the stock JDK version. We should also mention the final manifest contains some bnd specific headers that are not used by the OSGi framework (such as `Private-Package`, `Tool`, and `Bnd-LastModified`). They are left as a record of how bnd built the bundle and can be removed by adding this bnd instruction to "jedit-mega.bnd":

```
-removeheaders: Private-Package,Tool,Bnd-LastModified
```

The new manifest looks correct, but the real test is yet to come. We must now try to deploy and run our bundle on an actual OSGi framework. Will it work first time or fail with an obscure exception?

Running jEdit with OSGi

We can deploy our jEdit bundle by using the same simple launcher used to launch the earlier paint examples. Remember this launcher will first install any bundles found in the directory and then use the first `Main-Class` header it finds to bootstrap the application. Our manifest already has a `Main-Class` so we just need point the launcher at the "jEdit" directory, like so:

```
$ cd ..

$ cp ../../launcher/dist/launcher.jar .

$ java -jar launcher.jar jEdit
```

Unfortunately something's not quite right. While the bundle installs and the application starts, it hangs at the splash screen in Figure 6.9 and the main jEdit window never appears.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Figure 6.9 jEdit when first run as OSGi bundle

If we look closely at the top of the stack trace we see the following warning message:

```
java.net.MalformedURLException: Unknown protocol: jeditresource
```

Why would this work when the bundle was run as a classic application, but not when the bundle was installed in an OSGi framework? The answer lies in the URL Handlers service we discussed briefly back in 4.6.1. To implement this service, the OSGi framework installs its own `URLStreamHandlerFactory` which delegates requests to handlers installed via the service registry. Unlike the default `URLStreamHandlerFactory` this implementation does not automatically scan the class path for URL handlers. Instead all URL handlers must be registered as OSGi services, which also means the handlers are tied to their bundle lifecycle.

Fixing the URL handler issue

Your first thought might be to try and disable the URL Handlers service so it doesn't install this factory. Unfortunately, there is no standard switch for this, but to disable it in Felix you would set the `felix.service.urlhandlers` framework property to 'false'. Turning off the global URL Handlers service also has serious implications. It means no bundle can contribute dynamic protocol handlers, which would break applications that rely on the URL Handlers service. It also wouldn't fix this particular problem because the "jeditresource" handler is not visible to the default `URLStreamHandlerFactory` when we run jEdit as a bundle. The JDK's URL Handler factory uses `Class.forName()` to search the application class path for valid handlers, but our "jeditresource" handler is hidden from view inside the jEdit bundle class loader.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The solution is to register the “jeditresource” handler as a `URLStreamHandlerService` when the jEdit bundle is started and remove it when it is stopped. But how can we add OSGi specific code without affecting classic jEdit users? Cast your mind back to section 6.1.5 where we talked about using lifecycles to manage external resources. This is exactly the sort of situation that requires a bundle activator, such as the one shown in Listing 6.6:

Listing 6.6 Bundle Activator to manage “jeditresource” handler

```
package org.gjt.sp.jedit;

import java.io.IOException;
import java.net.*;
import java.util.Properties;

import org.osgi.framework.*;
import org.osgi.service.url.*;

import org.gjt.sp.jedit.proto.jeditresource.Handler;

public class Activator implements BundleActivator {
  private static class JEditResourceHandlerService                    #A
    extends AbstractURLStreamHandlerService {
    private Handler jEditResourceHandler = new Handler();              #B

    public URLConnection openConnection(URL url)
      throws IOException {
      return jEditResourceHandler.openConnection(url);                 #C
    }
  }

  public void start(BundleContext context) {
    Properties properties = new Properties();
    properties.setProperty(URLConstants.URL_HANDLER_PROTOCOL,          #D
        "jeditresource");

    context.registerService(                                           #E
      URLStreamHandlerService.class.getName(),
      new JEditResourceHandlerService(),
      properties);
  }

  public void stop(BundleContext context) {}                           #F
}
```

#A OSGi service wrapper class
#B real handler instance
#C delegate to real handler
#D for “jeditresource” protocol
#E publish URL handler service
#F automatically removed on stop

Once we've added this activator class to the build we must remember to declare it in the OSGi metadata, otherwise it will never get called. This is a common cause of head scratching

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

for people new to OSGi, because the framework can't tell when you accidentally forget a `Bundle-Activator` header. So when you've added an activator, but it's having no effect, always check your manifest to make sure it's been declared – it saves a lot of hair!

```
Bundle-Activator: org.gjt.sp.jedit.Activator
```

Our activator code uses OSGi constants and interfaces, so we must add the core OSGi API to the compilation class path in the jEdit `build.xml` otherwise our new code won't compile:

```
<javac ... >
        <classpath path="../../../lib/osgi.core.jar"/>
        <!-- the rest of the classpath -->
```

This API is only required when compiling the source, it won't be necessary at runtime unless the activator class is explicitly loaded. One more build and we now have a JAR file that can run as a classic Java application or an OSGi bundle! Our final set of bnd instructions is shown in Listing 6.7.

Listing 6.7 Final bnd instructions for jEdit mega bundle

```
-output: jedit.jar

-include: org/gjt/sp/jedit/jedit.manifest

Bundle-Name: jEdit
Bundle-SymbolicName: org.gjt.sp.jedit
Bundle-Version: 4.2

Private-Package: *

-removeheaders: Private-Package,Tool,Bnd-LastModified

Bundle-Activator: org.gjt.sp.jedit.Activator
```

One last wrinkle is that we have to tell jEdit where its installation directory is by using the `jedit.home` property. Normally jEdit can detect the installation directory containing its JAR file by peeking at the application class path, but this won't work when running it as a bundle on OSGi because the JAR file is loaded via a different mechanism.

```
$ ant dist

$ cd ..

$ java -Djedit.home=jEdit -jar launcher.jar jEdit
```

With this last piece of configuration in place you should see jEdit start and the main window appear, just as we saw before in Figure 6.8. It should also still work as a classic Java application.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Re-visiting mega-bundles

We've successfully created a mega bundle for jEdit with a small amount of effort. So what are the downsides of a mega bundle? Well, your application is still one single unit. You cannot replace or upgrade sections of it without shutting down the complete application and this might shutdown the whole JVM process if the application called `System.exit()`. Because nothing is being shared, you could end up with duplicate content between applications. Effectively you're in the same situation as before moving to OSGi, except with a few additional improvements in isolation and management. This doesn't mean the mega bundle approach is useless, as a first step it can be very reassuring being able to run your application on an OSGi framework with the minimum of fuss. It also provides a solid foundation for further separating (or slicing) your application into bundles, which is the focus of the next section.

6.2.2 Slicing code into bundles

We now have a single mega bundle containing our entire application. The next step towards a full-fledged flexible OSGi application is to start breaking it into bundles that can be upgraded independently of one another. How and where should you draw the lines between bundles? Bundles import and export packages in order to share them, so it makes sense to draw lines that minimize the number of imports and exports. If you have a high-level design document showing the major components and their boundaries, then you could take each major component and turn it into a bundle. If you don't have such a document, you should look for major areas of responsibility; such as business logic, data access, graphical components. Each major area could be represented by a bundle, as depicted in Figure 6.10.

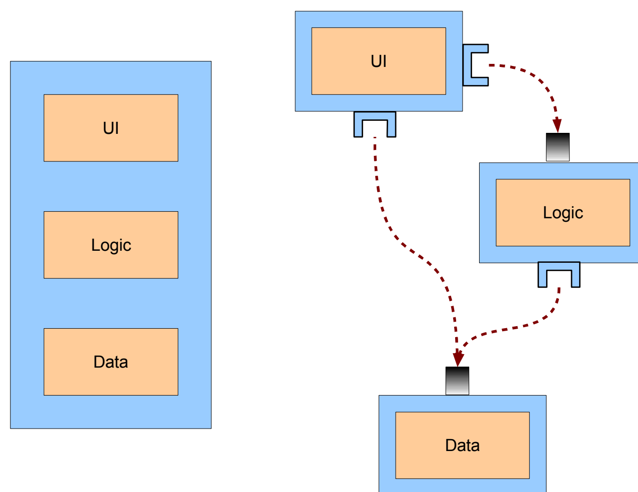


Figure 6.10 Slicing code into bundles

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

CUT ALONG THE DOTTED LINES

Returning to our jEdit example, what areas suggest themselves as potential bundles? Well the obvious choice to begin with is to separate out the jEdit code from third-party libraries, then try to extract the main top-level package. But how do we go about dividing the project class path into different bundles? Remember what we said about bnd back in 6.1.3, that it uses a "pull" approach to assemble bundles from a project class path based on a list of instruction files. All we need to do is provide our `bnd` task with different instruction files for each bundle:

```
<bnd classpath="${build.directory}"
  files="jedit-thirdparty.bnd,jedit-main.bnd,jedit-engine.bnd" />        #A
```

#A divide into three bundles

The first bundle will contain all third-party classes, basically any package from the build directory that doesn't start with `org.gjt.sp`. Bnd makes this very easy by allowing negated packages. For example:

```
Private-Package: !org.gjt.sp.*, *
```

This copies all other packages into the bundle and keeps them private. Using the earlier "jedit-mega.bnd" as a template we can flesh out the rest to get the "jedit-thirdparty.bnd" file shown in Listing 6.8. We also exclude the `installer` package because this is not actually required at execution time and doesn't belong in the third-party library bundle.

Listing 6.8 Initial bnd instructions for jEdit third-party library bundle

```
-output: jedit-thirdparty.jar

Bundle-Name: jEdit Third-party Libraries
Bundle-SymbolicName: org.gjt.sp.jedit.libs
Bundle-Version: 4.2

Private-Package: !org.gjt.sp.*, !installer.*, *

-removeheaders: Private-Package,Tool,Bnd-LastModified
```

The second bundle will contain the top-level package containing the main jEdit class. We should also add the `org.gjt.sp.jedit.proto` package containing the URL handler code because this is only used by the bundle activator in the top-level package. Listing 6.9 shows our initial attempt at "jedit-main.bnd". You might notice that the only difference between this file and the mega bundle instructions in Listing 6.7 is the selection of private packages, everything else is exactly the same. The main bundle also replaces the mega bundle as the executable JAR file.

Listing 6.9 Initial bnd instructions for jEdit main bundle

```
-output: jedit.jar
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
-include: org/gjt/sp/jedit/jedit.manifest

Bundle-Name: jEdit
Bundle-SymbolicName: org.gjt.sp.jedit
Bundle-Version: 4.2

Private-Package: org.gjt.sp.jedit, org.gjt.sp.jedit.proto.*

-removeheaders: Private-Package,Tool,Bnd-LastModified

Bundle-Activator: org.gjt.sp.jedit.Activator
```

The third and final bundle will contain the rest of the jEdit packages, which we'll call the "engine" for now. It should contain all packages beneath the `org.gjt.sp` namespace, except the top-level jEdit package and packages under `org.gjt.sp.jedit.proto`. The resulting "jedit-engine.bnd" file appears in Listing 6.10. Notice how the same packages listed in the main instructions are negated in the engine instructions. Refactoring packages between bundles is as simple as moving entries from one instruction file to another.

Listing 6.10 Initial bnd instructions for jEdit engine bundle

```
-output: jedit-engine.jar

Bundle-Name: jEdit Engine
Bundle-SymbolicName: org.gjt.sp.jedit.engine
Bundle-Version: 4.2

Private-Package:\
 !org.gjt.sp.jedit, !org.gjt.sp.jedit.proto.*,\                          #A
 org.gjt.sp.*

-removeheaders: Private-Package,Tool,Bnd-LastModified
```

#A exclude main packages

Stitching the pieces together

We now have three bundles that together form the original class path, but none of them share any packages. If you tried to launch the OSGi application at this point it would fail because of unsatisfied imports between the three bundles. Should we just go-ahead and export everything by switching all `Private-Package` instructions to `Export-Package`? We could, but what would you learn by doing that? Let's try and export only what we absolutely need to share, keeping as much as possible private.

There are three ways we can find out which packages a bundle must export:

1. Gain an understanding of the codebase and how the packages relate to one other. This could involve the use of structural analysis tools such as Structure 101 [ref].

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

2. Read the `Import-Package` headers from the generated manifests to compile a list of packages that 'someone' needs to export. Ignore JDK packages, like `javax.swing`.
3. Repeatedly deploy the bundles onto a live framework and use any resulting error messages and/or diagnostic commands (such as the "diag" command on Equinox) to fine-tune the exported packages until all bundles resolve.

The first option requires patience, but the reward will be a thorough understanding of the package structure. It will also help you determine other potential areas that could be turned into bundles. The third option can be quick if the framework gives you the complete list of missing packages on the first attempt, but sometimes feels like an endless loop of "deploy, test, update". The second option is a good compromise of the other two. The bnd tool has already analyzed the codebase to come up with the list of imports and we already know the framework will follow the import constraints listed in the manifest. The structured manifest also means we can write a script to do the hard work for us. For example, consider this rather obscure command on Linux:

```
$ java -jar ../../lib/bnd-0.0.355.jar print jEdit/*.jar \
    | awk '/^Import-Package$/ {getline;ok=1} /^[^ ]/ {ok=0} \
           {if (ok) print $1}' | sort -u
```

It uses bnd to print a summary of each jEdit bundle, extracts the package names from the `Import-Package` part of the summary, and sorts them into a unique list. Once we remove the JDK and OSGi framework packages we get Listing 6.11.

Listing 6.11 packages imported by jEdit bundles

```
bsh                                                                    #A
com.microstar.xml
gnu.regexp

org.gjt.sp.jedit                                                       #B

org.gjt.sp.jedit.browser                                               #C
org.gjt.sp.jedit.buffer
org.gjt.sp.jedit.gui
org.gjt.sp.jedit.help
org.gjt.sp.jedit.io
org.gjt.sp.jedit.menu
org.gjt.sp.jedit.msg
org.gjt.sp.jedit.options
org.gjt.sp.jedit.pluginmgr
org.gjt.sp.jedit.search
org.gjt.sp.jedit.syntax
org.gjt.sp.jedit.textarea
org.gjt.sp.util
```

#A third-party packages
#B main jEdit package

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

#C other jEdit packages

It is clear that the third-party library bundle only needs to export three packages and the main jEdit bundle just the top-level package. Unfortunately, the jEdit engine bundle needs to export almost all of its packages, indicating a tight coupling between the engine and the top-level jEdit package. This suggests it would be better to merge these two bundles back together, unless we were going to refactor the code to reduce this coupling. Let's ignore this for now and press on, as this separation will eventually lead to an interesting class loading issue that is worth knowing about. Anyone who's curious can skip ahead to section 6.2.4.

What's next on the JAR to bundle checklist? Ah yes, versioning. We should version all the exported jEdit packages with the current bundle version (4.2), but we won't bother versioning the individual third-party packages at the moment, because it's not obvious what releases are being used. We can always add the appropriate versions in the future, when we divide the combined third-party bundle into separate library bundles. We should also add version ranges to our imports, as suggested back in 6.1.3. Rather than go to the hassle of explicitly writing out all the ranges, we can take advantage of another bnd feature and compute them:

```
-versionpolicy: [${version;==;${@}},${version;+;${@}})
```

This instruction tells bnd to take the detected version `${@}` and turn it into a range containing the current "major.minor" version `${version;==;...}` up to (but not including) the next major version `${version;+;...}` [ref]. So if the bnd tool knows that a package has a version of 4.1.8, it would apply a version range of "[4.1,5)" to any import of that package. We add this to each of our bnd files, along with the changes to export the necessary packages. You can see the final third-party and engine instructions in listings 6.12 and 6.13.

Listing 6.12 Final bnd instructions for jEdit third-party library bundle

```
-output: jedit-thirdparty.jar

Bundle-Name: jEdit Third-party Libraries
Bundle-SymbolicName: org.gjt.sp.jedit.libs
Bundle-Version: 4.2

Export-Package: bsh, com.microstar.xml, gnu.regexp
Private-Package: !org.gjt.sp.*, !installer.*, *

-versionpolicy: [${version;==;${@}},${version;+;${@}})
-removeheaders: Private-Package,Tool,Bnd-LastModified
```

Listing 6.13 Final bnd instructions for jEdit engine bundle

```
-output: jedit-engine.jar
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Bundle-Name: jEdit Engine
Bundle-SymbolicName: org.gjt.sp.jedit.engine
Bundle-Version: 4.2

Export-Package:\
 !org.gjt.sp.jedit,\
 !org.gjt.sp.jedit.proto.*,\
 org.gjt.sp.*;version="4.2"

-versionpolicy: [${version;==;${@}},${version;+;${@}})
-removeheaders: Private-Package,Tool,Bnd-LastModified
```

We still have one more (non-OSGi) tweak to make to the main jEdit bundle instructions. Remember that we now create three JAR files in place of the original single JAR file. While we can rely on the OSGi framework to piece these together into a single application at execution time, this isn't true of the standard Java launcher. We need some way to tell it to include the two additional JAR files on the class path whenever someone executes:

```
$ java -jar jedit.jar
```

Thankfully there is a way, we simply need to add the standard `Class-Path` header to the main JAR file manifest. The `Class-Path` header takes a space-separated list of JAR files, whose locations are relative to the main JAR file. Listing 6.14 has the final main bundle instructions:

Listing 6.14 Final bnd instructions for jEdit main bundle

```
-output: jedit.jar

-include: org/gjt/sp/jedit/jedit.manifest
Class-Path: jedit-thirdparty.jar jedit-engine.jar

Bundle-Name: jEdit
Bundle-SymbolicName: org.gjt.sp.jedit
Bundle-Version: 4.2

Export-Package:\
 org.gjt.sp.jedit;version="4.2"

Private-Package:\
 org.gjt.sp.jedit.proto.*

-versionpolicy: [${version;==;${@}},${version;+;${@}})
-removeheaders: Private-Package,Tool,Bnd-LastModified

Bundle-Activator: org.gjt.sp.jedit.Activator
```

Update your three bnd files as shown above and rebuild or if you want a shortcut use:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
$ cd ..

$ ant jEdit.patch dist
```

Congratulations, you've just successfully separated jEdit into three JAR files that work with or without OSGi!

```
$ java -Djedit.home=jEdit -jar launcher.jar jEdit                    #A

$ java -jar jEdit/jedit.jar                                          #B
```

#A launch jEdit OSGi
#B launch jEdit classic

As this example hopefully demonstrates, once you have an application working in OSGi it doesn't take much effort to start slicing it up into smaller, more modularized bundles. But is this all we can do with jEdit on OSGi, just keep slicing it into smaller and smaller pieces?

6.2.3 Loosening things up

So far we've focused on using the first two layers of OSGi: modularity and lifecycle. There is another layer we have not yet used in this chapter: services. The service layer is different from the first two layers in that it can be very hard to tell when or where you should use it, especially when migrating an existing application to OSGi. Often people decide not to use services at all in new bundles, instead relying on sharing packages to find implementations. But as we saw in chapter 4, services make your application more flexible and help reduce the coupling between bundles. The good news is you can decide to use services at any time, but how will you know when the time is right?

There are many ways to share different implementations inside a Java application. You might construct instances directly, call a factory method, or perhaps apply some form of dependency injection. When you first move an application to OSGi you'll probably decide to use the same tried and tested approach as you did before, except now some of the packages will come from other bundles. But as we saw in chapter 4, these approaches have certain limitations compared to OSGi services. Services in OSGi are extremely dynamic, support rich metadata, and promote loose coupling between the consumer and provider.

If you expect to continue to use your application outside of OSGi, for example as a classic Java application, you might be worried about using the service layer in case it ties you to the OSGi runtime. No problem! You can get the benefits of services without being tied to OSGi, by using component-based dependency injection. Chapter 10 introduces a number of component models that transparently support services without forcing you to depend on the OSGi API. If you already use dependency injection then moving to these component models is straightforward, sometimes only a matter of reconfiguring the dependency bindings in your original application. If you're itching to try out these component models, feel free to skip ahead to Chapter 10. But make sure to come back and read the intervening chapters; they

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

will be an invaluable guide when it comes to managing, testing, and debugging your new OSGi application.

Let's get back to discussing services. Where might we use services in jEdit? Well jEdit has its own home-grown plugin framework for developers to contribute all sorts of gadgets, tools, and widgets to the GUI. In addition, jEdit uses its own custom class loader `org.gjt.sp.jedit.JARClassLoader` to allow hot deployment and removal of jEdit plugins. Plugins hook back into jEdit by accessing implementation classes and calling static methods, such as `jEdit.getSettingsDirectory()`. While these static method calls are convenient, they make it hard to mock out (or replace) dependencies for testing purposes.

Instead of relying on static methods, we could change jEdit to use dependency injection. Plugins would have their dependencies injected, rather than call jEdit directly. Once we replace the static methods calls with dependency injection, it is just another step to replace the static bindings with dynamic OSGi services (10.?). This also simplifies unit testing, as we can swap out the real bindings and put in stubbed or scripted test implementations. Unfortunately, refactoring jEdit to use dependency injection throughout is outside of the scope of this book, but chapter 10 will provide you with a general guide. With this in mind, is there a smaller task that would help bridge the gap between OSGi bundles and jEdit plugins and make it easier to use services?

We could consider replacing the jEdit plugin framework with OSGi, much like Eclipse replaced its original plugin framework. To do this we would have to take the `JARClassLoader` and `PluginJAR` classes and extract a common API we could then re-implement using OSGi, as shown in Figure 6.11. We would use the original jEdit plugin code when running in classic Java mode and the smaller OSGi mapping layer when running on an OSGi framework.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

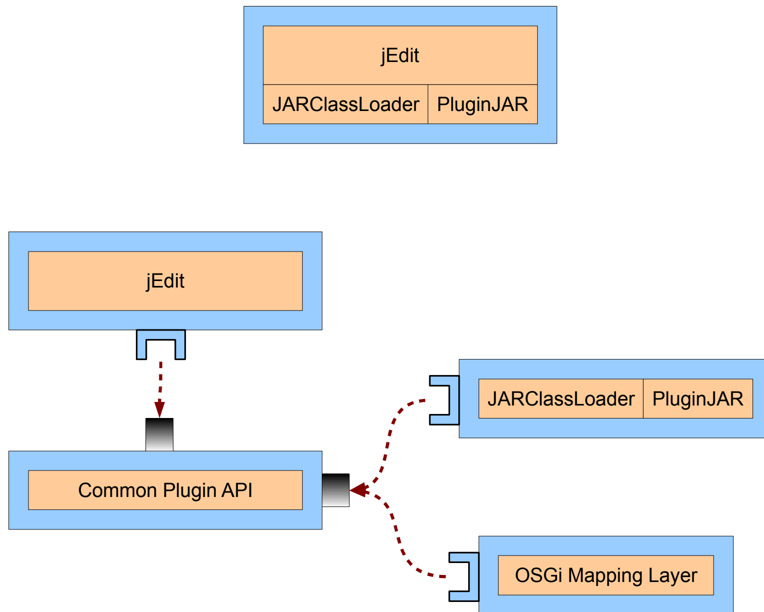


Figure 6.11 Extracting a common jEdit plugin API

Extracting the common plugin API is left as an interesting exercise for the reader; one wrinkle is jEdit assumes plugins are located on a filesystem, whereas OSGi supports bundles installed from opaque input streams. The new plugin API would need methods to iterate over and query JAR file entries to avoid having to know where the plugin was located. These methods would map to the resource entry methods on the OSGi `Bundle` interface.

How about being able to register OSGi bundles as jEdit plugins? This would be a stepping stone to using services, because we need a bundle context to access OSGi services. The main jEdit class provides two static methods to add and remove plugin JAR files:

```
public static void addPluginJAR(String path);

public static void removePluginJAR(PluginJAR jar, boolean exit);
```

Following the extender pattern introduced in section 3.4, let's use a bundle tracker to look for potential jEdit plugins. The code in Listing 6.15 uses a tracker to add and remove jEdit plugin bundles as they come and go. It identifies jEdit plugins by looking for a file called "actions.xml" at the bundle root #1. Because the jEdit API only accepts path-based plugins, it ignores bundles whose locations don't map to a file #4. To remove a plugin bundle it uses another jEdit method to map the location back to the installed `PluginJAR` instance #2. The last piece of the puzzle is to only start the bundle tracker when jEdit is ready to accept new

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

plugins. If you look at the jEdit startup code you might notice one of the last things it does in `finishStartup()` is send out the initial "EditorStarted" message on the EditBus (jEdit's event notification mechanism). So we simply register a one-shot component that listens for any message event, deregisters itself, and starts the bundle tracker #3.

Listing 6.15 Using the extender pattern to install jEdit plugins

```
package org.foo.jedit.extender;

import java.io.File;
import org.gjt.sp.jedit.*;
import org.osgi.framework.*;

public class Activator implements BundleActivator {

  BundleTracker pluginTracker;

  public void start(final BundleContext ctx) {
    pluginTracker = new BundleTracker(ctx) {

      public void addedBundle(Bundle bundle) {
        String path = getBundlePath(bundle);
        if (path != null && bundle.getResource("actions.xml") != null) { #1
          jEdit.addPluginJAR(path);
        }
      }

      public void removedBundle(Bundle bundle) {
        String path = getBundlePath(bundle);
        if (path != null) {
          PluginJAR jar = jEdit.getPluginJAR(path);                      #2
          if (jar != null) {
            jEdit.removePluginJAR(jar, false);
          }
        }
      }
    };

    EditBus.addToBus(new EBComponent() {
      public void handleMessage(EBMessage message) {
        EditBus.removeFromBus(this);
        pluginTracker.open();                                            #3
      }
    });
  }

  public void stop(BundleContext ctx) {
    pluginTracker.close();
    pluginTracker = null;
  }

  static String getBundlePath(Bundle bundle) {                           #4
    String location = bundle.getLocation().trim();
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    File jar;
    if (location.startsWith("file:")) {
      jar = new File(location.substring(5));
    } else {
      jar = new File(location);
    }

    if (jar.isFile()) {
      return jar.getAbsolutePath();
    }

    return null;
  }
}
```

Let's see this extender in action!

```
$ cd chapter06/jEdit-example

$ ant jEdit.patch dist

$ java -Djedit.home=jEdit -jar launcher.jar jEdit

-> install file:test/Calculator.jar
```

Look under the "Plugins" menu, there should be no plugins available. Now start the calculator bundle that you just installed:

```
-> start 9
```

You should now see the calculator under the "Plugins" menu. Selecting this item should bring up the window shown in Figure 6.12. If you stop the calculator bundle this window will immediately disappear, and the "Plugins" menu will once again show no available plugins.

```
-> stop 9
```

Cool, our extender successfully bridges the gap between OSGi bundles and jEdit plugins! We can now use existing OSGi management agents, such as the Apache Felix web-console [ref], to manage jEdit plugins. This small example hopefully shows you how standards like OSGi can make it much easier to re-use and assemble existing pieces into new applications.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

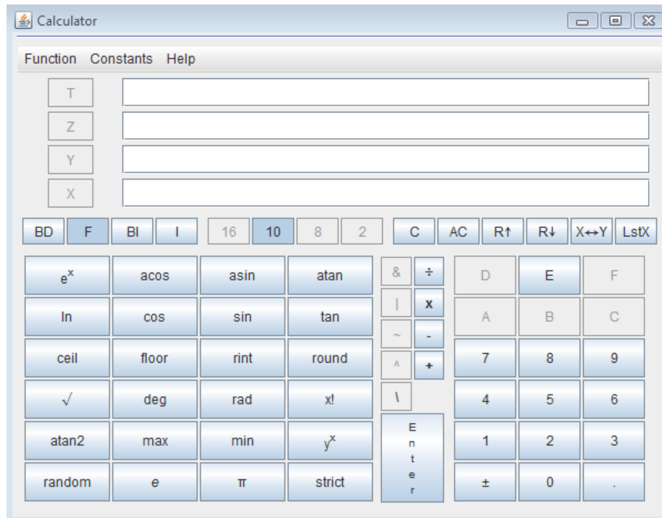


Figure 6.12 jEdit Calculator plugin

So are you eager to start moving your application to OSGi? Wait, not so fast! We have one last topic to discuss before we close out this chapter, and it's something you should keep asking yourself when modularizing applications: is this bundle adding any value?

6.2.4 To bundle or not to bundle?

There are times when you should take a step back and think, do I really need another bundle? The more bundles you create, the more work is required during build, test, and management in general. Creating a bundle for every individual package would obviously be overkill, while putting your entire application inside a single bundle means you're missing out on modularity. Some number of bundles in between is best, but where is the sweet spot?

One way to tell is to measure the benefit introduced by each bundle. If you find you're always upgrading a set of bundles at the same time and you never install them individually, then keeping them as separate bundles is not bringing much benefit. You could also look at how your current choice affects developers. If a bundle layout helps developers work in parallel or enforces separation between components, then it is worth keeping. But if a bundle is getting in the way of development, perhaps for legacy class loader reasons, then you should consider removing it; either by merging it with an existing bundle or by making it available via boot delegation (we briefly discussed this option at the start of 6.2.1). Consider our jEdit example, have we reached the right balance of bundles?

A BUNDLE TOO FAR

First let's refresh our memory. Cast your mind back to section 6.2.2 and the import package discussion following listing 6.11. We mentioned an interesting issue caused by placing the top-level package in its own bundle, separate from the rest of the jEdit engine. You can see

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

the problem for yourself by starting the OSGi version of jEdit, opening the "File..." menu, and selecting the "Print..." option. A message box should pop-up (Figure 6.13) describing a failure in a beanshell script.

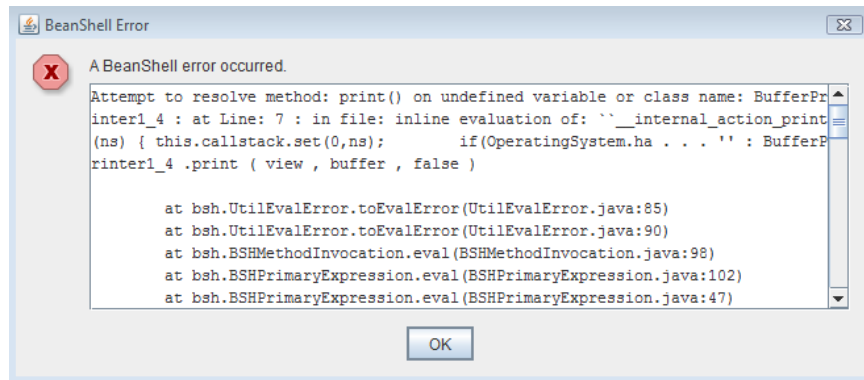


Figure 6.13 Error attempting to print from jEdit

Why did the script fail? The error message suggests a class loading problem. If you scroll down through the stack trace you will notice the last jEdit class before the call to `bsh.BshMethod.invoke()` is `org.gjt.sp.jedit.BeanShell`. This is a utility class which manages beanshell script execution for jEdit. It is part of the top-level jEdit package loaded by the main bundle class loader and it configures the beanshell engine to use a special instance of `JARClassloader` (previously discussed in 6.2.3) that delegates to each plugin class loader in turn. This is so beanshell scripts can access any class in the entire jEdit application. If none of the plugin class loaders can see the class then this special class loader delegates to its parent class loader. For a classic Java application this will be the application class loader, which can see all of the jEdit classes on the class path. For our OSGi application, the parent will be the class loader for the main bundle, which can only see the `org.gjt.sp.jedit` and `proto` packages it contains as well as any packages it explicitly imports. One thing we know it can't see is the `BufferPrinter1_4` class.

Who owns the `BufferPrinter1_4` class? It is part of the `org.gjt.sp.jedit.print` package, belonging to the jEdit engine bundle. We could check the manifest to make sure this package is being exported as expected, but if you're using the instructions from Listing 6.13 then it will be. It is being exported from the engine bundle, but is it being imported by the main bundle? Without an import this package won't be visible. Let's avoid cracking open the JAR file and instead use bnd to see the list of imports:

```
$ java -jar ../../lib/bnd-0.0.355.jar print -impexp jEdit/jedit.jar      #A
```
#A print imported and exported packages

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Aha! The main bundle manifest contains no mention of the `org.gjt.sp.jedit.print` package, which explains why the `BufferPrinter1_4` class wasn't found and the script failed. The last remaining question before we try to fix this issue: why didn't bnd pick up the reference to the `org.gjt.sp.jedit.print` package? Remember that bnd works primarily on byte code, not source code; it won't pick up packages referenced in scripts, arbitrary strings, or runtime configuration files. The only reference to this package was in a beanshell script, which was not analyzed by the bnd tool.

We now have all the answers as to why the script failed, but how should we solve it? Well, bnd does support adding custom analyzers to process additional content, so we could write our own beanshell analyzer for bnd. But what if writing such an analyzer is outside our expertise, can we instead fix the class loading problem at execution time? There are two approaches to solving this type of class loading issue:

1. Attempt to use a different class loader to load the class.
2. Add the necessary imports to the bundle doing the loading.

The first approach is only possible when the library provides some way of passing in the class loader or when it uses the Thread Context Class Loader (TCCL) to load classes. (You can read more about the TCCL in chapter 9.) The beanshell library does provide a method to set the class loader, but jEdit is already using it to pass in the special class loader that provides access to all currently installed jEdit plugins. Rather than mess around with jEdit's internal `JARClassLoader` code and potentially break the jEdit plugin framework, we shall take approach two and simply add the missing imports to the main bundle. This has the least impact on existing jEdit code, all we're doing is updating the OSGi part of the manifest.

We know we need to import the `org.gjt.sp.jedit.print` package, but what else might we need? To make absolutely sure we would need to run through a range of tests exercising the whole of the jEdit GUI. While this testing could be automated to save time, let's instead try the suggestion from the end of section 6.1.3 and allow the main jEdit bundle to import any package on-demand:

```
DynamicImport-Package: *
```

Add this to the "jedit-main.bnd" instruction file and rebuild one more time. You can now open the print dialog without getting the error message. The application will also continue to work even if you use a more restrictive dynamic import, such as

```
DynamicImport-Package: org.gjt.sp.*
```

Why does this work? Well, rather than say upfront what we import, we leave it open to whatever load requests come through the main bundle class loader. As long as another bundle exports the package and it matches the given wildcard, we will be able to see it. But is this really the right solution? Merging the main and engine bundles back together would

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

solve the beanshell problem without the need for dynamic imports. We already know these bundles are tightly coupled, keeping them apart is just causing us further trouble. In fact, this is a good example of where introducing more bundles does not make sense. OSGi is not a golden hammer and it won't magically make code more modular.

In short, if you're getting class loading errors or are sharing lots of packages between bundles, that could be a sign to start merging them back together. You might even decide to fall back to classic Java class loading by putting really troublesome JAR files on the application class path and adding their packages to the `org.osgi.framework.bootdelegation` property (section 9.2). You won't be able to use multiple versions or dynamically deploy them, but if it avoids tangled class loading problems and helps keep your developers sane then it's a fair trade. You can often achieve more by just concentrating on modularizing your own code. Leave complex third-party library JAR files on the application class path until you know how to turn them into bundles or until an OSGi compatible release is available. Not everything has to be a bundle. As we often say in this book: you can decide how much OSGi you want to use, it is definitely *not* an all-or-nothing approach!

6.3 Summary

At the start of this chapter we showed you how to turn a JAR file into a bundle (abracadabra!) followed by a lengthier discussion about turning a complete application into several bundles. Making decisions about where to slice an application into bundles comes with experience, but you can use existing knowledge about your application's design to drive this process. Feedback from developers along with other project measurements can indicate which of the new bundles bring the most bang for the buck and help focus future development. We also learned there is often a sweet spot in the number of bundles, where you get the most value for the least amount of management cost.

But what is actually involved in managing bundles? Once you've split your application into many independent parts, how do you keep everything consistent and how do you upgrade your application without bringing everything down? The next chapter will discuss this and more, as we look at managing real-world OSGi applications.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

7 Managing Bundles and Applications

By now you should be familiar with the mechanisms to create, deploy and interact with bundles. We have seen: how to use modularity to improve the cohesiveness of your application code; how to use lifecycle to bring dynamic installation and update to application environments; and how to use services to decouple our modules via interface-based programming techniques. We have also shown you some of the advanced features of the OSGi modularity toolkit and demonstrated how to begin migrating a classic Java application to an OSGi environment.

With the OSGi Service Platform, you can create loosely coupled and highly cohesive bundles. You can compose your bundles in many different ways. In a sense, your deployed set of bundles becomes your application's configuration. As such, the task of managing your bundles is one of the most important skills you will need to fully master OSGi. In this chapter, we will explore the four different aspects of application management:

- Evolving applications using versioning policies to avoid inconsistent or incompatible configurations,
- Configuring applications using the `ConfigurationAdmin`, `MetaTypeService`, and `Preferences` services,
- Deploying applications using the OSGi Bundle Repository or the `DeploymentAdmin`, and
- Ordering bundle activation using the `StartLevel` service.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

With these tools you will be better equipped to build, deploy, and configure sophisticated OSGi-based applications. Let's start by looking at versioning.

7.1 Versioning packages and bundles

From what we've learned so far, we know versioning is a core part of any OSGi application. Both bundles and their exports have versions. When the framework resolves bundle dependencies, it must take these versions into account. In this section, we'll discuss the recommended policy for versioning these artifacts and discuss advantages and disadvantages of different versioning strategies. To get things started, let's provide some motivation for OSGi's approach to versioning.

7.1.1 Meaningful versioning

In traditional Java programming, versioning is an afterthought. OSGi, on the other hand, treats versioning as a first-class citizen, which makes it easier to handle versioning in a meaningful way. This emphasis on versioning makes the need for a proper versioning strategy very important in order to get and keep everything working correctly.

You must be thinking, “Hey! I already version my JAR files!” Tools like Maven and Ivy already allow us to specify versions for JAR files and declare dependencies on those versions. We discussed these sorts of module-level dependencies in chapter [ref]. We also mentioned a number of reasons why they are bad for the health of your projects. In short, they are brittle when it comes to really expressing fine-grained dependencies between units of code.

As it turns out, not only do module-level dependencies have drawbacks, but so does module-level versioning. Such a model is too simple and forces all packages inside of a JAR file to be versioned in lockstep with the other packages. Let's look at some of these reasons in more detail.

Module versioning is opaque

Consider a case where you bundle some related packages together and assign a version number to the resulting JAR file. Later you may need to alter some code within one of the contained packages; such a change may be the result of a bug fix or a change to the API contract. This new JAR file needs a new version number associated with it.

With a single version number for all the packages, it is now up to upstream users of the JAR file to decide whether the change warrants an update. Since the only information they have is the module-level version number change, it is often a stab in the dark as to whether the updated functionality is required for their application. One reason for this is that upstream users do not always use all functionality provided by a JAR file and are actually only dependent on some subset. Depending on the subset being used, it is possible nothing of importance has changed for them.

A counter argument is if the bundle is highly cohesive, then it makes no sense to update a single package without it's siblings. While this is true, it is not uncommon for JAR files to be less than optimally cohesive. In fact, OSGi already caters for this situation with “uses”

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

constraints, which we mentioned in chapter [ref]. These ensure the cohesiveness of packages is maintained by capturing internal package dependencies. This means upstream users aren't forced to depend on anything more than the API-level contract of the exported packages.

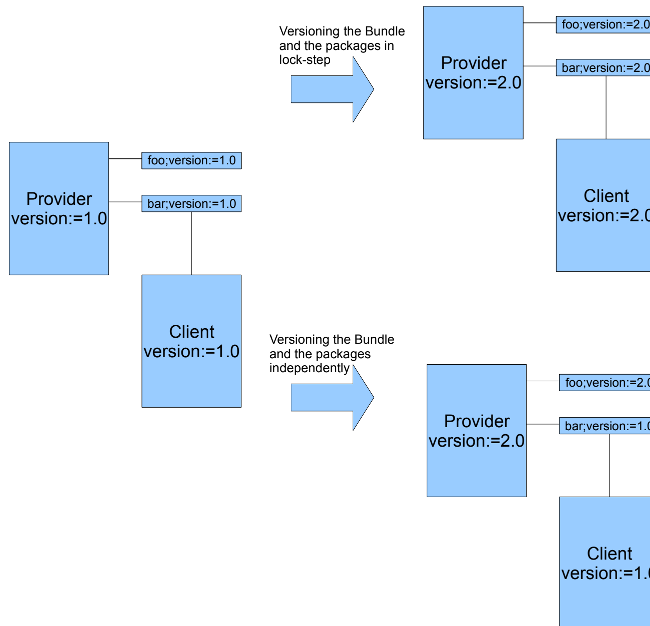


Figure 7.1 Versioning packages independently

Luckily, in OSGi, we can version our packages either independently or in lockstep with the bundle as shown in Figure 7.1. The OSGi approach of package-level versioning and dependencies leads to less churn in the development lifecycle. Less churn implies less risk, since existing modules are better understood than updated modules, which can introduce unexpected behavior into a complex system. This concept is extremely powerful and removes a lot of the pain from assembling applications out of independent JAR files, since we can make better informed decisions about when and what to update in our applications.

Multiple versions in the same JVM

Package-level versioning is also helpful when it comes to running different versions side-by-side. Java doesn't support this by default, but OSGi does. In many cases, this seemingly unimportant feature frees you from worrying about backward compatibility or changes to artifacts outside your control. Your bundles can continue to use the versions of packages with which they are compatible, since your entire application no longer has to agree on a single version to place on the class path.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

There is a price to pay for this flexibility. Versioning must be done as a core task throughout the development process, not as an afterthought. Versioning packages and maintaining a versioning policy is a lot of work. One easy way to reduce the amount of work is to have less to version. In OSGi, you have the option to not expose the implementation packages of a bundle (assuming that nobody else outside the bundle needs them). As a consequence, the simplest option you have is to not export packages to avoid the need to version them. When you need to export packages, however, then you need to version them. Let's look more closely at how we can implement a versioning policy for packages in OSGi.

7.1.2 Package versioning

Let's consider a package named `org.foo` with a version of `1.0.0.r4711` provided by a bundle called `foo` which is itself at version `1.0.0`. Its manifest would look like this:

```
Bundle-SymbolicName: foo
Bundle-Version: 1.0.0
Export-Package: org.foo;version:="1.0.0.r4711"
```

As we mentioned previously, the OSGi specification does not define a versioning policy, which means you can use any scheme that makes sense to you. However, the OSGi specification does recommend the following policy behind version number component changes:

- Major number change – signifies an incompatible update.
- Minor number change – signifies a backward compatible update.
- Micro number change – signifies an internal update (e.g., a bug fix or performance improvement).
- Qualifier change – signifies a trivial internal change with "outward" noticeable difference, but nonetheless is a new artifact (e.g., line number refactoring).

This is a very common version compatibility policy. Why? Since versions are important for the consumer to specify what is needed, this policy makes it possible to easily express a floor and a ceiling version in between which all versions are allowed. As we saw in chapter [ref], a version range is expressed as a pair of version numbers inside brackets or parentheses. This follows mathematical interval notation, where a square bracket signifies an inclusive value and a parenthesis signifies an exclusive value. As an example, consider a typical definition of a package import,

```
Import-Package: org.foo;version:="[1.0,2.0)"
```

The `org.foo` package is imported in version `1.0.0` up to, but excluding `2.0.0`. This makes sense if the recommended version policy is being used, since it would include all backward compatible versions and exclude all non-backward compatible versions, which a change in the major number would signify. Being able to specify such ranges is very useful since the import can be satisfied by a wider range of exports. This scheme only works if producers and consumers operate with a shared understanding of the kind of compatibility being expressed by a given version number.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

DOWNSIDES AND PITFALLS

The recommended OSGi versioning policy sounds good and it has been used successfully by many projects. But new users should still take care due to a subtlety related to the usage of Java interfaces, which is related to whether an interface is being used or implemented.

The difference seems trivial, but becomes very important in the context of versioning. Consider the following `1.0.0` version of the `Foo` interface:

```
public interface Foo {
  public Bar getBar();
}
```

What happens if we change this simple interface? For example, by adding a method:

```
public interface Foo {
  public Bar getBar();
  public void setBar();
}
```

The question to ponder is whether this change should cause a major or minor version number increase? It depends on whether the interface is intended to be implemented or used by the consumer. In the former case, the addition of the method is a binary incompatible change to the interface and should cause a major version number increase to `2.0.0`. In the latter case, a minor version number increase to `1.1.0` is sufficient because method addition is a backward compatible update to the interface. Figure 7.2 shows the situation.

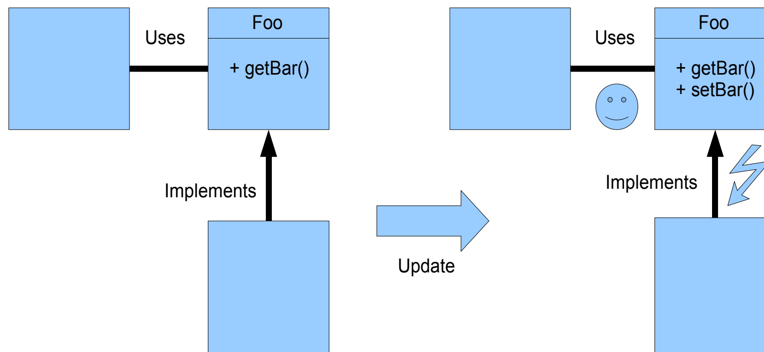


Figure 7.2: Difference between uses and implements in updates

If you are in control of all the bundles, you could define a policy to ensure that method addition always causes a major version number change, which allows all consumers of a package to use a `[1.0,2.0)` version range. In reality, you are unlikely to be in control of all the bundles. Furthermore, such a drastic policy would limit the reusability of your bundles, because consumers only using your interfaces would have no way to express they are fine with an added method.

A REFINED APPROACH

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The best strategy devised so far is to shift the burden to the consumer. This is pretty straightforward and requires implementers to specify a version range of `[1.0,1.1)`, while users can use the broader version range of `[1.0,2.0)` as shown in Figure 7.3.

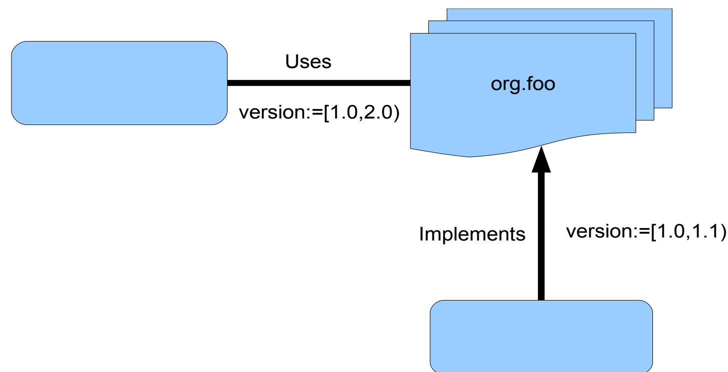


Figure 7.3: Best practice for interface users and implementers

Another important aspect about versioning is to be consistent. You don't want to define your versioning policy on a bundle by bundle basis. So, whether your follow the recommended approach or not, you should at least try to use the same global policy.

This gives us a fairly good understanding of versioning policy for packages, but what about versioning bundles? We'll explore bundle versioning policies next.

7.1.3 Bundle versioning

Bundles and packages are related through containment: bundles contain packages. Since both bundles and packages have version numbers, what is the relationship between them? We need to adopt a versioning policy to define this relationship. Let's look at that in more detail.

In the simple case, a bundle may contain several related implementation packages all with the same version number. Here it is advisable to make the bundle version mirror the version of the implementation packages. When dealing with a bundle containing packages of different versions, the most consistent versioning policy is to increment the bundle version based on the highest change of a package inside it. For example, if any package has a major number increase, then the major number of the bundle should increase as well. Likewise, if the highest change was to a minor or micro portion of the version. With this policy, it is possible for us to judge the impact of an updated bundle based on its version number. Unfortunately, this might not always make sense, especially if the versions of the individual packages actually represent a well-known product version.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

For example, let's assume we want to create a bundle for the core API of the OSGi framework. In this case, we have several independently versioned packages, but the collection of packages in this bundle has a version number based on the OSGi specification. Figure 7.4 graphically depicts this situation.

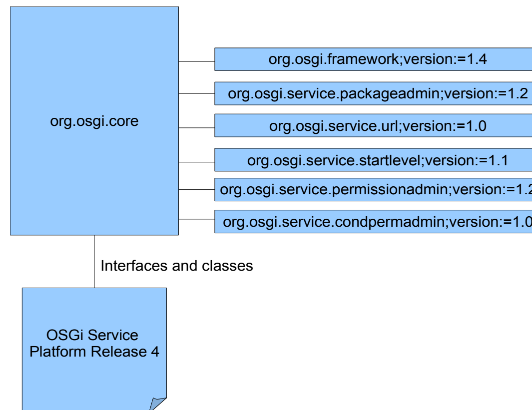


Figure 7.4: The platform implementation contains many sub packages that must evolve in line with the specification, but what is the version of the implementation?

Now the question is, what version should we assign to the `org.osgi.core` bundle? There is no single answer. We could have increased the major number on every major release of the OSGi specification, but this would have indicated a binary incompatible change in at least one of the provided packages which clearly is not the case (as indicated by the individual package versions). Another approach is to keep the version number at 1, indicating no binary incompatible change has happened. We would then need to use the minor number to match the release number of the specification. Since the OSGi specification has also had minor number releases (e.g., 4.1), we would then need to use the micro number for the minor number of the specification.

Unfortunately, this wouldn't be exactly what we want, since there have been updates in the minor numbers of the contained packages. To make matters worse, if we ever needed to update the bundle for a different reason (like a packaging mistake), then we'd need to use the qualifier to express that the bundle has changed. In the specific case of the core OSGi specification, the OSGi Alliance actually makes them available based on the version of the specification (i.e., 4.1.0, 4.2.0, etc.).

The important management task to take away from this section is that versioning is important and should not be left as an afterthought or ignored. There are issues you need to take into account when versioning your packages and bundles. If done correctly, the OSGi concept of versioning is extremely powerful and removes a lot of the pain from assembling applications. To get it correct, you will need to define a versioning policy and enforce it upon

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

all your bundles and exported packages. One possible policy is the one recommended by OSGi which we introduced in this section. With that established, we can now look into another important management task: configuring our bundles.

7.2 Configuring bundles

To make the bundles we create more reusable, we often introduce configuration properties to control their behavior. Recall from chapter [ref] when we introduced our shell example, we used configuration properties to alter its behavior, such as the port on which it listened for client connections. Configuring bundles is an important aspect of using them, so it would be very beneficial if there was a standard way of managing this for bundles. At a minimum, it would be nice if we had:

- A common format for specifying the type of configuration data a given bundle expects.
- A common way to actually set the configuration information for a given bundle.
- A common mechanism for bundles to safely store bundle- and user-related configuration information.

Fortunately for us, the OSGi Alliance defines the following three compendium specifications to help us address these issues:

3. Configuration Admin service – manages key/value pair configuration properties for bundles.
4. Metatype service – allows bundles to describes their configuration properties.
5. Preferences service – provides a place to store bundle- and user-related information.

Even with these specifications to help us, adding bundle configurations to the mix still creates more issues for us to worry about. Configuration data becomes yet another artifact to manage. For example, we have to make sure to consider them when we change the bundles in our systems as well, since configuration data are generally not compatible across bundles or even bundle versions. They are subject to deployment and provisioning just like bundles. In the remainder of this section, we will introduce you to the above configuration-related services and show you how you can manage configurations. We'll start with the Configuration Admin service.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

7.2.1 Configuration Admin service

The Configuration Admin service is an important piece of the deployment of an OSGi Service Platform. It allows you to set the configuration information of deployed bundles. This process involves setting a bundle's configuration data and ensuring it receive that data when it becomes active. What happens is pretty simple. Consider the scenario in Figure 7.5 where a bundle needs an integer port number and a boolean secure property. In this case, you provide these values to the Configuration Admin service and it provides these values to the bundle when it is activated. Using this approach, bundles have a simple, standard way of obtaining configuration data.

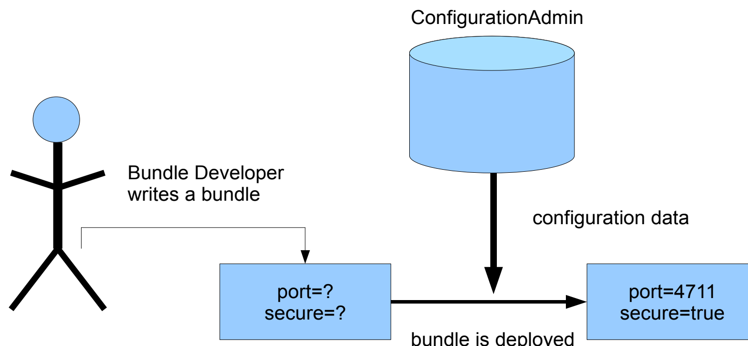


Figure 7.5: An administrator configures a bundle in the framework by interacting with the configuration admin service. This decouples the administrator from having to know the internal workings of the bundle that uses this configuration.

How does this work? The Configuration Admin service maintains a database of `Configuration` objects, each of which has an associated set of name-value pair properties. The Configuration Admin service follows the whiteboard pattern and monitors the service registry for two different "managed" services: `ManagedService` and `ManagedServiceFactory`. If you have a bundle needing configuration data, it must register one of these two services as defined by the Configuration Admin specification. The difference between these two is that a `ManagedService` accepts one configuration to configure a single service, while a `ManagedServiceFactory` accepts any number of configurations and configures a different service instance for each configuration; Figure 7.6 illustrates this difference.

When registering one of these managed services, you need to attach a `service.pid` (service persistent identity) property to it. Each managed `Configuration` object also has a `service.pid` associated with it, which the Configuration Admin service uses as a key to match configuration data to the bundle needing it.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

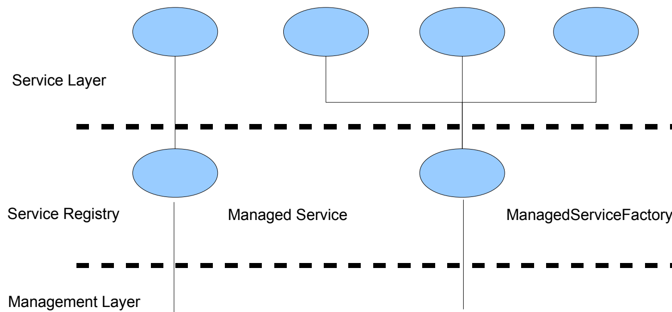


Figure 7.6: Differentiation of ManagedService and ManagedServiceFactory

WHAT IS A PID?

In a nutshell, each registered service can have a persistent identity or PID associated with it by specifying it in its service property dictionary when registering the service. If you specify a `service.pid` property, it must be unique for each service. Its purpose is to uniquely and persistently identify a given service, which allows the Configuration Admin service to use it as a primary key for bundles needing configuration data. This means the Configuration Admin service requires the use of a PID with `ManagedService` and `ManagedServiceFactory` service registrations. As a convention, PIDs starting with a bundle identifier and a dot are reserved for the bundle associated with that identifier. For example, the PID `42.4711` belongs to the bundle associated with bundle identifier 42. You are free to use other schemes for your PIDs, just make sure they are unique and persistent across bundle activations.

You may have noticed we are dealing with two conceptually different layers when using the Configuration Admin service. On one layer, we have a published `ManagedService` or `ManagedServiceFactory` service. On the other layer, we have a bundle and the services it provides that we actually want to configure. The Configuration Admin service connects these two layers together to deliver configuration data. Of course, the reverse is also possible, and the Configuration Admin service may tell a managed service that its configuration has gone away, which means it needs to stop performing its functionality since it no longer has a valid configuration. The benefit of this approach is you get a really flexible system, where you can configure and control any kind of service or any number of service instances in a common way. Let's look into the details of implementing a managed service next.

IMPLEMENTING A MANAGED SERVICE

Now that we understand the underlying basics of how the Configuration Admin service works by associating configuration data to managed services, let's explore an example. The actual interface we need to implement looks like the following:

```
public interface ManagedService {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
  public void updated(Dictionary properties) throws ConfigurationException;
}
```

Listing 7.1 shows an example `ManagedService` implementation.

Listing 7.1: Example of a managed service

```
public class ManagedServiceExample implements ManagedService {           #1
  private EchoServer m_server = null;

  public synchronized void updated(Dictionary properties)                #2
    throws ConfigurationException {
    if (m_server != null) {                                              #3
      m_server.stop();                                                   #3
      m_server = null;                                                   #3
    }
    if (properties != null) {                                            #4
      String portString = (String) properties.get("port");               #5
      if (portString == null) {                                          #6
        throw new ConfigurationException(null,"Property missing");       #6
      }
      int port;
      try {
        port = Integer.parseInt(portString);                             #7
      } catch (NumberFormatException ex) {                               #7
        throw new ConfigurationException(null, "Not a valid port number");
      }
      try {
        m_server = new EchoServer(port);                                 #8
        m_server.start();                                                #8
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
...
}
```

We implement the `ManagedService` interface at (#1), which has a single `updated()` method that we implement at (#2). The argument to this method is a `Dictionary` containing the configuration properties.

CONFIGURATION PROPERTIES

A configuration dictionary contains a set of properties in a `Dictionary` object. The name or key of a property must always be a `String` object and is not case-sensitive during look up, but will preserve the original case. The values should be of type `String`, `Integer`, `Long`, `Float`, `Double`, `Byte`, `Short`, `Character`, `Boolean`, or the primitive counterparts. Furthermore, they can be arrays or collections of them. For arrays and collections, they must only contain values of the same type.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

In this example, we have a simple “echo” server that listens on a port and sends back whatever it receives. Since it is good practice, we make the port configurable. When we receive a new configuration, we first stop the existing server if there is one at (#3). At (#4), we check if we received a null configuration, which indicates the previous configuration was deleted and there is no new one. If this is the case, then we there is nothing else to do. Otherwise, we get the port number from the dictionary at (#5) and verify its existence at (#6). If it exists, we parse it at (#7) and create and start a new server for the given port at (#8).

A ManagedService is associated with one configuration object. A bundle can register any number of ManagedService services, but each must be identified with its own PID. A ManagedService should be used when configuration is needed for a single entity in the bundle or where the service represents an external entity like a device. Then, for each detected device a ManagedService is published with a PID related to the identity of the device, such as the address or serial number. What about cases where we simply want to configure more than a single entity using the same PID, such as creating multiple instances of the same service with different configurations? We use a ManagedServiceFactory, which we'll explore next.

Implementing a managed service factory

A ManagedServiceFactory should be used when a bundle does not have an internal or external entity associated with the configuration information, but can handle more then one configuration at the same time. Remember, with a ManagedService there is only one configuration namely, the configuration for the specific PID. With a ManagedServiceFactory, there can be any number of configurations for the same factory. Using this approach, you can instantiate a service for each configuration associated with your managed service factory, for example. This way, by simply creating a new configuration for the managed service factory you actually create new service instances. A slightly different use case is related to services representing entities that cannot be identified directly, such as devices on a USB port that can't provide information about their type. Using a ManagedServiceFactory we can define configuration information for each available device attached to the USB port.

How does this work with respect to the PIDs? The trick in this case is that the ManagedServiceFactory is registered with a factory.pid property. This way, the Configuration Admin service can differentiate between a managed factory service and a managed service. For the managed factory service, it assigns a new and unique PID to each created configuration for the factory. The interface to implement looks like this:

```
public interface ManagedServiceFactory{
  public String getName();
  public void updated(String name, Dictionary properties)
    throws ConfigurationException;
  public void deleted(String name);
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Listing 7.2 shows an example where we use a `ManagedServiceFactory` to configure echo services that will read from their configured port and send back whatever they receive along with their name.

Listing 7.2: ManagedServiceFactory example

```
public class ManagedServiceFactoryExample implements
  ManagedServiceFactory {                                             #1
  private final Map<String, EchoServer> m_servers =                   #2
    new HashMap<String, EchoServer>();

  public synchronized void deleted(String pid) {                      #3
    EchoServer server = m_servers.remove(pid);                        #4
    if (server != null) {                                             #4
      server.stop();                                                  #4
    }
  }

  public String getName() {                                           #5
    return getClass().getName();
  }

  public synchronized void updated(String pid, Dictionary properties)
    throws ConfigurationException {
    EchoServer server = m_servers.remove(pid);                        #6
    if (server != null) {
      server.stop();
    }
    if (properties != null) {
      String portString = (String) properties.get("port");
      if (portString == null) {
        throw new ConfigurationException(null, "Property missing");
      }
      int port;
      try {
        port = Integer.parseInt(portString);
      } catch (NumberFormatException ex) {
        throw new ConfigurationException(null,"Not a valid port number");
      }
      try {
        server = new EchoServer(port);
        server.start();
        m_servers.put(pid, server);                                   #7
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
  ...
}
```

This example is not significantly different than the last one. We now implement the `ManagedServiceFactory` interface at (#1). Since we are going to manage a number of servers, we introduce a map to hold them at (#2). The factory interface defines two new

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

methods, `deleted()` and `getName()`, which we implement at (#3) and (#5), respectively. The latter is simply a descriptive name for the factory, while the former notifies our factory that a previously updated configuration has gone away, which results in us stopping the corresponding server at (#4). Notice that the `updated()` method actually has a different signature from the `ManagedService` interface at (#6). It now accepts a PID, which is needed because our managed factory service needs to know the PID for the supplied configuration, which it will correlate with a specific echo server. For each one, we need a PID and a configuration. The rest is similar to what we did for a single server in the `ManagedService` example. The only exception is now we need to add the resulting server instance to our list of servers, which we do at (#7).

This covers the basics about what we need to do to make our bundles configurable, now we need to look into how we actually configure our bundles by creating configurations.

Creating configurations

It is one thing to make our bundles configurable, but we need some way to specify and set the actual property values we want to use to configure them. We need to learn how to create and manage configurations; we use the Configuration Admin service for this. It provides methods to maintain configuration data by means of `Configuration` objects associated with specific configuration targets which can be created, listed, modified, and deleted. The `ConfigurationAdmin` service interface is defined as follows:

```
public interface ConfigurationAdmin{
  public Configuration createFactoryConfiguration(String factoryPid)
    throws IOException;
  public Configuration createFactoryConfiguration(String factoryPid,
    String location) throws IOException;
  public Configuration getConfiguration(String pid, String location)
    throws IOException;
  public Configuration getConfiguration(String pid) throws IOException;
  public Configuration[] listConfigurations(String filter) throws
    IOException, InvalidSyntaxException;
}
```

`Configuration` objects are represented by the following interface:

```
public interface Configuration{
  public String getPid();
  public Dictionary getProperties();
  public void update(Dictionary properties) throws IOException;
  public void delete() throws IOException;
  public String getFactoryPid();
  public void update() throws IOException;
  public void setBundleLocation(String location);
  public String getBundleLocation();
}
```

To illustrate how these all fit together, we can continue to improve our shell example by creating a new command to manage configurations. Have a look at Listing 7.3.

Listing 7.3: ConfigurationAdmin service shell command

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
public class ConfigAdminCommand extends BasicCommand {
  public void exec(String args, PrintStream out, PrintStream err)
    throws Exception {
    args=args.trim();
    if (args.startsWith("list")) {                                #1
      listConfigurations(args.substring("list".length()).trim(),
        out);
    } else if (args.startsWith("add-cfg")) {                      #2
      addConfiguration(args.substring("add-cfg".length()).trim());
    } else if (args.startsWith("remove-cfg")) {                   #3
      removeConfiguration(args.substring(
        "remove-cfg".length()).trim());
    } else if (args.startsWith("add-factory-cfg")) {              #4
      addFactoryConfiguration(args.substring("add-factory-
      cfg".length()).trim());
    } else if (args.startsWith("remove-factory-cfg")) {           #5
      removeFactoryConfiguration(args.substring(
        "remove-factory-cfg".length()).trim());
    }
  }
```

In this example, we create a "cm" command that accepts five different subcommands namely: list, add-cfg, remove-cfg, add-factory-cfg, and remove-factory-cfg. The above code is largely just responsible for delegating to private methods to perform the functionality of the subcommands at (#1), (#2), (#3), (#4), and (#5), respectively. Listing 7.4 shows how "cm list" lists available configurations.

Listing 7.4 Implementation of the "cm list" subcommand

```
private void listConfigurations(String filter, PrintStream out)
  throws IOException, InvalidSyntaxException {
  Configuration[] configurations = admin().listConfigurations(          #1
    ((filter.length() == 0) ? null : filter));
  if (configurations != null) {
    for (Configuration configuration : configurations) {                #2
      Dictionary properties = configuration.getProperties();            #2
      for (Enumeration e = properties.keys(); e.hasMoreElements();) {   #2
        Object key = e.nextElement();                                   #2
        out.println(key + "=" + properties.get(key));                   #2
      }
      out.println();
    }
  }
  ...
}
```

We get the `ConfigurationAdmin` service and use it's `listConfigurations()` method to get the configuration objects at (#1). We can optionally specify an LDAP filter to limit which configurations are returned; specifying no filter results in all configurations. In either case, an array of `Configuration` objects is returned, which are the holders of the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

actual configuration properties. At (#2) we print the configuration properties using the getProperties() method on the Configuration object to retrieve them.

We can use the "add-cfg" subcommand to create new configuration objects. The subcommand accepts the PID of the ManagedService and the configuration properties as a whitespace delimited list of name-value pairs, where the name and value are separated by an equals sign. The actual implementation is as follows:

```
private void addConfiguration(String args) {
  String pid = args.substring(0, args.indexOf(" ")).trim();
  Configuration conf = admin.getConfiguration(pid, null);            #1
  createConfiguration(args.substring(pid.length()).trim(), pid, conf);   #2
}
```

To create a Configuration object, we call getConfiguration() on ConfigurationAdmin at (#1). This method creates the configuration object on the first call and returns the same object on subsequent calls. We initialize the new configuration with a call to the private method createConfiguration() at (#2), which is defined in Listing 7.5.

Listing 7.5 Private method to initialize configuration objects

```
private void createConfiguration(
  String args, String pid, Configuration conf) throws IOException {
  conf.setBundleLocation(null);                                      #1
  Dictionary dict = conf.getProperties();                            #2
  if (dict == null) {
    dict = new Properties();
  }
  StringTokenizer tok = new StringTokenizer(args, " ");              #3
  while (tok.hasMoreTokens()) {                                      #3
    String[] entry = tok.nextToken().split("=");                     #3
    dict.put(entry[0], entry[1]);                                    #3
  }
  conf.update(dict);                                                 #4
}
```

This sets the Configuration object's bundle location to null at (#1), which means it is not currently associated with any bundle. We finish initializing the new configuration by getting any existing properties at (#2), parsing the specified properties and merging them with existing properties at (#3), and finally updating the configuration at (#4). Since we handle existing properties, the "add-cfg" subcommand can be used to create and modify configurations.

CONFIGURATION AND LOCATION BINDING

When a Configuration object is created using either getConfiguration() or createFactoryConfiguration(), it becomes bound to the location of the calling bundle. This location is obtained via the calling bundle's getLocation() method. Location binding is a security feature to assure only management bundles can modify configuration data and other bundles can only modify their own configuration data. If the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

bundle location of a configuration for a given PID is set to `null` (as we did in Listing 7.5), then the Configuration Admin service will bind the first bundle registering a managed service with the given PID to this configuration. Once the bundle location is set, then configurations for the given PID are only delivered to the bundle with that location. When this dynamically bound bundle is subsequently uninstalled, the location is set to `null` again automatically so it can be bound again later.

The "remove-cfg" subcommand can be used to remove configuration objects. The implementation of this subcommand is much simpler:

```
private void removeConfiguration(String pid) {
  Configuration conf = admin.getConfiguration(pid);                    #1
  conf.delete();                                                       #2
}
```

The subcommand accepts a PID which we use to get the `Configuration` object from the `ConfigurationAdmin` service at (#1). Once we have the `Configuration` object, we call `delete()` on it at (#2).

The "add-factory-cfg" subcommand creates a configuration object for a managed factory service. It is implemented as follows:

```
private void addFactoryConfiguration(String args) {
  String pid = args.substring(0, args.indexOf(" ")).trim();
  Configuration conf = admin.createFactoryConfiguration(pid, null);    #1
  createConfiguration(args.substring(pid.length()).trim(), pid, conf);
}
```

It accepts the PID of the managed factory service and the configuration properties as a whitespace delimited list of name-value pairs. It is very similar to the "add-cfg" subcommand, except we use `ConfigurationAdmin.createFactoryConfiguration()` to create a new `Configuration` object for the factory at (#1). This always creates a new `Configuration` object for the factory service (unlike `getConfiguration()`, which only creates one the first time for a given PID).

The "remove-factory-cfg" subcommand allows us to remove a factory configuration, it is implemented as follows:

```
private void removeFactoryConfiguration(String pid) {
  Configuration[] configurations = admin.listConfigurations(
    "(service.pid=" + pid + ")");                                      #1
  configurations[0].delete();                                          #2
}
```

The subcommand accepts a PID which we use to find the associated configuration using `listConfigurations()` with an appropriated filter at (#1). Once we have it, we call `delete()` on it at (#2) as before.

To experiment with this new command, go into the `code/chapter07/shell-example/` directory of the companion code. Type `ant` to build the example and `java -jar launcher.jar bundles` to execute it. To interact with the shell use `telnet localhost`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

7070. For this example, we use the Apache Felix Configuration Admin implementation[2]. Listing 7.6 shows a session using the “cm” command.

Listing 7.6 Configuration Admin Command Example

```
-> cm add-cfg foo first=bar second=baz
-> cm add-factory-cfg bar first=foo
-> cm list
service.pid=foo
second=baz
first=bar

service.pid=bar.b1925070-fdc8-4009-b961-3a8853ab2854
service.factoryPid=bar
first=foo

-> cm remove-cfg foo
-> cm remove-factory-cfg bar.b1925070-fdc8-4009-b961-3a8853ab2854
-> cm list
->
```

That finishes our quick tour of the Configuration Admin service. You should now be able to use Configuration Admin to create externally configurable bundles, instantiate services using configurations, and manage configurations. But wait, how do we know what kind of data our configurable bundles accept? All we've said so far is that managed services are configured with simple name-value pairs. Sometimes that may suffice, but often you might want to tell other bundles or entities, like a user, about the structure of your bundle's configuration data. The Metatype service, which we'll introduce next, allows you to define your own meta types and associate them with your bundles and services.

7.2.2 Meta type service

Assume for a moment, that we are deploying a new bundle for the first time into a framework that has our Configuration Admin shell command available. If this new bundle provides some services which are configurable, then we can use our shell command to configure it, right? Unfortunately, since this bundle is new to us, we have no idea which properties it accepts, nor which ones are mandatory. In this kind of scenario, it would certainly be helpful if the bundle could convey to us what a valid configuration look likes.

The OSGi standard Meta Type service makes this possible. It aggregates meta types (i.e., descriptions of types) contributed by bundles and allows others to look up these definitions. Using this service allows us to introspect what a managed service accepts as valid configuration and also validate configurations against these schema, which are subject to the same update and versioning mechanisms as the bundles that provide them.

As you can see in Figure 7.7, there are two ways to provide meta type information about for your managed services:

[2] http://felix.apache.org/site/apache-felix-configuration-admin-service.html

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

1. A bundle can contain XML resources in its `OSGI-INF/metatype` directory which will then be picked-up by the Meta Type service using the extender pattern or
2. A managed service can implement a second interface, called `MetaTypeProvider`.

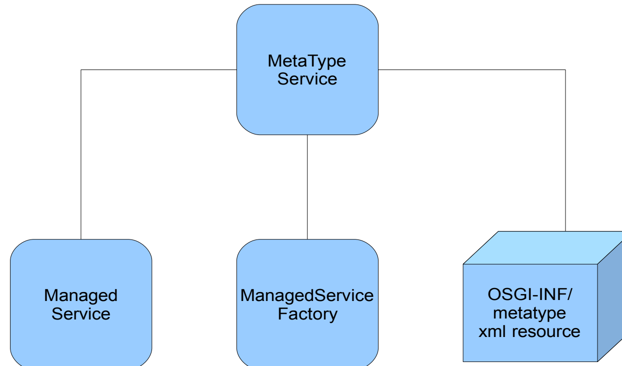


Figure 7.7: Meta Type service overview

If for some reason a bundle does both, then only the XML resources are considered and the `MetaTypeProvider` service is ignored.

From a client perspective, the Meta Type service defines a dynamic typing system for properties. This allows us, for example, to construct reasonable user interfaces dynamically. The service itself provides unified access to the meta type information provided by deployed bundles. A client can request `MetaTypeInformation` associated with a given bundle which in turn provides a list of `ObjectClassDefinition` objects for this bundle. An object class contains descriptive information and a set of name-value pairs. Listing 7.7 shows what this looks like for the example Echo Server.

Listing 7.7: Example Meta Type XML resource file

```
<?xml version="1.0" encoding="UTF-8"?>
<MetaData xmlns="http://www.osgi.org/xmlns/metatype/v1.0.0">
  <OCD name="EchoServer" id="4.7.1.1" description="Echo Server Config">  #1
    <AD name="port" id="4.7.1.1.1" type="Integer"                         #2
     description="The port the Echo Server listens on"/>
  </OCD>
 <Designate pid="org.foo.managed.service">
  <Object ocdref="4.7.1.1"/>
 </Designate>
</MetaData>
```

Don't let this somewhat obtuse XML fool you. It is actually quite simple. We first define an `ObjectClassDefinition` (OCD) at (#1), called `EchoServer`, with a unique identifier of `4.7.1.1` (if you have a matching LDAP/X.500 object class OID, then you can use that one; otherwise, any other reasonably unique name that follows the same grammar as the LDAP/X.500 OID) and a human readable description. At (#2) we specify an attribute

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

definition (AD) to describe the configuration properties the Echo Server needs. In this case, there is only one, called port. Notice, the Designate element? This is where you make the link between the type (i.e., the OCD) and the instance (i.e., the PID). In this example, we say the EchoServer description applies to the configurations of managed services with the PID org.foo.managed.service.

Using meta type information

To use meta type information, we use the Meta Type service to look up meta type definitions. The Meta Type service is represented by the following interface:

```
public interface MetaTypeService {
    public MetaTypeInformation getMetaTypeInformation(Bundle bundle);
}
```

Using the discovered meta type information, we could generate user interfaces or validate configurations, for example. To demonstrate how to use the Meta Type service, we will add a "type" command to our shell to display meta type information; see Listing 7.8.

Listing 7.8: Meta Type service shell command example

```
public class MetaDataCommand extends BasicCommand {

  public void exec(String args, PrintStream out, PrintStream err)
    throws Exception {
    MetaTypeService mts = getMetaTypeService();                          #1
    Bundle b = getBundle(args);                                          #2
    MetaTypeInformation mti = mts.getMetaTypeInformation(b);             #3
    String[] pids = mti.getPids();                                       #4
    for (int i = 0; i < pids.length; i++) {
      out.println(pids[i]);
      ObjectClassDefinition ocd = mti.getObjectClassDefinition(          #5
        pids[i], null);
      AttributeDefinition[] ads = ocd                                    #6
        .getAttributeDefinitions(ObjectClassDefinition.ALL);
      for (int j = 0; j < ads.length; j++) {
        out.println("\tOCD=" + ocd.getName());                           #7
        out.println("\t\tAD=" + ads[j].getName() + " - " +               #7
          ads[j].getDescription());                                      #7
      }
    }
  }

  private MetaTypeService getMetaTypeService() {...}

}
```

The command is quite simple, we just ask the MetaTypeService if a specified bundle has MetaTypeInformation objects associated with it. The "type" command accepts a bundle identifier as an argument. At (#1), we get the MetaTypeService and at (#2) we retrieved the Bundle object associated with the specified bundle identifier. We invoke the getMetaTypeInformation() method at (#3) for retrieving an associated meta type

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

information. If there is meta type information, then we get the PIDs at (#4) and for each PID we get the object class definition at (#5). Likewise, for each object class definition, we get the `AttributeDefinition`s at (#6) and print their names and descriptions at (#7). We can now use this command to get a list of all known PIDs and their respective properties for any given bundle identifier.

To run this example, go into the `code/chapter07/managed-example` directory of the companion code. Type `ant` to build the example and `java -jar launcher.jar bundles` to execute it. To interact with the shell use `telnet localhost 7070`. For this example, we use the Apache Felix Metatype implementation[3]. Listing 7.9 shows a session using the "type" command.

Listing 7.9 Metatype command session

```
-> bundles
  ID      State       Name
[   0] [   ACTIVE] System Bundle
                     Location: System Bundle
                     Symbolic Name: org.apache.felix.framework
[   1] [   ACTIVE] managed.service
                     Location: file:bundles/managed.service-1.0.jar
                     Symbolic Name: org.foo.managed.service
…
-> type 1
org.foo.managed.service
    OCD=EchoServer
       AD=port - The port the Echo Server listens on
```

As you can see, all we need to do is to execute the "type" command with a bundle identifier of a bundle providing metadata and we get a description of what kind of properties a given PID can understand.

So where are we now? We learned how we can configure our bundles and provide meta type information about our configuration properties. This combination allows us to create externally and generically configurable applications. What more do we need? Not all configuration information is intended to be externally managed; for example, most preference settings in an application fall under this category. Where should a bundle store such configuration information? The OSGi Preferences service can help us out here, let's look at how it works next.

7.2.3 Preferences service

In many cases, your applications need to store preferences and settings persistently. Of course, this chapter is about managing bundles and, technically, dealing with preference settings is not really a management activity. Still, we include it here since it is related to configuration data in general and it gives us an opportunity to present another standard OSGi compendium service.

[3] http://felix.apache.org/site/apache-felix-metatype-service.html

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The OSGi Preferences service gives bundles a mechanism to persistently store data. You might recall from chapter 3 that a bundle already has a private file system area, which it can access via `BundleContext.getDataFile()`. While we could use this mechanism to store preference settings, the Preferences service has several advantages:

- It defines a standard way to handle such data.
- It supports hierarchical system and per-user settings.
- It does not actually require a file system.
- It can abstract access to the underlying operating system's settings mechanism, if one exists.

The Preferences service provides simple, lightweight access to stored data. It does not define a general database service, but is optimized to deliver stored information when needed. It will, for example, return defaults instead of throwing exceptions when the back-end store is not available.

The Preferences service data model is a multi-rooted hierarchy of nodes, where a "system root" node exists for system settings and any number of named "user root" nodes can be created for user settings. Each one of these root nodes is the root of a tree of `Preferences` objects. A `Preferences` object has a name, a single parent node (except for a root node which has no parent), and zero or more child nodes. It is possible to navigate a tree either by walking from one node to its parent or children or by addressing nodes directly via a relative or absolute path. This is possible using the node names separated with the "/" character, much like file system paths. Figure 7.8 shows a conceptual picture of the trees.

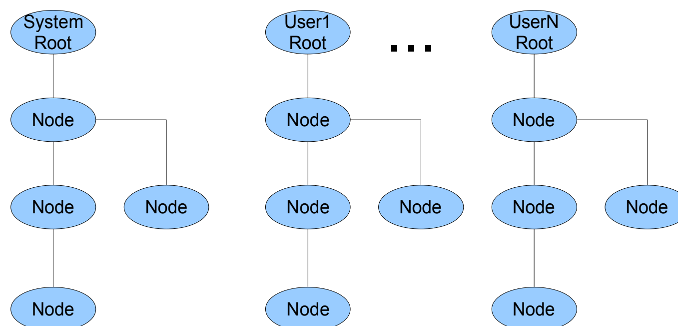


Figure 7.8: System and User Preferences Trees

Each `Preferences` object has a set of key/value pairs, called properties. The key is a case sensitive string that lives in a separate name space from that of the child nodes. So the same node can have a property with the same key as the name of one of its children. The value must always be able to be stored and retrieved as a string. Therefore, it must be

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

possible to encode/decode all values into/from strings. A number of methods are available to store and retrieve values as primitive types.

PREFERENCES ARE PER BUNDLE

The preferences saved by one bundle are completely distinct from the preferences saved by another bundle. The Preferences service does not provide a mechanism for one bundle to access another bundle's preferences storage. If this is needed, a reference to the source bundle's preferences must be attained in another way, such as directly passing a reference to the other bundle.

Using the Preferences service is straightforward. To access the system preferences root, all you have to do is the following:

```
Preferences root = service.getSystemPreferences();
```

To access a user preferences root, you just do this:

```
Preferences fooUser = service.getUserPreferences("foo");
```

When you have the node you can navigate the preference tree using the `childrenNames()`, `parent()`, and `node()` methods on the returned `Preferences` node. For setting values, the `Preferences` interface offers some simple methods to store key/value pairs:

```
public void put(String key, String value);
public void putInt(String key, int value);
public void putLong(String key, long value);
public void putBoolean(String key, boolean value);
public void putFloat(String key, float value);
public void putDouble(String key, double value);
public void putByteArray(String key, byte[] value);
```

For each of these methods, a correspondent getter method exists. Getter methods always accept two arguments: the first to specify the key of the property to retrieve and the second to specify a default value in case the property doesn't exist (or in case of errors). For instance:

```
public float getFloat(String key, float def);
```

So assuming you want to store the last time your bundle was started, you can do this using the system preferences:

```
Preferences startPreferences =
  service.getSystemPreferences().node("start");
startPreferences.putLong("time", new Date().getTime());
```

This stores the current time as a `long` in the systems preferences "start" node. As you can see, this is pretty simple stuff, but it is convenient to have a standard service definition rather than having to invent it yourself.

Isn't this just Java Preferences?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

> Generally speaking, the Preferences services is very similar to the `java.util.prefs.Preferences` introduced in Java 1.4. One of the reasons the OSGi Preferences service exists is because the Java Preferences API isn't available before Java 1.4 and OSGi still supports Java 1.3. At the same time, the OSGi Preferences service saves preferences for each bundle independently of other bundles, while Java Preferences saves preferences of one user of the system independently of other users. So the two, while similar, are not identical.

This concludes our section on bundle configuration. We've covered a lot of ground. The combination of the Configuration Admin, Meta Type, and Preferences services provides for very flexible approaches when it comes to configuring your bundles, which can save you a lot of management effort. But to manage our bundles in the first place, we have to deploy them into an OSGi framework. Next we will look at some management tools to help us deploy bundles and OSGi-based applications.

7.3 Deploying bundles

Once we have created some configurable bundles and versioned them according to a meaningful policy, we need to install them in an OSGi framework. In chapter [ref], we looked at the various details of the lifecycle layer API, which allows us to install, start, update and uninstall bundles from a running framework. Given the nature of modularity, it is likely your applications will grow over time to include too many bundles for you to manage their deployment in an ad-hoc fashion. Manually installing and updating 10's, 100's, or even 1000's of bundles becomes becomes impractical. What can we do? This is when it becomes important to think about how you (or your users) are going to discover and deploy bundles.

Introducing management agents

The solution, in OSGi lingo, is to create a specific type of bundle, called a management agent. Although we have shown how to programmatically manipulate the lifecycle of a bundle, it's typically not a good idea for a bundle to change its own state or the state of other bundles. Such a bundle is very difficult to reuse in other compositions, since it is tightly bound to the other bundles it expects to control. The solution employed by most management agents is to externalize the information about which bundles to install or start. For example, this management information could refer to bundles using URIs and could aggregate useful groups of bundles using some sort of composition language/mechanism. Your management agent is able to generically process such information, leaving it nicely decoupled from the bundles it is managing.

An example of this is our shell from chapter 3. It is, in fact, a management agent. Granted, it's a simple agent since it only accepts and executes commands, but if it is sufficient for your application, then it is perfectly fine. A management agent can be much more powerful, however. Even for our shell, we could easily extend it to handle a command

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

scripts to execute commands in batches. You could create a couple scripts, one for each configuration you need. Switching between application configurations would then be trivial.

Even more sophisticated management agents are possible. The shell as a management agent assumes human interaction to either directly or indirectly make the right decisions and issue commands to manage the bundles. We could devise a system with rules to automate some of this by reacting to certain conditions autonomously. Consider a home automation system able to detect a new device and automatically discover a driver for it in a remote repository and subsequently install the driver along with its dependencies. Or an application that automatically adapts itself to the language of the current user by installing the necessary locale bundles.

In essence, a management agent manages a running framework. OSGi supports us in developing such an agent by providing us with the means to monitor and manipulate a running framework. One of the more critical aspects of managing the framework is determining which bundles should be deploy to it. Various strategies are possible to manage complex sets of interdependent bundles. The two most prominent at the moment are the OSGi Bundle Repository (OBR) and Deployment Admin.

OBR and Deployment Admin address bundle deployment from different angles, but both can help when it comes to developing a management agent. The difference in focus between the two can be summarized as:

- OBR focuses on remote discovery and deployment of individual bundles.
- Deployment Admin focuses on the deployment of sets of bundles and associated resources.

In the following sections we will explore these two technologies in more detail and show you how to use them to provision or deploy your applications and bundles.

Alternative technologies

There are a number of other technologies attempting to address deployment and provisioning for OSGi, including Apache Ace, Eclipse p2, and Paremus Nimble:

- Ace is a software distribution framework based on Deployment Admin. It focuses on centrally managing target systems and distributing software components, configuration data, and other artifacts to them. The target systems are usually OSGi based, but don't have to be.
- Nimble is based on open source work from the Newton project and focuses on building an extensible resolver architecture able to deal with other types of dependencies outside of the OSGi modularity layer such as service-level dependencies; for example, if a bundle containing servlets is deployed and activated, then a servlet container should be deployed and activated along side it.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

- p2 is a subproject of the Eclipse Equinox framework. p2 focuses on extending the types of deployable artifacts to encompass things outside of an OSGi environment, including Unix RPM packages or windows services, for example.

We won't discuss the details of any of these in the remainder of this book. If you are interested in any of them, they are just a Google search away.

7.3.1 OSGi Bundle Repository

The OSGi Bundle Repository (OBR) is officially not an OSGi standard specification; rather, it is a proposal for a specification, internally referred to as RFC 112 in the OSGi Alliance. Since OBR is only an RFC, its details may change in the future, but it is still a useful tool as it is.

OBR started life as the Oscar Bundle Repository, which was associated with the Oscar OSGi framework (and ultimately became the Apache Felix framework). OBR is intended to address two aspects of bundle deployment:

1. Discovery – provide a simple mechanism to discover which bundles are available for deployment.
2. Dependency deployment – provide a simple mechanism to deploy a bundle and its transitive set of dependencies.

To achieve (1), OBR defines a simple bundle repository with an API for accessing it and a common XML interchange format for describing deployable resources. An OBR repository can refer to other OBR repositories, defining a federation of repositories. The OSGi Alliance hosts their own repository at bundles.osgi.org. However, it is not necessary to define federations, so it is possible to create independent repositories specifically for your own purposes and applications. One of the main goals of OBR was simplicity, so it was easy for anyone to provide a bundle repository. One of the benefits of define an XML-based repository format, is

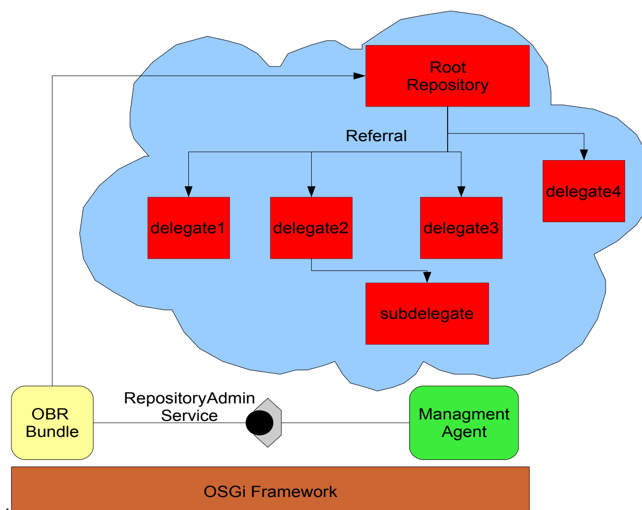


Figure 7.9: The OBR specification provides a federated index that allows a management agent to resolve and install large numbers of bundles from a number of remote locations. The OBR index files are aggregated by the RepositoryAdmin service which resolves bundle dependencies on behalf of a ManagementAgent.

©Manning Pub ... line forum:

that the no server-side process is needed (although a server-side process is also possible). Figure 7.9 shows the federated structure of an OBR repository.

The key concept of an OBR repository is a generic description of a resource and its dependencies. A resource is an abstract entity that can be used to represent any type of artifact such as a bundle, a certificate, or a configuration file. The resource description allows an agent to discover applicable artifacts, typically bundles, and deploy them along with their transitive dependencies. Each resource description has:

- Zero or more requirements on other resources or the environment and
- Zero or more capabilities that are used to satisfy other resources' requirements.

Resource requirements are satisfied by capabilities provided by other resources or the environment. OBR maps bundle metadata from `Import-Package` and `Require-Bundle` headers onto resource requirements and from `Export-Package` and `Bundle-SymbolicName` headers onto resource capabilities. Figure 7.10 shows the relationship among the repository entities.

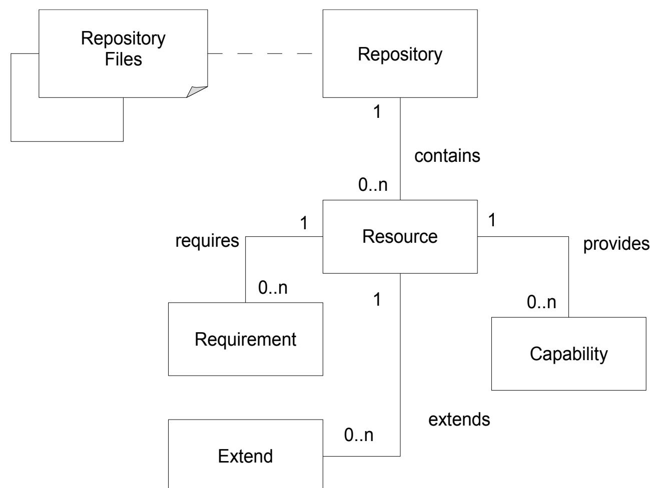


Figure 7.10: Relationships among the OBR repository entities

Using this information, an OBR implementation is able to resolve a consistent set of bundles for deployment given an initial set of bundles and a set of bundles to be deployed. OBR's dependency resolution algorithm is basically the same as the framework's dependency resolution algorithm.

OBR versus framework resolution

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

> Although the dependency resolution algorithms for OBR and the framework are similar, they are not identical. OBR starts from a given set of bundles and will pull in resources from its available repositories in an attempt to satisfy any dependencies. The framework resolution algorithm will never pull in additional resources; it only considers installed bundles. Another gotcha is the current OBR RFC does not mandate "uses" constraints when resolving dependencies. This can lead to unexpected failures at execution time if a "uses" constraint prevents bundles from resolving. OBR is an active area of work within the OSGi Alliance, so future revisions of the RFC may address this issue.

With this overview of OBR, let's look at how we can create a repository for it.

Creating OBR repositories

To illustrate how to create an OBR repository, we will use the bundles from our service-based paint program example. The repository is just an XML file containing the metadata of the bundles. We will go through the entries in the XML file and explain the schema along the way. Let's assume we have the bundles from the example in a directory called `paint-bundles`. The directory contains the paint frame bundle, the API bundle, and the three shape bundles:

```
paint-bundles/
  frame-4.0.jar
  circle-4.0.jar
  triangle-4.0.jar
  shape-4.0.jar
  square-4.0.jar
```

We could create the repository XML file by hand, but there are several different tools we can use to create them instead. We will use probably the simplest, called bindex[4], provided by the OSGi Alliance. For Maven users, there is also Maven integration, which we will discuss in appendix [ref]. To create a repository using bindex, run the following from above the bundles directory (this example assumes you are in `code/chapter07/shell-example/`),

```
java -jar bindex.jar -r repository.xml -n Paint paint-bundles/*.jar
```

This creates a `repository.xml` file which contains the metadata of the bundles from the example. The main XML element is a repository tag defining the repository:

```
<repository lastmodified='20090215101706.874' name='Paint'>
…
</repository>
```

The `lastmodified` attribute is used as a timestamp by the OBR service to determine whether something has changed. The most interesting element is the resource tag. It describes the bundles we want to make available. The created repository XML file contains a resource block per bundle. Our shape API bundle converted into OBR is shown in Listing 7.10.

Listing 7.10: Shape API bundle converted into OBR repository XML syntax

```
<resource id='org.foo.shape/4.0.0' presentationname='shape'
```

[4] http://www.osgi.org/Repository/BIndex

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
symbolicname='org.foo.shape' uri='paint-bundles/shape-4.0.jar'
version='4.0.0'>
  <size>
5742
  </size>
  <license>
    http://www.apache.org/licenses/LICENSE-2.0
  </license>
  <documentation>
http://code.google.com/p/osgi-in-action/
  </documentation>
  <capability name='bundle'>                                           #1
    <p n='manifestversion' v='2'/>                                     #1
    <p n='presentationname' v='shape'/>                                #1
    <p n='symbolicname' v='org.foo.shape'/>                            #1
    <p n='version' t='version' v='4.0.0'/>                             #1
  </capability>                                                        #1
  <capability name='package'>                                          #2
    <p n='package' v='org.foo.shape'/>                                 #2
    <p n='version' t='version' v='4.0.0'/>                             #2
  </capability>                                                        #2
  <require extend='false' filter='(&amp;(package=org.foo.shape)       #3
(version&gt;=4.0.0)(version&lt;5.0.0))' multiple='false' name='package'  #3
 optional='false'>                                                     #3
    Import package org.foo.shape ;version=[4.0.0,5.0.0)               #3
  </require>                                                           #3
</resource>
```

The capability elements at (#1) and (#2) represent what the bundle provides. In this case, (#1) represents the bundle itself, since the bundle itself can be required (e.g., `Require-Bundle`), while (#2) represents the package exported by the bundle. Bundle dependencies are represented as requirement elements, such as the one at (#3) for an imported package. Both capabilities and resources have a name, which is actually a namespace and is how capabilities are matches to requirements. For example, capabilities representing exported packages and requirements representing imported packages both have the `package` namespace.

In general, a capability is actually a set of properties specified using a `<p>` element with the following attributes:

- `n` – the name of the property
- `v` – the value of the property
- `t` – the type of the property, which is one of:
  - `string` – A string value, which is the default
  - `version` – An OSGi version
  - `uri` – A URI
  - `long` – A long value
  - `double` – A double value

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

- `set` – A comma separated list of values

Looking more closely at the bundle capability at (#1), we can see it is a fairly straightforward mapping from the bundle identification metadata:

```
Bundle-ManifestVersion: 2
Bundle-Name: Simple Paint API
Bundle-SymbolicName: org.foo.shape
Bundle-Version: 4.0
```

Likewise, the package capability at (#2) is also a simple mapping from the bundle's Export-Package header:

```
Export-Package: org.foo.shape;version="4.0"
```

A requirement is an LDAP query over the properties of a capability. So, to match a requirement to a capability, first the namespace must match. If that matches, then the requirements LDAP query must match the properties supplied by the capabilities. Even with the LDAP query, the package requirement at (#3) is a fairly easy mapping from the Import-Package header:

```
Import-Package: org.foo.shape;version="[4.0,5.0)"
```

One reason why the filter at (#3) looks somewhat more complicated than necessary is that version ranges are not directly supported by the filter syntax and must be expressed as the lower and upper bound.

If our bundle had a `Require-Bundle`, `Fragment-Host`, or `Bundle-ExecutionEnvironment` header, these would all be mapped to requirements. Even though the mappings are straightforward, it is still nice to have a tool like bindex doing this for us. We could even integrate bindex into in our build cycle so our repository is updated whenever our bundles change.

The repository XML is fine and all of that, but you are probably wondering how you can use repositories in your management agent. In fact, you don't need to know anything about the XML format to use OBR. All we need to do is grab the service implemented by OBR and use it. Let's have a closer look at this.

Browsing OBR repositories

The best way to get familiar with how to use repositories is to give an example and explain what it does along the way. We will reuse the shell example again and extend it with a new command to add/remove/list repositories and browse the bundles inside them. The programatic entry point to the OBR specification is the `RepositoryAdmin` service, which is represented by the following interface:

```
public interface RepositoryAdmin {
  Resource[] discoverResources(String filterExpr);
  Resolver resolver();
  Repository addRepository(URL repository) throws Exception;
  boolean removeRepository(URL repository);
  Repository[] listRepositories();
  Resource getResource(String respositoryId);
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

This `RepositoryAdmin` service provides centralized access to the federated repository. An OBR implementation implements this interface as well as the other types referenced by it. Figure 7.11 shows the relationships among the involved entities.

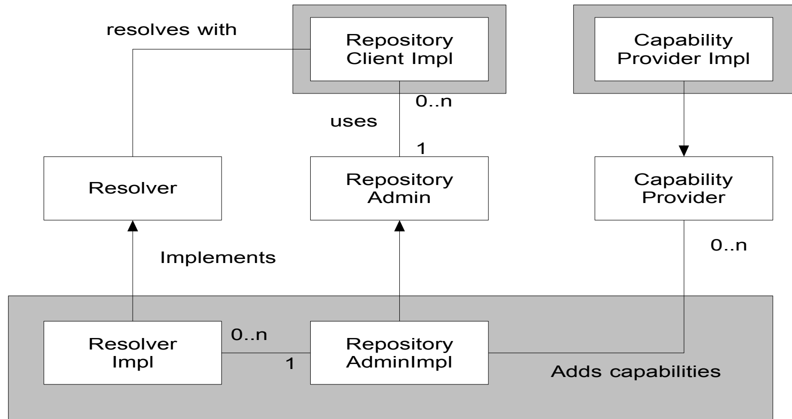


Figure 7.11: UML diagram of the RepositoryAdmin service. An external RepositoryClient uses the RepositoryAdmin and Resolver interfaces to install download and install bundles and their dependencies.

The code in Listing 7.11 shows the code for our new "obr-repo" command. It uses `RepositoryAdmin` to add, remove, and list repositories as well as to discover resources.

Listing 7.11: OBR repository shell command example

```
public class RepositoryCommand extends BasicCommand {
  public void exec(String args, PrintStream out, PrintStream err)
    throws Exception {
    args = args.trim();
    RepositoryAdmin admin = getRepositoryAdmin();
    if (admin != null) {
      if ("list-urls".equalsIgnoreCase(args)) {                    #1
        for (Repository repo : admin.listRepositories()) {         #1
          out.println(repo.getName() + " (" + repo.getURL() + ")"); #1
        }                                                          #1
      } else if (args != null) {
        if (args.startsWith("add-url")) {                          #2
          admin.addRepository(                                     #2
            new URL(args.substring("add-url".length())));          #2
        } else if (args.startsWith("remove-url")) {                #3
          admin.removeRepository(                                  #3
            new URL(args.substring("remove-url".length())));       #3
        } else if (args.startsWith("list")) {                      #4
          String query = (args.equals("list"))                     #4
            ? "(symbolicname=*)"                                   #4
            : args.substring("list".length()).trim();              #4
          for (Resource res : admin.discoverResources(query)) {    #4
            out.println(res.getPresentationName() + " ("           #4
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
          + res.getSymbolicName() + ") " + res.getVersion());         #4
        }                                                             #4
      }                                                               #4
    } else {
      out.println(
        "Unknown command - use {list-urls|add-url|remove-url|list}");
    }
  } else {
    out.println("No RepositoryAdmin service found...");
  }
}

private RepositoryAdmin getRepositoryAdmin() {
  ...
}
}
```

The “obr-repo” command has the following subcommands: list-url, add-url, remove-url, and list. A `RepositoryAdmin` provides access to a number of repositories referenced by URLs. At (#1), we implement the “list-url” subcommand to list these repositories by retrieving the `RepositoryAdmin` service and calling it's `listRepositories()` method, which gives us access to the associated `Repository` objects. In this case, we loop through the repositories and print their names and URLs.

You can add or remove repository URLs with the “add-url” and “remove-url” subcommands, respectively. As you can see at (#2) and (#3), there is a one-to-one mapping to the `addRepository()` and `removeRepository()` methods of the `RepositoryAdmin` service. Finally, the “list” subcommand expects an LDAP query which it passes to `discoverRepositories()` to discover resources at (#4). If no query is specified, then all resources are listed. We loop through the discovered resources and print their presentation name, symbolic name, and version.

We can now use this command to configure repositories and discover bundles. Once we've discovered a bundle we want to use, we need to deploy it. We'll implement a separate command for that next.

Deploying bundles with OBR

Discovering bundles is one half of the OBR story, the other half is deploying them and their dependencies into the framework. The `RepositoryAdmin.getResolver()` method gives us access to a Resolver object to select, resolve, and deploy resources. A Resolver has these methods:

```
public interface Resolver {
  void add(Resource resource);
  Requirement[] getUnsatisfiedRequirements();
  Resource[] getOptionalResources();
  Requirement[] getReason(Resource resource);
  Resource[] getResources(Requirement requirement);
  Resource[] getRequiredResources();
  Resource[] getAddedResources();
  boolean resolve();
  void deploy(boolean start);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
}
```

The process for deploying resources is fairly simple, just follow these steps:

1. Add desired resources using `Resolver.add()`.
2. Resolve the desired resources' dependencies with `Resolver.resolve()`.
3. If the desired resources resolve, then deploy them with `Resolver.deploy()`.

In Listing 7.12 we implement an “obr-resolver” shell command to resolve and deploy resources.

Listing 7.12: OBR resolver shell command example

```
public class ResolverCommand extends BasicCommand {
  public void exec(String args, PrintStream out, PrintStream err)
    throws Exception {
    RepositoryAdmin admin = getRepositoryAdmin();
    Resolver resolver = admin.resolver();                                #1
    Resource[] resources = admin.discoverResources(args);                #2
    if ((resources != null) && (resources.length > 0)) {
      resolver.add(resources[0]);                                        #3
      if (resolver.resolve()) {                                          #3
        for (Resource res : resolver.getRequiredResources()) {           #4
          out.println("Deploying dependency: " +                         #4
            res.getPresentationName() +                                  #4
            " (" + res.getSymbolicName() + ") " + res.getVersion());     #4
        }                                                                #4
        resolver.deploy(true);                                           #5
      } else {                                                           #6
        out.println("Can not resolve " + resources[0].getId() +          #6
          " reason: ");                                                  #6
        for (Requirement req : resolver.getUnsatisfiedRequirements()) {  #6
          out.println("missing " + req.getName()                         #6
            + " " + req.getFilter());                                    #6
        }                                                                #6
      }
    } else {
      out.println("No such resource");
    }
  }

  private RepositoryAdmin getRepositoryAdmin() {
    ...
  }
}
```

We get the `Resolver` from the `RepositoryAdmin` service at (#1). We use the `RepositoryAdmin.discoverResources()` method with a LDAP filter argument to discover a resource to deploy at (#2). If we find any resources, we add the first one to the `Resolver` and call `resolve()` to resolve its dependencies from the available repositories at (#3). If the resource is successfully resolved, then we print out all of the dependencies of the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

resource we are deploying at (#4). At (#5) we use `Resolver.deploy()` to install and start the discovered bundle and its dependencies. If the resource couldn't be resolved, we print out the missing requirements at (#6).

To run this example, go into the `code/chapter07/shell-example/` directory of the companion code. Type `ant` to build the example and `java -jar launcher.jar bundles` to execute it. To interact with the shell use `telnet localhost 7070`. For this example, we use the Apache Felix OBR implementation[5]. Listing 7.13 shows a session using the "obr-repo" and "obr-resolver" commands.

Listing 7.13 OBR command example session

```
-> obr-repo add-url file:repository.xml                                    #1
-> obr-repo list-urls                                                      #2
Paint (file:repository.xml)
-> obr-repo list                                                           #3
circle (org.foo.shape.circle) 4.0.0
frame (org.foo.paint) 4.0.0
shape (org.foo.shape) 4.0.0
square (org.foo.shape.square) 4.0.0
triangle (org.foo.shape.triangle) 4.0.0
-> obr-resolver (symbolicname=org.foo.paint)                               #4
Deploying dependency: shape (org.foo.shape) 4.0.0
-> obr-resolver (symbolicname=org.foo.shape.circle)                        #5
```

In this session, we first use the "add-url" subcommand to add our repository containing the paint program bundles at (#1). We verify the configured repository using the "list-url" subcommand at (#2). Using the "list" subcommand at (#3), we browse the bundles contained in the repository. At (#4) we use the "obr-resolver" command with an LDAP filter to select and deploy the paint frame bundle, which also installs its dependencies. At (#5), we install the circle bundle based on its symbolic name.

That's about all you need to know to start using OBR to discover and deploy your bundles. Often, this is all you need to manage the growing complexity of your applications. However, sometimes you will be faced with a slightly different scenario which doesn't fit as well with what OBR provides. Perhaps you want to package your application in a single deployment unit composed of several bundles. What can you do in this case? There is another OSGi compendium specification that targets such needs. Let's look at that next.

7.3.2 Deployment Admin

With OBR you tend to think about deploying specific bundles and letting OBR automatically calculate and deploy any dependent bundles. With Deployment Admin you tend to think about deploying entire applications or subsystems as a single unit. The Deployment Admin specification standardizes some of the responsibilities of a management agent; specifically, it addresses lifecycle management of interlinked resources on an OSGi Service Platform.

---

[5] http://felix.apache.org/site/apache-felix-osgi-bundle-repository.html

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Deployment Admin defines a way to package a number of resources in a deployment package. A deployment package is a JAR file with a format similar to bundles. Deployment packages can be installed using the `DeploymentAdmin` service. The `DeploymentAdmin` service can process bundle resources itself, but other types of resources in the deployment package by handing them off to a `ResourceProcessor` service for that specific type of resource. A `ResourceProcessor` service will appropriately process a given type of resource. The uninstallation and update of a deployment package works similarly, where bundles are processed by the `DeploymentAdmin` service and other types of resources are handed off to `ResourceProcessor`s. All `ResourceProcessor` services are notified about any resources that are dropped or changed. If all resources have been processed, the changes are committed. If an operation fails, all changes are rolled back.

NOT ACTUALLY TRANSACTIONAL

Although we are talking in terms of commits and rollbacks, the DeploymentAdmin service is not guaranteed to support all features of transactions. Most implementations tend to provide only a best effort rollback.

This sounds fairly promising for managing applications. To get a better idea of how it works, we will present some of the details of deployment packages next. After that, we'll give an example of how you can use the Deployment Admin to install and manage deployment packages.

Creating deployment packages

As an example, let's think about how we would provision our paint program. The paint program has the following artifacts:

```
paint-4.0.jar
shape-4.0.jar
circle-4.0.jar
square-4.0.jar
triangle-4.0.jar
```

In order to be able to show all of what deployment packages have to offer, let's assume we want to provide a core version of the program containing the drawing frame and the shape API bundles. This way we are able to deploy the actual shape implementations separately via an “extension pack”. The latter contains the square, circle, and triangle bundles. Let's go with this example and explore the different ways you can use deployment packages to make it work.

The general structure of a deployment package is shown in Figure 7.12. This ordering is carefully designed to allow deployment packages to be streamed in such a way that the contents can be processed without needing to download the entire JAR file.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

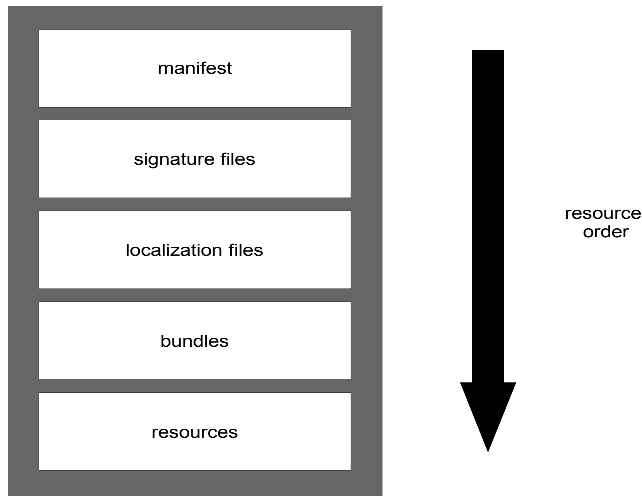


Figure 7.12: Structure of a deployment package JAR file

The deployment package design has a few other desirable characteristics. First, the deployment package puts metadata in its manifest, similar to bundles, which allows us to turn it into a named and versioned set of resources. Second, by taking advantage of the fact that JAR files can be signed, we can use signed JAR files to make our deployment packages tamperproof.

For our example, we could chose to (shown in Figure 7.13):

1. Create a deployment package for the core bundles and one package for all shape bundles or
2. Create a deployment package for the core bundles and individual deployment packages for each shape bundle.

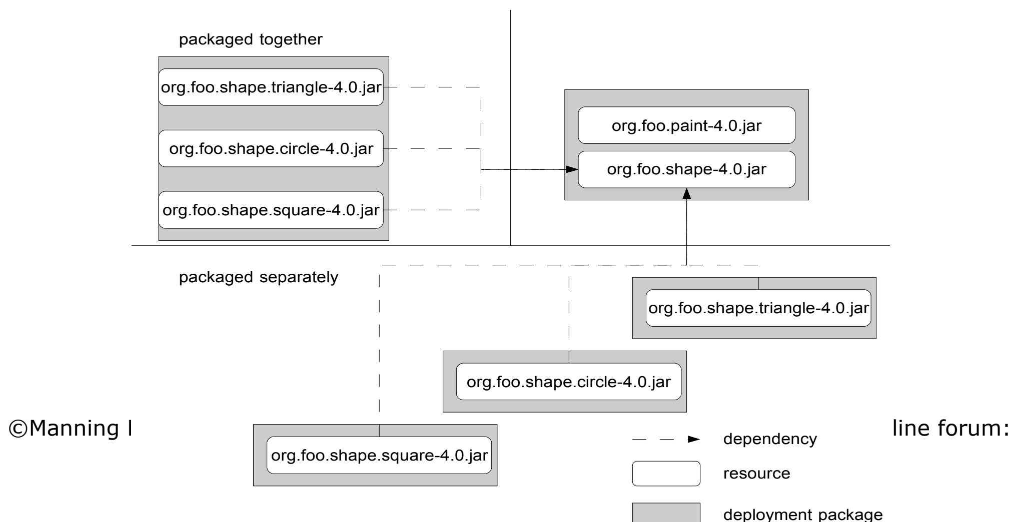


Figure 7.13: Paint program packaging alternatives

©Manning I ... line forum:

DEPLOYMENT PACKAGES ARE GREEDY

These two different packaging strategies cannot be used simultaneously. The specification only allows resources to belong to a single resource package. Using both approaches at the same time or changing the approach after the fact would move ownership of the bundle resources to another deployment package and thus violate the specification. A deployment package is defined as a set of resources with the need to be managed as a unit. The resources in a deployment package are assumed to be tightly coupled, such as a bundle and its configuration data. As a consequence, a resource can only belong to one deployment package, otherwise you could run into situations where you have two different, conflicting configurations for the same bundle, for example.

In terms of our example this leaves use with the need to make a decision. We will go with the first approach and create a single deployment bundle for all shapes. However, since deployment packages can be updated we could gain some flexibility by starting with only one shape in the deployment package and then adding another one in an updated version and another for the third or other combinations. Actually, when we create an update which just adds or removes resources from a previous version then we don't even have to package the resources inside the update; instead we can use fix packages.

WHAT'S A FIX PACKAGE?

A fix-package is a deployment package that minimizes download time by excluding resources that are not required to upgrade or downgrade a deployment package. It can only be installed if a previous version of that deployment package is already installed. A fix package only contains the changed and new resources. A fix package (called the source) therefore must specify the range of versions that the existing deployment package (called the target) must have installed. You will see shortly when we walk through the example.

Now that we've figured out our packaging approach, how do we proceed? We need to create a manifest for the target which will contain the paint frame and shape API bundles, which we'll use to provision the paint program core. Then we need to create the manifest of the fix package which we'll use to add the three shape bundles to the core. Once we have our manifests, we need to create two JAR files with the corresponding manifests and our bundles, optionally sign them, and we are good to go. Listing 7.14 shows the manifest of the core deployment package.

Listing 7.14: Core paint program deployment package manifest

```
Manifest-Version: 1.0
DeploymentPackage-SymbolicName: org.foo.paint                          #1
DeploymentPackage-Version: 1.0.0                                        #2
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Name: paint-4.0.jar                                           #3
Bundle-SymbolicName: org.foo.paint                            #4
Bundle-Version: 4.0.0                                         #5

Name: shape-4.0.jar
Bundle-SymbolicName: org.foo.shape
Bundle-Version: 4.0.0
```

We specify deployment package's symbolic name and version at (#1) and (#2), respectively. Next we specify the list of resources contained in the JAR file. We specify the name of a resource at (#3), its symbolic name at (#4), and its version at (#5). We must do this for each resource. For this example, we only have bundle resources. To finish, we just need to use the `jar` tool to create the JAR file with the appropriate content and we are done with our first deployment package.

Signing deployment packages

In this example we didn't sign our deployment package, nor is it required for us to do so. If we wanted to create a signed deployment package, we just use the `jarsigner` tool from the standard Java SDK. The signing process is no different than signing a normal JAR file, which results in the signatures being placed inside the deployment package JAR file inside the `META-INF` directory and after the `MANIFEST.MF` file. Additionally, each entry section in the manifest will contain a digest entry.

Now we need to create the manifest for our fix package containing the shape bundles. Listing 7.15 shows the manifest.

Listing 7.15: Paint program Fix-Package

```
Manifest-Version: 1.0
DeploymentPackage-Symbolicname: org.foo.paint                 #1
DeploymentPackage-Version: 2.0                                #2
DeploymentPackage-FixPack: [1,2)                              #3

Name: paint-4.0.jar
Bundle-SymbolicName: org.foo.paint
Bundle-Version: 4.0.0
DeploymentPackage-Missing: true                               #4

Name: shape-4.0.jar
Bundle-SymbolicName: org.foo.shape
Bundle-Version: 4.0.0
DeploymentPackage-Missing: true                               #5

Name: triangle-4.0.jar                                        #6
Bundle-SymbolicName: org.foo.shape.triangle
Bundle-Version: 4.0.0

Name: circle-4.0.jar                                          #7
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Bundle-SymbolicName: org.foo.shape.circle
Bundle-Version: 4.0.0

Name: square-4.0.jar                                                    #8
Bundle-SymbolicName: org.foo.shape.square
Bundle-Version: 4.0.0
```

Since the fix package is an update to our core package, the symbolic name stays the same at (#1), but the version is upgraded to `2.0.0` at (#2). The `DeploymentPackage-FixPack` header at (#3) indicates this is a fix package; we use version range syntax indicate that the fix package can be applied to any previously installed version of the deployment package from `1.0.0` inclusive to `2.0.0` exclusive. This version numbering scheme expresses the assumption that only major version number changes indicate added bundles. We now don't need to package the bundles already present in the core package, but we still need to mention them in the manifest. We use the `DeploymentPackage-Missing` header to do this at (#4) and (#5). Then we just specify the shape bundles at (#6), (#7), and (#8) in the same fashion as before. To use the deployment packages we simply need to make each available via a URL.

DEPLOYMENT PACKAGE MIME TYPE

If you make deployment packages available via a protocol that supports mime-types, the standard MIME type is for deployment packages is `application/vnd.osgi.dp`.

Next, we can use the provided `DeploymentAdmin` service in your management agent to install, update, and uninstall deployment packages.

Managing deployment packages

To demonstrate how a management agent can use Deployment Admin, we'll once again create a new "dpa" shell command to list, install, and uninstall deployment packages; we'll introduce subcommands for each of these. Our command will use the DeploymentAdmin service, which is represented by the following interface:

```
public interface DeploymentAdmin {
  DeploymentPackage installDeploymentPackage(InputStream in)
    throws DeploymentException;
  DeploymentPackage[] listDeploymentPackages();
  DeploymentPackage getDeploymentPackage(String symbName);
  DeploymentPackage getDeploymentPackage(Bundle bundle);
  boolean cancel();
}
```

Listing 7.16 shows the implementation of the command.

Listing 7.16: Deployment Admin shell command example

```
public class DeploymentPackageCommand extends BasicCommand {

  public void exec(String args, PrintStream out, PrintStream err)
    throws Exception {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    DeploymentAdmin admin = getDeploymentAdmin();

    if (admin == null) {
      out.println("No DeploymentAdmin service found.");
      return;
    }
    if (args != null) {
      if (args.trim().equalsIgnoreCase("list")) {                            #1
        for (DeploymentPackage dp : admin.listDeploymentPackages()) {        #1
          out.println(dp.getName() + " " + dp.getVersion());                 #1
        }                                                                    #1
      } else if (args.trim().startsWith("uninstall ")) {                     #2
        DeploymentPackage dp = admin.getDeploymentPackage(                   #2
          args.trim().substring("uninstall ".length()));                     #2
        if (dp != null) {                                                    #2
          dp.uninstall();                                                    #2
        } else {                                                             #2
          out.println("No such package");                                    #2
        }                                                                    #2
      } else if (args.trim().startsWith("install ")) {                       #3
        DeploymentPackage dp = admin.installDeploymentPackage(new URL(       #3
          args.trim().substring("install ".length())).openStream());         #3
        out.println(dp.getName() + " " + dp.getVersion());                   #3
      }                                                                      #3
    } else {
      out.println("Use {list|install <url>|uninstall <name>}");
    }
  }

  private DeploymentAdmin getDeploymentAdmin() {
    ...
  }
}
```

Like the previous example commands, we more or less map the command onto the `DeploymentAdmin` service interface. At (#1), we get installed deployment packages using the `listDeploymentPackages()` service method and print their names and versions. At (#2), we uninstall an existing deployment package associated with a specified symbolic name using `DeploymentPackage.uninstall()`. Finally, at (#3) we install a deployment package from the specified URL using the `installDeploymentPackage()` service method. The approach is fairly similar to managing bundles.

To run this example, go into the `code/chapter07/shell-example/` directory of the companion code. Type `ant` to build the example and `java -jar launcher.jar bundles` to execute it. To interact with the shell use `telnet localhost 7070`. For this example, we use the Apache Felix Deployment Admin implementation[6]. Listing 7.17 shows the command in action.

Listing 7.17: Deployment Admin command session

```
-> dpa install file:org.foo.paint-1.0.dp                                     #1
```

[6] WE NEED A URL FOR THIS!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
org.foo.paint 1.0.0
-> dpa install file:org.foo.paint-2.0.dp                                    #2
org.foo.paint 2.0.0
-> dpa list                                                                 #3
org.foo.paint 2.0.0
-> dpa uninstall org.foo.paint                                              #4
```

In this session we install the core paint program deployment package at (#1). We then update it to include the fix package for the shapes at (#2). At (#3), we list the installed deployment packages. Finally, at (#4) we uninstall the deployment package. This highlights the difference between the OBR and Deployment Admin approaches, since we are able to manage our bundles as a single unit of deployment rather than individual bundles.

Before concluding our discussion on Deployment Admin, we'll discuss resource processors. Resource processors are an important part of the Deployment Admin specification, since they extend OSGi deployment beyond bundles.

Resource processors

Deployment Admin can process bundle resources in deployment packages by itself, but when it comes to other types of resources it needs to enlist the help of `ResourceProcessor` services. A `ResourceProcessor` is a service to appropriately process arbitrary resource types; they implement the following interface:

```
public interface ResourceProcessor {
  void begin(DeploymentSession session);
  void process(String name, InputStream stream)
    throws ResourceProcessorException;
  void dropped(String resource) throws ResourceProcessorException;
  void dropAllResources() throws ResourceProcessorException;
  void prepare() throws ResourceProcessorException;
  void commit();
  void rollback();
  void cancel();
}
```

Deployment Admin connects resource types to resource processors using the `Resource-Processor` header in the resource entry of the deployment package manifest. We use this header to specify the service PID of the needed resource processor. These kind of services are provided by "customizer" bundles delivered as part of the deployment dackage.

A customizer bundle is indicated by using the `DeploymentPackage-Customizer` header in the resource entry for a bundle in the deployment package. This allows Deployment Admin to start customizers first, so they can provide the necessary `ResourceProcessor` services to handle the deployment package content. Resource processors may result in new file system artifacts, but can perform other tasks like database initialization or data conversion, for example. Each non-bundle resource should have a processor associated with it. With the necessary resource processor specified, Deployment Admin is able to process all resource package content.

Before processing of the deployment package starts, the `DeploymentAdmin` service creates a session in which all actions needed to process the package will take place. A

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

session is not visible to clients of the DeploymentAdmin service, it is used to join the required resource processors to the processing of the deployment package. If an exception is raised during a session by any of the resource processors or the session is canceled, then Deployment Admin rolls back the changes. As we mentioned before, this may only be a best-effort rollback, but it is normally sufficient to leave the framework in a consistent state. If no exceptions are raised during a session, then Deployment Admin commits the changes. During a commit, the DeploymentAdmin service tells all joined ResourceProcessor services to prepare and subsequently commit their changes. Figure 7.14 shows the transactional aspects of the session.

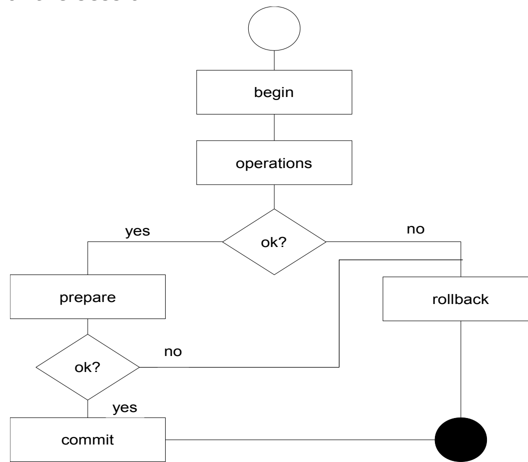


Figure 7.14: Transactional aspects of a session

As you can see, this essentially provides a two-phase commit implementation. This allows ResourceProcessors to cleanly handle rollbacks. However, rolling back a bundle update, as well as re-installing a stale bundle, requires an implementation specific back door into the OSGi framework, because the framework specification is not transactional over multiple lifecycle operations. This is the reason why the Deployment Admin specification does not mandate full transactional behavior.

In this section, we've looked at two different ways of deploying bundles. Which approach to choose depends on your needs. In summary, OBR is geared toward discovery and installation of bundles together with the transitive closure of their dependencies. Deployment Admin provisions sets of bundles and their required resources as complete units. These provide solutions for many of the deployment and discovery tasks you'll need for a management agent. Of course, if you need something else, you can always use the core OSGi API to create it as well.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Now that we know how to deploy bundles to the OSGi framework, we have one final manage-related task we should look into. Sometimes, you need to control the activation order of the deployed bundles. We'll discuss this final management activity next.

7.4 Ordering bundle activation

In certain scenarios, it might be necessary for you to control the relative order in which deployed bundles get activated and/or deactivated. There are some good reasons to control such ordering, but there are even more bad ones. Best practice dictates you should create your bundles to be independent of activation and deactivation ordering. OSGi allows bundles to listen for lifecycle events from other bundles since it eliminates the need of ordering dependencies and allows bundles to be aware of changes and react to them. Ordering constraints are another form of coupling among bundles, which severely limits their ability to be reused and arbitrarily composed. A bundle shouldn't require some functionality from another bundle be available for it to be started; instead, it should wait for the functionality to become available and then continue with its own functionality.

Having said that, there are a few valid reasons why you might want to ensure a given bundle is activated before another. For example, you might want to implement a splash screen to display the progress of your application's startup. If your splash screen is developed as a bundle, you need a way to ensure it is activated first. After all, what good would a splash screen showing the startup progress be if it came up last? We can generalize this kind of functionality as a high-priority feature, which in general requires ordering because it needs preferred treatment. Besides high-priority features, there are two other scenarios where ordering may be needed:

- When a bundle violates the best practices mentioned above and does rely on implicit activation ordering during startup. In reality, you should consider fixing or replacing this bundle, but if you cannot then you must ensure the bundles it depends on are started first. Again, this is extremely bad practice and you should feel a generous amount of shame until the bundle is fixed.
- When bundles can be grouped into sets with certain desirable properties. For example, you might define a set of bundles comprising a safe mode, where you deactivate all but a small set of trusted bundles and provide limit core functionality for safety or security reasons. Other examples could be diagnostic or power save modes.

So how can you influence and control relative activation and deactivation ordering among bundles? By using the standard Start Level service provided by the OSGi framework.

7.4.1 Introducing the Start Level service

The Start Level service allows a management agent to control the relative activation/deactivation order among bundles as well as when transitions should occur. The idea is pretty simple and you might already be familiar with it from other contexts, such as in

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Unix environments where system services are started or stopped based on the current run level of the system.

In OSGi, the framework has an active start level associated with it, which is a non-negative integer indicating the start level in which it is executing. The framework starts with an active start level of zero and, by default, will transition to an active start level of one when it is fully running. Each bundle also has an integer start level associated with it, which indicates the required start level of the bundle. Only bundles with a start level less than or equal to the framework's active start level are allowed to be in the `ACTIVE` state. The Start Level service is represented by the following interface:

```
public interface StartLevel {
  int getStartLevel();
  void setStartLevel(int startlevel);
  int getBundleStartLevel(Bundle bundle);
  void setBundleStartLevel(Bundle bundle, int startlevel);
  int getInitialBundleStartLevel();
  void setInitialBundleStartLevel(int startlevel);
  boolean isBundlePersistentlyStarted(Bundle bundle);
  boolean isBundleActivationPolicyUsed(Bundle bundle);
}
```

This service interface supports the following operations:

- Modify the active start level of the framework – You can change the framework's active start level with `setStartLevel()`. Doing so will result in all active bundles with a higher start level being stopped and bundles with a lower or equal start level that are persistently marked as started being activated.
- Assign a specific start level to a bundle – You can change an individual bundle's start level with `setBundleStartLevel()`. The framework will activate the bundle if it is persistently marked as started and the new start level is less than or equal to the active start level or will stop the bundle if the new start level is greater than the active start level.
- Set the initial start level for newly installed bundles – All bundles are installed with a default start level of one. With `setInitialBundleStartLevel()`, you can change this default value to any desired initial start level. This only impacts subsequently installed bundles.
- Query relevant values – You are able to query the framework's active start level, the start level of a bundle, and the initial bundle start level. Additionally, you can query whether a given bundle is persistently marked as started.

So what does all this mean in simple terms? The framework's active start level and a bundle's start level control whether or not a bundle can actually be started. This means if you explicitly start a bundle (i.e., invoke `Bundle.start()` on it), it will not actually activate unless the bundle's start level is less than or equal to the framework's active start level. In such a case, the only effect of invoking `Bundle.start()` is that the bundle is persistently

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

marked as started. If the framework's active start level is eventually changed to a greater or equal value, then the bundle will be activated by the framework.

As you can imagine, changing the active start level of the framework can have a dramatic impact on the framework depending, since a lot of bundles may be started or stopped as a result. When the Start Level service is used to change the framework's active start level, all active bundles with start levels greater than the target start level are stopped, while all bundles persistently marked as started with start levels less than or equal to the target start level are started. When invoking `StartLevel.setStartLevel()`, the actual process will occur on a background thread, so the method will return immediately. The background thread will effectively increment or decrement the current active start level one step at a time, depending on whether the new active start level is greater then or less than the current active start level, respectively. At each step, the background thread starts or stops the bundles at that level until the new target level is reached.

7.4.2 Using the Start Level service

To illustrate how you use the Start Level service, we will add “startlevel” and “bundlestartlevel” commands to the shell. These two commands, implemented in Listing 7.18, perform the four functions mentioned above.

Listing 7.18 Start Level service shell commands example

```
package org.foo.shell;

import java.io.PrintStream;
import org.osgi.service.startlevel.StartLevel;

public class StartLevelCommand extends BasicCommand {

  public void exec(String args, PrintStream out, PrintStream err)
    throws Exception {
    if (args == null) {
      out.println(getStartLevelService().getStartLevel());              #1
    } else {
      getStartLevelService().setStartLevel(                              #2
        Integer.parseInt(args.trim()));
    }
  }
…
}
…

public class BundleLevelCommand extends BasicCommand {

  public void exec(String args, PrintStream out, PrintStream err)
    throws Exception {
    StringTokenizer tok = new StringTokenizer(args);
    if (tok.countTokens() == 1) {
      out.println("Bundle " + args + " has level " +
```

```
          getStartLevelService().getBundleStartLevel(                      #3
            getBundle(tok.nextToken())));
      } else {
        String first = tok.nextToken();
        if ("-i".equals(first)) {
          getStartLevelService().setInitialBundleStartLevel(               #4
            Integer.parseInt(tok.nextToken()));
        } else {
          getStartLevelService().setBundleStartLevel(                      #5
            getBundle(tok.nextToken()), Integer.parseInt(first));
        }
      }
    }
  …
  }
```

As you can see at (#1), executing the "startlevel" command without an argument prints the framework's active start level. We implement this with the `StartLevel.getStartLevel()` method. If the "startlevel" command is passed a argument, then the new active start level is parsed from the argument and we call the `StartLevel.setStartLevel()` method at (#2), which cause the framework to move to the specified active start level.

Next, the "bundlelevel" command allows us to set and get the start level of an individual bundle. When the command is given only one argument, we use the argument as the bundle identifier and retrieve and output the associated bundle's start level with `StartLevel.getBundleStartLevel()` at (#3). At (#4), we add a "-i" switch to the command to set the initial bundle start level using the `StartLevel.setInitialBundleStartLevel()` method. Lastly, at (#5) we add the ability to change an individual bundle's start level using the `StartLevel.setBundleStartLevel()` method.

When the framework's active start level is changed, the background thread doing the work will fire a `FrameworkEvent.STARTLEVEL_CHANGED` event to indicate when it is done doing the work; we will capture this event in the history we added in the last section. Listing 7.19 shows a simple session demonstrating what you can do with these commands.

Listing 7.19 Using the startlevel and bundlestartlevel commands

```
-> bundles
   ID      State       Name
[   0] [   ACTIVE] System Bundle
                   Location: System Bundle
                   Symbolic-Name: system.bundle
[   1] [   ACTIVE] Simple Shell
                   Location: file:org.foo.shell-1.0.jar
                   Symbolic-Name: org.foo.shell
-> startlevel                                                               #1
1
-> bundlelevel -i 2                                                         #2
-> install file:foo.jar                                                     #3
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Bundle: 3
-> start 3
-> bundles
   ID      State       Name
[   0] [   ACTIVE] System Bundle
                   Location: System Bundle
                   Symbolic-Name: system.bundle
[   1] [   ACTIVE] Simple Shell
                   Location: file:org.foo.shell-1.0.jar
                   Symbolic-Name: org.foo.shell
[   3] [INSTALLED] Foo Bundle
                   Location: file:foo.jar
                   Symbolic-Name: org.foo.foo
-> startlevel 2                                                    #4
-> bundles
   ID      State       Name
[   0] [   ACTIVE] System Bundle
                   Location: System Bundle
                   Symbolic-Name: system.bundle
[   1] [   ACTIVE] Simple Shell
                   Location: file:org.foo.shell-1.0.jar
                   Symbolic-Name: org.foo.shell
[   3] [   ACTIVE] Foo Bundle
                   Location: foo.jar
                   Symbolic-Name: org.foo.foo
-> bundlelevel 3 3                                                 #5
-> bundles
   ID      State       Name
[   0] [   ACTIVE] System Bundle
                   Location: System Bundle
                   Symbolic-Name: system.bundle
[   1] [   ACTIVE] Simple Shell
                   Location: file:org.foo.shell-1.0.jar
                   Symbolic-Name: org.foo.shell
[   3] [ RESOLVED] Foo Bundle
                   Location: file:foo.jar
                   Symbolic-Name: org.foo.foo
```

In the example session, we first use the “startlevel” command at (#1) to display the current active start level of the framework, which is one by default. We use the “bundlelevel” command with the “-i” switch at (#2) to set the initial bundle start level of installed bundles to two. Subsequently, when we install and start the foo bundle at (#3), we can see from the following “bundles” command output that it is not actually started yet. This is expected, because the bundle's start level is two, but the framework's active start level of one is less than it. At (#4), we raise the framework's active start level to two, which ultimately causes the foo bundle to be started. Using the “bundlelevel” command at (#5) to set the foo bundle's start level to three stops the bundle again.

That's all there is to the Start Level service. You will not likely need this service often, since bundle activation ordering is not a good practice, but it can come in handy in certain situations. We've covered a lot of ground with respect to bundle and application management, let's summarize what we've learned.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

7.5 Summary

In this chapter we discussed how to manage bundles and your OSGi-based applications. Management is a broad topic, so we covered a range of topics including:

- The versioning of both packages and bundles must be carefully considered when working with OSGi.
- The OSGi specification recommends, but does not prescribe versioning policies. It is up to you to define and adhere to one.
- Managing bundles also involves managing bundle configuration data.
- The Configuration Admin service provides a way to externalize and standardize management of bundle configuration data, while the Meta Type service provides a standard way to describe a bundle's configuration data.
- Related to configuration data, the Preferences service provides a standard mechanism for bundles to manage system and user preference settings.
- One of the key management tasks is deploying bundles to the OSGi framework and there are multiple approaches for doing so, including rolling your own approach or using technologies like OBR or Deployment Admin.
- OBR focuses on discovering and deploying bundles and their transitive closure of their dependencies, while Deployment Admin focuses on defining and deploying sets of bundles and needed resources.
- Lastly, we discussed how the Start Level service can be used to control the relative activation order of your deployed bundles, which arises in a few situations like creating splash screens or different execution modes.

These topics have given us a fairly good foundation for managing our bundles. Now that we know how to build and manage our OSGi applications, how do we go about testing them? This is the topic of the next chapter.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

8 Testing Applications

You are now just over halfway through this book, congratulations! At this point you should have confidence in applying OSGi to new and existing projects. In fact migrating applications to OSGi should be especially fresh in your mind from the last chapter. But what can we do to make sure we are on the right track to modularity and not turning our application into tangled spaghetti? Like any piece of software, the best way to track quality is with regular testing. Testing can confirm your modularized code meets the same requirements as your original application. Testing can verify your code will continue to work when deployed inside the target container. It can even help you practice different deployment scenarios in the safety of your friendly neighborhood test server. Even a simple non-functional test, such as checking the number of shared packages between bundles, can avoid tangles forming early on in development.

So why wait until the end of a project to discover if your code works in the strict environment of an OSGi framework or how well your chosen bundles fit together? Migrate and modularize your tests along with your code! This chapter will help put this advice into practice by taking you through three different approaches:

- running existing tests on OSGi
- mocking out calls to OSGi APIs
- writing tests with OSGi in mind

The last section in particular takes a closer look at how unit and integration test concepts relate to modular applications and introduces the idea of management testing. If you are eager to learn more about testing modularity and you are already familiar with in-container tests and object mocking, feel free to skip ahead to the third section starting on page 318.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

By the end of this chapter you should be comfortable with testing OSGi applications, which will lead to better quality bundles for everyone. Let's start by continuing the theme from chapter 6 and get some existing tests running on an OSGi framework.

8.1 Migrating tests to OSGi

Imagine you have an application that you want to modularize and move to OSGi, you almost certainly have existing tests that check requirements and expected behavior. You can use these tests to verify and validate the modularization process; either by manually running them at key stages or by using an automated build system that runs tests on a regular schedule, say whenever people check-in code. These tests give you confidence your modularized application is to some extent equivalent to the original, at least when run with the test framework. But what they don't tell you is whether your code behaves the same inside an OSGi container.

To find out we need to run our tests *twice*: inside the target container as well as outside.

8.1.1 Container testing

Would you develop and deploy a web application without ever testing it inside an application server? Would you ship a product without testing its installer? Of course not! It is important to test code in the right environment. If you expect to use a class with OSGi you should test it inside an OSGi framework, how else would you discover potential class loading or visibility issues? But before we can run tests from inside the container, we first need to deploy them.

As we just saw in chapter 6, whenever you want to deploy something into an OSGi framework, you must consider packaging and placement. If the test classes are (accidentally) exposed from the boot class path then the tests will effectively be running outside of the container. So does this mean you should bundle tests along with the application code? It really depends on how you expect the code to be used in OSGi. Internal classes can only be tested from inside the same bundle, but public facing code can and should be tested from another bundle to mimic real-life conditions. Testing code inside the same bundle typically means the "caller" and "callee" share the same class loader, but many OSGi-related issues only appear when different class loaders are involved. So wherever possible, test from another bundle.

Figure 8.1 summarizes the four main test deployment options:

- Boot class path
- System export
- Intra-bundle
- Inter-bundle

{FIGURE}

Figure 8.1 Test deployment options

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

We can deploy tests just like any other piece of code, but how much effort is actually involved in getting tests up and running in an OSGi framework? Let's find out right now by converting an existing open source library and its test suite into bundles.

8.1.2 Bundling tests

The Apache Commons Pool project [ref] provides a small library for managing object pools. We're going to use the source distribution for Commons Pool 1.5.3, which contains the code for both the library and its test suite:

```
chapter08/migration-example/commons-pool-1.5.3-src.zip
```

We begin our example by splitting the Commons Pool library and tests into two bundles. The `main` subproject extracts the library source, compiles it, and creates a simple bundle that exports the main package, but hides the implementation (`.impl`) package. The `test` subproject does exactly the same thing for the test source, but appends "-test" to the bundle symbolic name to make sure the bundles are unique.

The Commons Pool tests are jUnit tests, so we also need access to the jUnit library [ref]. Should it be deployed as a bundle or placed on the boot class path? Exposing the packages from the boot class path means we don't have to turn jUnit into a bundle, but it also means jUnit can't see test classes unless they are also on the boot class path or explicitly passed in via a method call. We would have to write our own code to scan bundles for tests and feed the class instances to jUnit, instead of relying on the standard test runner. We'll look at tool that does this later in section 8.3, so let's try the other approach here: bundling jUnit.

We use the "bndwrap" Ant task from the bnd tool [ref] to quickly wrap the JAR file. The bndwrap task analyzes the JAR and creates a bundle that exports all packages contained inside it. It also adds *optional* imports for any packages that are needed, but not contained in the JAR file. Unfortunately this import list won't contain our test packages, because jUnit doesn't know anything about them yet. To avoid having to explicitly list our test packages at build-time, we instead use `DynamicImport-Package: *` (discussed in 5.2.2). This dynamic import means jUnit will be able to see any future test class, as long as some bundle exports it.

We also add the following `Main-Class` header:

```
Main-Class: junit.textui.TestRunner
```

This tells our example launcher to start the jUnit test runner after deploying all the bundles. The `TestRunner` class expects to receive the name of the primary test class, so we add `org.apache.commons.pool.TestAll` to the OSGi launcher command-line in `build.xml`. (Our launcher will automatically pass any arguments after the initial bundle directory setting on to the `Main-Class`.)

Figure 8.2 shows our test deployment, which resembles the fourth option in Figure 8.1:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

{FIGURE}

Figure 8.2 Testing Commons Pool inside an OSGi framework

Let's try it out for real:

```
$ cd chapter08/migration-example

$ ant clean test.osgi
...
[junit.osgi] Class not found "org.apache.commons.pool.TestAll"
[junit.osgi] Java Result: 1
```

Hmm, our jUnit bundle couldn't see the `TestAll` class even though the test bundle clearly exports it. If you look closely at the package involved and cast your mind back to the visibility discussion from 2.5.3 you should understand why. This is the same package that's exported by the main Commons Pool bundle! Remember that packages cannot be split across bundles unless you use bundle dependencies (section 5.3) and we are using package dependencies. We could use `Require-Bundle` to merge the packages together and re-export them (see section 5.3.1 for more about re-exporting packages), but we would then need to use mandatory attributes to make sure jUnit and other related test bundles were correctly wired to the merged package. This would lead to a fragile test structure and cause problems with package-private members (to find out why see the discussion near the start of 5.4.1).

A better solution is to use fragments (section 5.4) to augment the original bundle with the extra test classes. To do this we just need to add one line to `test/build.properties`:

```
Fragment-Host: ${module}                                                   #1
```

The `module` property refers to the `org.apache.commons.pool` package, which we also use as the symbolic name of the main bundle. This is all we need to declare our test bundle as a fragment of the main library bundle (#1). With this change in place we can rebuild and repeat the test. You should see jUnit run through the complete Commons Pool test suite, which takes around two minutes:

```
$ ant clean test.osgi
...
[junit.osgi] .........................................
[junit.osgi] .........................................
[junit.osgi] .........................................
[junit.osgi] .........................................
[junit.osgi] .........................................
[junit.osgi] .....................................
[junit.osgi] Time: 118.127
[junit.osgi]
[junit.osgi] OK (242 tests)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

We are now running all our tests inside the combined library bundle (option 3 from Figure 8.1) because our test fragment contains both internal and public facing tests. We could go one step further and use a plain bundle for public tests and a fragment for internal tests, but we would need some way to give jUnit access to our internal tests. At the moment the public `org.apache.commons.pool.TestAll` class loads internal tests from inside the same fragment, but this won't work when we separate them. We don't want to export any internal packages from the fragment because that would also expose internals from the main bundle, potentially affecting the test results.

The least disruptive solution is to keep a single public test class in the fragment that can be used to load the internal tests. The remaining public facing tests can be moved to a new package that doesn't conflict with the library API (such as `.test`) and deployed in a separate bundle. Figure 8.3 shows an example of such a structure for testing Commons Pool.

{FIGURE}

Figure 8.3 Recommended test structure for OSGi bundle tests

Finally, try running the tests outside of the container, you should see the same results:

```
$ ant clean test
...
[junit] ..........................................
[junit] ..........................................
[junit] ..........................................
[junit] ..........................................
[junit] ..........................................
[junit] ......................................
[junit] Time: 117.77
[junit]
[junit] OK (242 tests)
```

We just saw how easy it is to run tests both inside and outside of a container, but how do we know if we're testing all possible scenarios and edge cases? Most projects use coverage to measure test effectiveness, although this doesn't guarantee you have well-written tests! Given the importance of test coverage, let's continue with our example and find out how we can record coverage statistics inside an OSGi container.

8.1.3 Covering all the bases

It is always good to know how much of your code is being tested. Like test results, coverage can vary depending on whether you're testing inside or outside a container. This makes it just as important to include container-based tests when determining overall test coverage.

We can break the coverage gathering process into three stages:

3. Instrument the classes

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

4. Execute the tests
5. Analyze the results

The first and third stages can be done outside of the OSGi container. This leaves us with the second stage: testing the instrumented classes inside the chosen container. We already know we can run the original tests in OSGi, so what difference does instrumentation make? It obviously introduces some sort of package dependency to the coverage library, but it also introduces a configuration dependency. The instrumented code needs to know where to find the coverage database so it can record results. We can deal with the package dependency in three ways: wrap the coverage JAR file up as a bundle, export its packages from the system bundle using `org.osgi.framework.system.packages.extra`, or expose them from the boot class path with `org.osgi.framework.bootdelegation`. When using boot delegation we must make sure coverage packages are excluded from the generated `Import-Package` in the library bundle or at least made optional. (Not doing this would lead to a missing constraint during resolution, because no bundle exports these packages.)

The simplest approach is to add the coverage JAR file and its dependencies to the launcher's class path and update the system packages. Next simplest is boot delegation, here we have the extra step of removing coverage packages from the `Import-Package` of our instrumented bundle. We're going to take the interesting route and turn the coverage JAR file into a bundle. Our chosen coverage tool for this example is Cobertura 1.9.3 [ref], but all of the techniques mentioned above should work for other tools as well.

First step is to create a new JAR file which contains the original Cobertura JAR file and all of its execution time dependencies. We embed these dependencies because we want this to be a standalone bundle. Remember, this bundle will only be used during testing, so we have more leeway than if we were creating a production quality bundle. We then use the bnd tool to wrap the JAR file in the same way we wrapped jUnit, making sure we set `Bundle-ClassPath` so the bundle can see its embedded dependencies. You can find the complete bundling process in `cobertura.osgi/build.xml`.

All we need to do now is instrument the classes and run the tests:

```
$ ant clean test.osgi "-Dinstrument=true"
```

We use the instrument property to enable the various instrumentation targets. Before launching the tests, the build also sets the `net.sourceforge.cobertura.datafile` system property so that instrumented tests know where to find the coverage database. As soon as the tests complete the build runs the Cobertura report task to process the results. Point your browser at `reports/index.html` to see the results, which should look like Figure 8.4.

{FIGURE}

Figure 8.4 Cobertura coverage report for Commons Pool

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Throughout this section we saw how to take existing tests (and test tools) and run them inside an OSGi container. You may have noticed that this process is very similar to the "JAR to bundle" process described in the first half of chapter 6. In fact, deciding how to bundle tests is really no different from deciding how to bundle an application. Visibility and modularity are just as important when it comes to testing. But what about going the other way? Can we take OSGi-related code and test it outside of the container?

When you first begin to modularize and migrate your application over to OSGi, you probably won't have a direct dependency on the OSGi API itself. This means your code can still be tested both inside and outside of the container. But at some point you will want to use the OSGi API. It may start with one or two bundle activators, then maybe using the bundle context to lookup a service. Dependency injection, component models (discussed in chapter 10), and other similar abstractions can all help reduce the need to deal directly with the container. But what if you have code that uses the OSGi API. Such code cannot be tested outside of the container – or can it?

Imagine if we could mimic the container without having to implement a complete OSGi framework. Well there is a technique for doing this and it goes by the name of mocking.

8.2 Mocking OSGi

OSGi is just a load of fancy class loaders! Oh wait, we didn't mean that sort of mocking. (Besides, we all know by now that there's a lot more to OSGi than enhanced class loading.) We are actually talking about using *mock* objects to test portions of code without requiring a complete system. A mock object is basically a simulation, not a real implementation. It provides the same API, but its methods are scripted and usually return expected values or additional mocked objects. Object mocking is a powerful technique because it lets you test code right from the start of a project, even before your application is complete. You can also use it to test situations that are hard to recreate with the real object, such as external hardware failures. Figure 8.5 shows an example of mocking in action.

{FIGURE}

Figure 8.5 Mocking in action

8.2.1 Testing expected behavior

So how might we use mocking to test an OSGi application? Let's look at code from earlier in this book, specifically the `LogService` lookup example from chapter 4 that contained a potential race condition. Listing 8.1 provides a quick reminder of the problematic code:

Listing 8.1 Broken service lookup containing race condition

```
public class Activator implements BundleActivator {
  BundleContext m_context;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    public void start(BundleContext context) {
      m_context = context;                                                    #1
      startTestThread();
    }

    public void stop(BundleContext context) {
      stopTestThread();
    }

    class LogServiceTest implements Runnable {
      public void run() {
        while (Thread.currentThread() == m_logTestThread) {
          ServiceReference logServiceRef =
              m_context.getServiceReference(LogService.class.getName());   #2

          if (logServiceRef != null) {
            ((LogService)m_context.getService(logServiceRef)).log(         #3
                LogService.LOG_INFO, "ping");
          } else {
            alternativeLog("LogService has gone");
          }

          pauseTestThread();
        }
      }
    }

    //  The rest of this class is just support code...
  }
```

Notice how this code interacts with the OSGi container. It receives a context object in the activator start method (#1), uses this context to get a service reference (#2), and uses this reference to get the actual instance (#3). Each of these objects has a well-defined interface we can mock out, and the example code only uses a few methods from each API. This is good news because when mocking objects you only need to simulate the methods that are actually used, not the complete API.

We already know this code compiles against the OSGi API and back in chapter 4 we even tried it out on an actual framework. But does it use the service API correctly? This is the sort of test which is hard to write without mocking. Sure you can run tests on the container by invoking your code and checking the results as we did back in 8.1, but this doesn't tell you if the code is using the container in the right way. For example, the container won't complain if you forget to “unget” a service after you're done with it, but forgetting to do this skews service accounting and makes it look like your bundle is still using the service when it isn't. The container also doesn't know if you use the result of `getService()` without checking for `null`. In our example you could get a `NullPointerException` if the service disappeared in the short time between checking the reference and using it. Writing a test that's guaranteed to expose this race condition on a live framework is very hard, but trivial with mock objects.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

How exactly does mocking help? Because mock objects are scripted, we can verify that the right methods are called in the appropriate order. We can throw exceptions or return `null` values at any point in the sequence to see how the client handles it. Enough talk, let's actually try mocking ourselves.

8.2.2 Mocking in action

There are typically five steps involved in mocking out an API:

- mock – create prototype mock objects
- expect – script the expected behavior
- replay – prepare the mock objects
- test – run code using the mock objects
- verify – check the behavior matches

We're going to use EasyMock [ref] in this example, but any mocking library would do. You can find our initial setup under `chapter08/mocking-example` in the companion code samples. It contains the log client code from Listing 8.1 and a skeleton test class which we'll expand on during this section, `mock_test/src/org/foo/mock/LogClientTests.java`. You can also find a completed version of the unit test in the `solution` directory if you don't feel like typing out all this code.

Let's go through each of the five steps in detail and mock out the OSGi API:

1. First create prototype objects for parts of the API that we want to mock out; namely `BundleContext`, `ServiceReference`, and `LogService`. You can do this by adding the following lines to the empty test case:

   ```
   BundleContext context = createStrictMock(BundleContext.class);
   ServiceReference serviceRef = createMock(ServiceReference.class);
   LogService logService = createMock(LogService.class);
   ```

   We use a strict mock for the context, because we want to check the call sequence.

2. Script the expected behavior of the log client as it finds and calls the `LogService`:

   ```
   expect(context.getServiceReference(LogService.class.getName()))      #1
       .andReturn(serviceRef);                                          #2

   expect(context.getService(serviceRef))                               #3
       .andReturn(logService);                                          #4

   logService.log(
       and(geq(LogService.LOG_ERROR), leq(LogService.LOG_DEBUG)),       #5
       isA(String.class));                                              #6
   ```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Using our knowledge of the service API from chapter 4, we expect the client will call our mock context to find a reference to the `LogService` (#1), to which we respond by returning a mock service reference (#2). We expect the client to pass this reference back our mock context (#3) in order to get our mock `LogService` (#4). Finally we expect the client to call our mock `LogService` with a valid log level and some sort of message string.

3. Replay the expected behavior to initialize our mock objects:

```
replay(context, serviceRef, logService);
```

4. Now we get to use our mock objects and pretend to be the OSGi container:

```
BundleActivator logClientActivator = new Activator();

logClientActivator.start(context);                                    #1
try {
  Thread.sleep(1000);                                                 #2
} catch (InterruptedException e) {}
logClientActivator.stop(context);                                     #3
```

Consider the active lifecycle of an OSGi bundle: it is first started (#1) and some time later it is stopped (#3). We don't worry about mimicking the resolution stage in this test because we want to test service usage, not class loading. We know the client will spawn some sort of thread to use the `LogService`, so we wait one second (#2) to give that thread time to make the call and pause. (Using `sleep` here is not ideal, later on we'll see how we can replace it with proper handshaking.) Then when our one second is up, we stop the client bundle.

5. The last step is to make sure we saw the expected behavior during the test:

```
verify(context, serviceRef, logService);
```

This method throws an exception if the observed behavior does not match.

At this point you should have a complete test which will compile and run successfully:

```
$ cd chapter08/mocking-example

$ ant test
...
test:
[junit] Running org.foo.mock.LogClientTests
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 1.157 sec
```

Excellent, we've confirmed that our client uses the OSGi API correctly when a `LogService` is available. But what happens when a `LogService` is not available, does it handle that too?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

8.2.3 Mocking unexpected situations

As we mentioned back at the start of this section, mocking is a powerful testing technique because it lets you script situations that are hard to recreate inside a test environment. While it is easy to arrange a test in an OSGi container without a `LogService`, it would be very difficult to arrange for this service to appear and disappear at exactly the right time to trigger the race condition we know exists in our client code. With mocking it is easy.

First, let's test what happens when no `LogService` is available by adding the following expectation between our last `expect` and the call to `replay`:

```
expect(context.getServiceReference(LogService.class.getName()))
    .andReturn(null);
```

This states that we expect the client to begin another call to lookup the `LogService`, but this time we return a `null` reference to indicate no available service. If you try and run the test now it will fail because we don't give the client enough time to make a second call before stopping the bundle. Our log client pauses five seconds between each call, so we just need to add five seconds onto the existing `sleep`:

```
try {
  Thread.sleep(6000);
} catch (InterruptedException e) {}
```

The client now gets enough time to begin a second log call, but the test still fails:

```
$ ant test
...
[junit] Running org.foo.mock.LogClientTests
[junit] Exception in thread "LogService Tester" java.lang.AssertionError:
[junit]   Unexpected method call getBundle():
```

It appears that our client is using another method (`getBundle()`) on the `BundleContext` to find the owning bundle when no `LogService` is available. If you look at the rest of the client code under `chapter08`, you'll see that it uses this to get the bundle id when logging directly to the console. We don't really mind how many times our client calls `getBundle()`, if at all, so lets use a "wildcard" expectation:

```
Bundle bundle = createNiceMock(Bundle.class);                          #1

expect(context.getServiceReference(LogService.class.getName()))
    .andReturn(null);

expect(context.getBundle())
    .andReturn(bundle).anyTimes();                                     #2
```

We need to provide a new mock to represent our `Bundle` object. This time instead of simulating each method the client actually uses, we take a shortcut and use a "nice" mock

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

(#1). Nice mocks automatically provide empty implementations and default return values. We expect our log client to request this mock bundle from our mock bundle context after we return the `null` service reference, but it might ask for it zero or more times (#2). One last thing we must remember to do is add our mock bundle to the replay list. (If you happen to forget to replay a mock before it is used you will get an `IllegalStateException` from EasyMock about missing behavior definitions.)

```
replay(context, serviceRef, logService, bundle);
```

With the new expectation in place and everything replayed, the test passes once more:

```
$ ant test
...
[junit] Running org.foo.mock.LogClientTests
[junit] <--> thread="LogService Tester", bundle=0 : LogService has gone
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 6.125 sec
```

Having `sleep` in our unit test is annoying though. Every time we want to test additional log calls we would need to extend the `sleep`, which makes our tests run longer and longer. We should really try to replace it with some form of handshaking. But even with handshaking, our log client will still pause for five seconds between each call. If only we could replace the pause method while keeping the rest of the code intact.

8.2.4 Coping with multi-threaded tests

We're currently testing a simple log client that spawns a separate thread to make log calls. Knowing how to test multi-threaded bundles is very useful, because people often use threads to limit the amount of work done in the activator's start method. As we mentioned at the end of the last section, the main difficulty is synchronizing the test thread with the threads being tested. Up to now we relied on `sleep`, but this is a fragile solution. Some form of barrier or handshake procedure (Figure 8.6) is needed to hold client threads back until the test is ready to proceed and vice-versa.

{FIGURE}

Figure 8.6 Synchronizing tests with multi-threaded code

Thankfully, the log client has an obvious place where we can add such a barrier: the protected `pauseTestThread` method, which currently puts the client thread to sleep for five seconds. We could consider using aspect-orientated programming to add a barrier to this method, but let's avoid pulling in extra test dependencies and use an anonymous class to override it instead:

```
final CountDownLatch latch = new CountDownLatch(2);                    #1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
BundleActivator logClientActivator = new Activator() {
  @Override protected void pauseTestThread() {
    latch.countDown();                                          #2

    if (latch.getCount() == 0) {                                #3
      LockSupport.park();                                       #4
    }
  }
};
```

Our anonymous class replaces the original `pauseTestThread` method with one that uses a countdown latch, initialized with the number of expected log calls (#1). Each time the client makes a log call it calls `pauseTestThread` and counts down the latch (#2). When no more log calls are expected (#3) the client thread suspends itself and waits for the rest of the test to shutdown (#4). All our test code needs to do is wait for the latch to count down to zero before it stops the client bundle:

```
logClientActivator.start(context);
if (!latch.await(5, TimeUnit.SECONDS)) {                        #1
  fail("Still expecting" + latch.getCount() + " calls");
}
logClientActivator.stop(context);
```

The test includes a timeout (#1) in case the client thread aborts and can't complete the countdown, but if everything goes as expected the updated test will finish in under a second:

```
$ ant test
...
[junit] Running org.foo.mock.LogClientTests
[junit] <--> thread="LogService Tester", bundle=0 : LogService has gone
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.14 sec
```

So far so good, all we have to do to test additional log calls is increment the latch count. But what should we do if our client thread doesn't contain a “pause” method or this method cannot be replaced or extended? Another solution is to add barriers to the mocked out objects themselves by using so-called “answer” objects. Answers let us perform basic AOP by intercepting method calls, which we can use to add synchronization points. For example:

```
expect(context.getServiceReference(isA(String.class)).andAnswer(
    new IAnswer<ServiceReference>() {                           #1
      public ServiceReference answer() {
        LockSupport.park();                                     #2
        return null;
      }
    });
```

In the above (incomplete) example, we script an answer (#1) that always returns a `null` service reference and use it to suspend the client thread whenever it makes this call (#2). This works as long as the client thread initiates the expected call at the right time and there

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

are no problems with suspending the client in the middle of this call. But it also leaves the client code untouched, which for us would mean a five second pause between log calls. We're going to test another log call in the next section, so let's stick with our original latch solution.

8.2.5 Exposing race conditions

OSGi is very dynamic; bundles and services might come and go at any time. The key to developing a robust application is being able to cope with and react to these events. This same dynamism makes testing robustness very difficult. You could deploy your bundles into a real framework and attempt to script events to cover all possibilities (we'll look at this in more detail in 8.3), but some scenarios require micro-second timing. Remember the race condition we mentioned at the start of this section? This would only be exposed if we could arrange for the `LogService` to disappear between two method invocations, a very narrow window. Many factors could cause us to miss this window: unexpected garbage collection, differences in thread scheduling. With mocking we can easily script the exact sequence of events we want:

```
expect(context.getServiceReference(LogService.class.getName()))
    .andReturn(serviceRef);                                          #1

expect(context.getService(serviceRef))
    .andReturn(null);                                                #2

expect(context.getBundle())
    .andReturn(bundle).anyTimes();                                   #3
```

We begin by expecting another log call, so remember to bump the latch count up to three calls. The `LogService` is still available at this point, so we return our mock reference (#1). The client is expected to dereference this by calling `getService()` and it is at this point that we pretend the `LogService` has vanished and return `null` (#2). We follow this by expecting another wildcard call to get the bundle (#3) just as we did in 8.2.3, because the log client might need it to do some alternative logging to the console.

Our test is now complete. You might want to compare it with the class in the `solution` sub-directory. It covers normal and missing service conditions, and the edge case where the service is there to begin with, but quickly disappears. Running it should expose the problem we all know is there, but we weren't able to recreate reliably on a real framework:

```
$ ant test
...
[junit] Running org.foo.mock.LogClientTests
[junit] <--> thread="LogService Tester", bundle=0 : LogService has gone
[junit] Exception in thread "LogService Tester"
          java.lang.NullPointerException
[junit]     at org.foo.log.Activator$LogServiceTest.run(Activator.java:66)
[junit]     at java.lang.Thread.run(Thread.java:619)
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 5.205 sec
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

At this point, adventurous readers might like to copy the working service lookup example from chapter 4 (`chapter04/dynamics/correct_lookup`) and try testing it. One tip: you'll need to extend the test to expect calls to `ungetService`, because the working example attempts to release the service reference after each successful call to `getService`. Whether you mandate calls to `ungetService()` or make them optional by appending "`times(0, 1)`" to the expectation is completely up to you.

In this section, we learned how to mock out the OSGi API and script different scenarios when testing bundle-specific code that uses OSGi. Mocking helps us test situations that are next to impossible to recreate in a real container. It also provides a counterpoint to the first section where we were running existing tests inside a real container on code that often had no dependency on OSGi at all. Our last section will attempt to harmonize both approaches, by explaining how to script modular tests and run them on a variety of frameworks.

8.3 Testing modularity

In the previous section, we successfully mocked out the OSGi API and ran our tests without requiring a framework. Of course, the less you depend directly on an API, the easier it is to mock. It is even easier if you use one of the component models from chapter 10 because your dependencies will be indirectly injected by the component framework. Such components rarely need to use the OSGi API themselves, so testing becomes a matter of reconfiguring bindings to inject mocked out dependencies in place of the original instances. But as we discussed in 8.1.1, eventually you will want to run tests on a real OSGi framework. These container tests typically won't increase your code coverage – any unit and mocked out tests should have already tested the critical paths. Instead, these tests verify that your code conforms to the container: is it packaged correctly, does it follow the container programming model, does it use standard APIs?

You should run your tests on as many containers as possible to guard against container-specific behavior. But keeping all these containers up to date and managing their different settings and configurations soon becomes tiresome. The newly standardized OSGi embedding and launching API (discussed in chapter 11) helps, but it lacks features that would make testing on OSGi much easier: automatic test wrapping, dynamic bundle creation, common deployment profiles. Luckily for us, there are several recently released OSGi test tools that provide all these features and more.

OSGi-enabled test tools bring other benefits because they embrace OSGi, such as improved test modularity and management. You can use them to run a complete range of tests from basic unit tests, through various combinations of integration tests, all the way up to advanced management tests. We'll see a real-world example of this later on and explain exactly what we mean by management testing, but first let's see what OSGi test tools are available right now.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

8.3.1 OSGi test tools

At the time of writing this book, there are three major test tools available for OSGi:

- OPS4J's Pax-Exam [ref]
- Spring DM's test support [ref]
- Dynamic Java's DA-Testing [ref]

All follow the same basic approach to building and deploying tests:

- Prepare the OSGi container
- Deploy the selected bundles
- Create test bundle on the fly
- Deploy and execute the tests
- Shutdown the container

Each tool has its own advantages and disadvantages. The Spring-DM test support obviously works best with Spring based applications. While you can also use it to test non-Spring applications, it requires several Spring dependencies which make it appear rather heavy. Spring-DM testing also only supports jUnit 3, which means no annotated tests. DA-Testing on the other hand provides its own test API, optimized for testing service dynamics such as the race condition we saw in 8.2.5. This makes it hard to move existing jUnit or TestNG tests over to DA-Testing, since developers have to learn another test API, but it does make dynamic testing much easier. Pax-Exam goes to the other extreme and supports both jUnit 3 and 4, with TestNG support in the works. Table 8.1 summarizes the differences between the tools:

Test Tool	jUnit3	jUnit4	TestNG	OSGi Mocks	OSGi Frameworks	OSGi Profiles
Pax-Exam	X	X	future		Felix / Equinox / KF (multiple versions)	over 50
Spring-DM	X	future		X	Felix / Equinox / KF (single version only)	
DA-Testing					Equinox (others planned)	

Table 8.1 OSGi Test Tool Features

We are going to use Pax-Exam from the OPS4J community because we believe it's a good general purpose solution, but a lot of the techniques covered in this section can also be adapted for use with the other tools. One of Pax-Exam's strengths is its support for a wide

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

range of different OSGi frameworks, which is really important if you want to produce robust portable bundles. But why is this?

8.3.2 Running tests on multiple frameworks

OSGi is a standard, with a detailed specification and a set of framework compliance tests. Even with all of this there can be subtle differences between implementations. Perhaps part of the specification is unclear or is open to interpretation. On the other hand, maybe your code relies on behavior that isn't part of the specification and is left open to framework implementers, such as the default Thread Context Class Loader (TCCL) setting. The only way to make sure your code is truly portable is to run the same tests on different frameworks. This is just like the practice of running tests on different operating systems – even though the JDK is supposed be be portable and standardized, differences can exist and it is better to catch them during development than fix problems in the field.

Unfortunately, a lot of OSGi developers only test against a single framework. This might be because they only expect to deploy their bundles on that particular implementation, but it is more likely that they believe the cost of setting up and managing multiple frameworks far outweighs the perceived benefits. This is where Pax-Exam steps in – it makes testing on an extra OSGi framework as simple as adding a single line of Java code.

Let's see for ourselves how easy it to use Pax-Exam. We're going to continue to use Ant to run these tests, although Pax-Exam is primarily Maven based. This means we need to explicitly list execution-time dependencies in our `build.xml`, instead of letting Maven manage this for us. You can find our initial setup under `chapter08/testing-example`.

Take a look at the `fw` subproject, it contains a very simple test class that prints out various framework properties. The contents of this test class are shown below in Listing 8.2.

Listing 8.2 Simple container test

```
@RunWith(JUnit4TestRunner.class)                                         #1
public class ContainerTest {

  @Configuration                                                         #2
  public static Option[] configure() {
    return options(
      mavenBundle("org.osgi", "org.osgi.compendium", "4.2.0")            #3
    );
  }

  @Test
  public void testContainer(BundleContext ctx) {                         #4
    System.out.println(
      format(ctx, FRAMEWORK_VENDOR) +
      format(ctx, FRAMEWORK_VERSION) +
      format(ctx, FRAMEWORK_LANGUAGE) +
      format(ctx, FRAMEWORK_OS_NAME) +
      format(ctx, FRAMEWORK_OS_VERSION) +
      format(ctx, FRAMEWORK_PROCESSOR) +
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
        "\nTest Bundle is " +
        ctx.getBundle().getSymbolicName());                               #5
  }

  private static String format(
      BundleContext ctx, String key) {

    return String.format("%-32s = %s\n",
        key, ctx.getProperty(key));
  }
}
```

We begin by annotating our test class with `@RunWith` (#1). This tells jUnit to use the named test runner instead of the standard jUnit one. The Pax-Exam `JUnit4TestRunner` class is responsible for starting the relevant framework, deploying bundles, and running the tests. The `@Configuration` (#2) annotation identifies the method that provides the Pax-Exam configuration. Right now we just ask it to deploy the standard OSGi compendium bundle (#3) from Maven central on to the default framework. The actual test method is annotated with the usual jUnit4 annotation, `@Test`. It accepts a `BundleContext` argument (#4) which will be supplied by Pax-Exam at execution time. We use this bundle context to print out various properties, including the symbolic name of the test bundle (#5).

To run this test, type the following:

```
$ cd chapter08/testing-example

$ ant test.container
```

You should see something like Listing 8.3, but with properties that match your system.

Listing 8.3 Using Pax-Exam to run tests on an OSGi framework

```
[junit] Running org.foo.test.ContainerTest
[junit] __________                 ___________
[junit] \______   \_____  ___  ___ \_   _____/__  ________    _____
[junit]  |     ___/\__  \ \  \/  /  |    __)_\  \/  /\__  \  /     \
[junit]  |    |     / __ \_>    <   |        \>    <  / __ \|  Y Y  \
[junit]  |____|    (____  /__/\_ \ /_______  /__/\_ \(____  /__|_|  /
[junit]                 \/      \/         \/      \/     \/      \/
[junit]
[junit] Pax Exam 1.1.0 from OPS4J - http://www.ops4j.org
[junit] ------------------------------------------------
[junit]
[junit]
[junit]
[junit] Welcome to Felix
[junit]
[junit] ================
[junit]
[junit] org.osgi.framework.vendor        = Apache Software Foundation
[junit] org.osgi.framework.version       = 1.5
[junit] org.osgi.framework.language      = en
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
[junit] org.osgi.framework.os.name       = windowsvista
[junit] org.osgi.framework.os.version    = 6.0
[junit] org.osgi.framework.processor     = x86
[junit]
[junit] Test Bundle is pax-exam-probe
[junit]
[junit]
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 3.424 sec
```

You might have noticed that the symbolic name of the test bundle is “pax-exam-probe”. This bundle is generated at execution time by Pax-Exam and contains our test classes. The default container is Apache Felix, but we can easily ask Pax-Exam to run the same test on other frameworks as well. All we need to do is add a few lines to the configuration method in our test class `fw/container/src/org/foo/test/ContainerTest.java`:

```
@Configuration
public static Option[] configure() {
  return options(
    frameworks(
      felix(), equinox(), knopflerfish()
    ),
    mavenBundle("org.osgi", "org.osgi.compendium", "4.2.0")
  );
}
```

Pax-Exam will do the hard work of downloading the necessary JAR files and setting up any framework specific configuration files. We just need to sit back and re-run our test:

```
$ ant test.container
```

This time you should see three distinct sets of output as shown in Listing 8.4.

Listing 8.4 Using Pax-Exam to run tests on multiple frameworks

```
[junit] Running org.foo.test.ContainerTest
[junit] __________                 ___________
[junit] \______   \_____  ___  ___ \_   _____/__  ________    _____
[junit]  |     ___/\__  \ \  \/  /  |    __)_\  \/  /\__  \  /     \
[junit]  |    |     / __ \_>    <   |        \>    <  / __ \|  Y Y  \
[junit]  |____|    (____  /__/\_ \ /_______  /__/\_ \(____  /__|_|  /
[junit]                 \/      \/         \/      \/     \/      \/
[junit]
[junit] Pax Exam 1.1.0 from OPS4J - http://www.ops4j.org
[junit] ------------------------------------------------
[junit]
[junit]
[junit]
[junit] Welcome to Felix
[junit] ================
[junit]
[junit] org.osgi.framework.vendor        = Apache Software Foundation
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
[junit] org.osgi.framework.version       = 1.5
[junit] org.osgi.framework.language      = en
[junit] org.osgi.framework.os.name       = windowsvista
[junit] org.osgi.framework.os.version    = 6.0
[junit] org.osgi.framework.processor     = x86
[junit]
[junit] Test Bundle is pax-exam-probe
[junit]
[junit]
[junit] __________                 ___________
[junit] \______   \_____  ___  ___ \_   _____/__  ________    _____
[junit]  |     ___/\__  \ \  \/  /  |    __)_\  \/  /\__  \  /     \
[junit]  |    |     / __ \_>    <   |        \>    <  / __ \|  Y Y  \
[junit]  |____|    (____  /__/\_ \ /_______  /__/\_ \(____  /__|_|  /
[junit]                 \/      \/         \/      \/     \/      \/
[junit]
[junit] Pax Exam 1.1.0 from OPS4J - http://www.ops4j.org
[junit] ------------------------------------------------
[junit]
[junit]
[junit] org.osgi.framework.vendor        = Eclipse
[junit] org.osgi.framework.version       = 1.5.0
[junit] org.osgi.framework.language      = en
[junit] org.osgi.framework.os.name       = WindowsVista
[junit] org.osgi.framework.os.version    = 6.0.0
[junit] org.osgi.framework.processor     = x86
[junit]
[junit] Test Bundle is pax-exam-probe
[junit]
[junit]
[junit] __________                 ___________
[junit] \______   \_____  ___  ___ \_   _____/__  ________    _____
[junit]  |     ___/\__  \ \  \/  /  |    __)_\  \/  /\__  \  /     \
[junit]  |    |     / __ \_>    <   |        \>    <  / __ \|  Y Y  \
[junit]  |____|    (____  /__/\_ \ /_______  /__/\_ \(____  /__|_|  /
[junit]                 \/      \/         \/      \/     \/      \/
[junit]
[junit] Pax Exam 1.1.0 from OPS4J - http://www.ops4j.org
[junit] ------------------------------------------------
[junit]
[junit]
[junit] Knopflerfish OSGi framework, version 4.1.10
[junit] Copyright 2003-2009 Knopflerfish. All Rights Reserved.
[junit]
[junit] See http://www.knopflerfish.org for more information.
[junit]
[junit] Loading xargs url file:knopflerfish/config.ini
[junit] Installed and started:
    file:bundles/org.ops4j.pax.exam_1.1.0.jar (id#1)
[junit] Installed and started:
    file:bundles/org.ops4j.pax.exam.junit.extender_1.1.0.jar (id#2)
[junit] Installed and started:
    file:bundles/org.ops4j.pax.exam.junit.extender.impl_1.1.0.jar (id#3)
[junit] Installed and started:
    file:bundles/org.ops4j.pax.url.dir_1.0.0.jar (id#4)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
[junit] Installed and started:
    file:bundles/com.springsource.org.junit_4.4.0.jar (id#5)
[junit]
[junit] Installed and started:
    file:bundles/org.ops4j.pax.exam.rbc_1.1.0.jar (id#6)
[junit] Installed and started:
    file:bundles/osgi.cmpn_4.2.0.200908310645.jar (id#7)
[junit] Framework launched
[junit] org.osgi.framework.vendor        = Knopflerfish
[junit] org.osgi.framework.version       = 1.3
[junit] org.osgi.framework.language      = en
[junit] org.osgi.framework.os.name       = Windows Vista
[junit] org.osgi.framework.os.version    = 6.0
[junit] org.osgi.framework.processor     = x86
[junit]
[junit] Test Bundle is pax-exam-probe
[junit]
[junit]
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 12.513 sec
```

Notice how some of the properties vary slightly between each framework, in particular the OS name. This is a reminder why it is a good idea to test on a variety of frameworks, to make sure you are not depending on unspecified or undocumented behavior.

We just saw how easy it was to run a test on many different frameworks using Pax-Exam. But how well does it work with existing unit tests and existing test tools?

8.3.3 Unit testing

At the start of this section we mentioned how OSGi test tools can help us modularize and manage tests. Because Pax-Exam integrates with jUnit as a custom runner, it can be used in any system that can run jUnit tests. This means you can mix non-OSGi unit and integration tests with Pax-Exam based tests and have the results collected in one place. A good example of this mixture can be found in the Configuration Admin service implementation from the Apache Felix project [ref]. Configuration Admin is a compendium service that provides and persists configuration data for bundles.

The Felix Configuration Admin build uses Maven and has a single test directory. This test directory contains both mocked-out unit tests that test internal details, along with Pax-Exam integration tests that test the expected Configuration Admin behavior. We've taken these tests and separated them out into unit and integration tests so you can see the difference. The unit tests are in the `ut` subproject and you can run them with this command:

```
$ ant test.unit
...
[junit] Running
    org.apache.felix.cm.file.FilePersistenceManagerConstructorTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.027 sec
[junit] Running org.apache.felix.cm.file.FilePersistenceManagerTest
[junit] Tests run: 8, Failures: 0, Errors: 0, Time elapsed: 0.255 sec
[junit] Running org.apache.felix.cm.impl.CaseInsensitiveDictionaryTest
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
[junit] Tests run: 10, Failures: 0, Errors: 0, Time elapsed: 0.012 sec
[junit] Running org.apache.felix.cm.impl.ConfigurationAdapterTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 0.013 sec
[junit] Running org.apache.felix.cm.impl.ConfigurationManagerTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 0.037 sec
[junit] Running org.apache.felix.cm.impl.DynamicBindingsTest
[junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0.055 sec
```

These are still considered unit tests because they don't run inside an OSGi container. We could bundle them up into a fragment like we did in the first section and deploy them using Pax-Exam, in which case they would be called bundle tests. Bundle tests are somewhere between unit and full-blown integration tests. They test more than a single class or feature, but don't involve more than one bundle. Figure 8.7 shows the difference:

{FIGURE}

Figure 8.7 Unit, bundle, and integration testing

Once you have tested your core functionality both inside and outside the OSGi container you can move onto integration testing. Integration testing is where Pax-Exam really shines.

8.3.4 Integration testing

Integration tests are where you start to piece your application together and test interactions between individual components. In order to test combinations of components, you need some way to compose them. For standard Java applications it can be tricky deciding which JAR files you need to load, but with OSGi applications all the dependency information is available in the metadata. Deployment becomes a simple matter of picking a set of bundles.

Let's look at a concrete example. You can find the Apache Felix Configuration Admin integration tests under the `it` subproject. To run all of these tests in sequence, type:

```
$ ant test.integration
...
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 26.523 sec
...
[junit] Tests run: 7, Failures: 0, Errors: 0, Time elapsed: 24.664 sec
...
[junit] Tests run: 15, Failures: 0, Errors: 0, Time elapsed: 55.839 sec
...
[junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 14.45 sec
...
[junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 13.809 sec
...
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 5.723 sec
```

You might be wondering why there isn't much output during the tests. This is because we've set the local logging threshold to `WARN`. To see more details about what Pax-Exam is running, edit the local `log4j.properties` file and change the threshold from `WARN` to `INFO`.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Let's take a closer look at one of the integration tests from `ConfigurationBaseTest`:

Listing 8.5 Basic configure then start integration test

```
@Test                                                                    #1
public void test_basic_configuration_configure_then_start()
  throws BundleException, IOException
{
  final String pid = "test_basic_configuration_configure_then_start";
  final Configuration config = configure( pid, null, true );             #2

  bundle = installBundle( pid, ManagedServiceTestActivator.class );      #3
  bundle.start();
  delay();                                                               #4

  final ManagedServiceTestActivator tester =
      ManagedServiceTestActivator.INSTANCE;

  TestCase.assertNotNull( tester.props );                                #5
  TestCase.assertEquals( pid, tester.props.get(
      Constants.SERVICE_PID ) );
  TestCase.assertNull( tester.props.get(
      ConfigurationAdmin.SERVICE_FACTORYPID ) );
  TestCase.assertNull( tester.props.get(
      ConfigurationAdmin.SERVICE_BUNDLELOCATION ) );
  TestCase.assertEquals( PROP_NAME, tester.props.get( PROP_NAME ) );
  TestCase.assertEquals( 1, tester.numManagedServiceUpdatedCalls );

  config.delete();                                                       #6
  delay();

  TestCase.assertNull( tester.props );
  TestCase.assertEquals( 2, tester.numManagedServiceUpdatedCalls );      #7
}
```

This integration test checks that the Configuration Admin implementation successfully records configuration data that is registered before the managed bundle starts. The managed bundle is the bundle being configured. The test method has the standard jUnit4 annotation (#1) and extends a base class called `ConfigurationTestBase` which provides general helper methods. One such method is used to set configuration data using the current `ConfigurationAdmin` service (#2). The test creates and installs a managed bundle on the fly (#3) and waits for the configuration to be delivered to this managed bundle (#4). It makes sure the delivered configuration is correct (#5) before removing the configuration (#6). The test waits for the managed bundle to be notified about this removal and verifies it was correctly notified (#7).

This is a very clear test. It almost looks like a unit test except that calls are being made between components instead of inside a single component or class. The other tests under the `it` subproject follow the same basic pattern, which may be repeated several times:

- Check the initial system state

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

- Disrupt the state (by calling services, adding or removing bundles)
- Check the resulting system state

As we saw just, the Configuration Admin integration tests all extend a single base class called `ConfigurationTestBase` that defines helper methods to deal with configurations, synchronize tests, and create additional bundles at execution time. These additional bundles consume and validate the configuration data. Right now the tests are only configured to run on Apache Felix, but let's see if they also pass on other frameworks.

Add the following lines to the Pax-Exam options inside the `configuration()` method in `ConfigurationTestBase`, just like we did with the container test back in section 8.3.2:

```
CoreOptions.frameworks(
  CoreOptions.felix(), CoreOptions.equinox(), CoreOptions.knopflerfish()
),
```

Pax-Exam will now run each test three times – once per framework:

```
$ ant test.integration
...
[junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 84.585 sec
...
[junit] Tests run: 21, Failures: 0, Errors: 0, Time elapsed: 99.05 sec
...
[junit] Tests run: 45, Failures: 0, Errors: 0, Time elapsed: 220.184 sec
...
[junit] Tests run: 12, Failures: 0, Errors: 0, Time elapsed: 55.269 sec
...
[junit] Tests run: 12, Failures: 0, Errors: 0, Time elapsed: 54.686 sec
...
[junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 26.417 sec
```

No failures or errors! The Apache Felix Configuration Admin implementation works the same on all three frameworks. This should not be unexpected, because one of the goals driving OSGi is reusable modules. In fact, many framework bundles can be reused on other frameworks. So when you find you need a particular compendium service and your current framework doesn't provide it, take a look around in case you can reuse a bundle from another site. You could even use Pax-Exam to try out different combinations of compendium bundles.

Pax-Exam makes integration testing as simple as unit testing, but like any good tool you have to be careful not to overuse it. Each integration test has the overhead of starting and stopping an OSGi container, so the overall test time can soon build up as you add more and more tests. People are looking into re-using containers during testing, but for some tests you need complete isolation. So, while work is being done to reduce the cost of each test, it will never be zero. In practice this means you should look carefully at your tests and try to get the most from each one.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Integration testing is normally considered the last phase of testing before starting system or acceptance tests. You've tested each piece of functionality separately and tested they work together. There's nothing else to test before verifying your application meets the customers' requirements or is there?

8.3.5 Management testing

This book contains a whole chapter on how to manage OSGi applications, so it is clear that management is an important aspect. We should reflect that by testing applications to make sure they can be successfully managed, upgraded, and restarted before releasing them into production. Too often we see bundles that work perfectly until they are restarted or bundles that cannot be upgraded without causing ripples that affect the whole application.

So what might management testing cover? Table 8.2 has some suggestions:

Task	Involves
Install	Installing new bundles (or features) alongside existing used implementations
Uninstall	Uninstalling old bundles (or features) that may or may not have replacements
Upgrade	Upgrading one or more bundles with new functionality or bug fixes
Downgrade	Downgrading one or more bundles because of an unexpected regression
Graceful degradation	See how long the application functions as elements are stopped or uninstalled

Table 8.2 Management testing ideas

We're going to show you how OSGi and Pax-Exam can help with management testing. Our current test example exercises the latest Configuration Admin implementation from Apache Felix. But what if you have an application that uses an earlier version, can you upgrade to the new edition without losing any configuration data? Why not write a quick test to find out!

You can find our example upgrade test under `mt/upgrade_configadmin_bundle`. It is based on the `listConfiguration` test from the existing Apache Felix integration test suite. Listing 8.6 shows the custom configuration for our upgrade test. We want to re-use the helper classes from the earlier tests, so we explicitly deploy the integration test bundle alongside our management test (#1). We also deploy the old Configuration Admin bundle (#2) and store the location of the new bundle in a system property (#3) so we can use it later to upgrade Configuration Admin during the management test. We use a system property because the configuration and test methods are executed by different processes, and system properties are a cheap way to communicate between processes.

Listing 8.6 Configuring the upgrade management test

```
private static String toFileURI(String path) {
  return new File(path).toURI().toString();
}

@org.ops4j.pax.exam.junit.Configuration
public static Option[] configuration() {
  return options(
    provision(
      bundle(toFileURI("bundles/integration_tests-1.0.jar")),             #1
      bundle(toFileURI("bundles/old.configadmin.jar")),                   #2
      mavenBundle("org.osgi", "org.osgi.compendium", "4.2.0"),
      mavenBundle("org.ops4j.pax.swissbox", "pax-swissbox-tinybundles",
                  "1.0.0")
    ),
    systemProperty("new.configadmin.uri").
      value(toFileURI("bundles/configadmin.jar"))                         #3
  );
}
```

The rest of the test follows the same script as the original `listConfiguration` test with three key differences. First, we make sure that the installed Configuration Admin bundle is indeed the older 1.0.0 release by checking the OSGi metadata:

```
Dictionary headers = getCmBundle().getHeaders();
TestCase.assertEquals("org.apache.felix.configadmin",
    headers.get(Constants.BUNDLE_SYMBOLICNAME));
TestCase.assertEquals("1.0.0",
    headers.get(Constants.BUNDLE_VERSION));
```

Second, we do an in-place update of the Configuration Admin bundle to the new edition:

```
cmBundle.update(new URL(
    System.getProperty("new.configadmin.uri")).openStream());
```

We perform an in-place update to preserve the existing configuration data in the bundle's persistent data area (3.3.4). This only works when upgrading bundles to a new version, if we wanted to switch to a Configuration Admin implementation from another vendor then we would need both bundles installed while we copied the configuration data between them.

Third, we make sure the Configuration Admin bundle was successfully updated to the new version before finally checking that the configuration data still exists:

```
headers = cmBundle.getHeaders();
TestCase.assertEquals("org.apache.felix.configadmin",
    headers.get(Constants.BUNDLE_SYMBOLICNAME));
TestCase.assertEquals("1.2.7.SNAPSHOT",
    headers.get(Constants.BUNDLE_VERSION));
```

You can run this management test with a single command:

```
$ ant test.management
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
[junit] Running org.apache.felix.cm.integration.mgmt.ConfigAdminUpgradeTest
...
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 5.344 sec
```

You can even extend the upgrade test to make sure it works on other OSGi frameworks, like we did with the original Apache Felix Configuration Admin integration tests. You'll see the test passes on all three frameworks, which is more proof that this service implementation is truly independent of the underlying OSGi framework.

This was only a small test, but if you look at the management examples from chapters 3 and 7, hopefully you can see that you could easily script larger, more complex scenarios in Java (or any other JVM language) by using the standard OSGi lifecycle and deployment APIs. Imagine building up a modular library of management actions (install, start, stop, upgrade, downgrade) which you can quickly tie together to test a particular task. Such management testing can help squash potential problems well in advance, minimizing real-world downtime.

Earlier on in this chapter we showed you how to test an application all the way up from individual classes, to single bundles, and combinations of bundles. Just now we looked at testing different management strategies, such as upgrading and downgrading components, to make sure the application as a whole (and not just this release) continues to behave over its lifetime. At this point you should be ready to move onto system and acceptance tests. These tests do not need special treatment regarding OSGi, because OSGi is just an implementation detail. As long as the application can be launched, it can be tested.

8.4 Summary

This chapter covered three different approaches to testing OSGi applications:

- bundling existing non-OSGi tests to run inside OSGi
- mocking existing OSGi tests to run outside of OSGi
- using OSGi test tools to automate test deployment

In an ideal world you would use a combination of these three approaches to test all of your code, both inside and outside of one or more OSGi containers. In the real world, projects have deadlines and developers need their sleep, so we suggest using tools such as Pax-Exam to automate as much of the test bundling and deployment work as possible. These tests should grow along with your application, giving you confidence that you do indeed have a robust, modular application. But what should you do if one of your tests fails inside OSGi, what tools and techniques can you apply to find the solution? Help is available in chapter 9.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

9
Debugging Applications

We just learned how to test individual bundles and application deployments in OSGi, but what should you do when an integration test unexpectedly fails with a class loading exception or a load test runs out of memory? If you were working on a classic Java application, you would break out the debugger, start adding or enabling instrumentation, and capture various diagnostic dumps. Well, an OSGi application is still a Java application, so you can continue to use many of your well-honed debugging techniques. The key area to watch out for is usually related to class loading, but that's not the only pitfall.

OSGi applications can have multiple versions of the same class running at the same time requiring greater awareness of versioning, missing imports can lead to groups of classes that are incompatible with other groups, and dangling services can lead to unexpected memory leaks when updating bundles. In this chapter, we'll show you how to debug examples of all these problems and suggest best practices based on our collective experience of working with real-world OSGi applications in the field.

Let's kick off with something simple. Say we have an application composed of many working bundles and one misbehaving bundle, how do we find the bad bundle and debug it?

9.1 Debugging bundles

Applications continue to grow over time – more and more features get built on top of existing functionality; each code change can introduce errors, expose latent bugs, or break original assumptions. In a properly modularized OSGi application, this should only lead to a few misbehaving bundles rather than a completely broken application. If you can identify these bundles you can decide whether to remove or replace them, potentially fixing the application without having to restart it. But first you need to find out which bundles are broken!

Take our paint example, imagine we get a request to allow users to pick the colors of shapes. Our first step might be to add a `setColor()` method to our `SimpleShape` interface:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
/**
 * Change the color used to shade the shape.
 *
 * @param color The color used to shade the shape.
 **/
public void setColor(Color color);
```

You probably think adding a method to an API is a minor, backwards-compatible change, but in this case the interface is implemented by various client bundles that we may not have control over. In order to compile against the new `SimpleShape` API they need to implement this method, so from their perspective this is actually a major change. We should therefore increment the API version in the main paint example "build.xml" file to reflect this. The last version we used was 5.0, so the new version will be:

```
<property name="version" value="6.0"/>
```

We now need to implement the `setColor()` method in each of the three shape bundles. Listing 9.1 shows the updated implementation for the triangle shape bundle:

Listing 9.1 Implementing the setColor() method for the triangle shape

```
public class Triangle implements SimpleShape {

  Color m_color = Color.GREEN;                                              #A

  public void draw(Graphics2D g2, Point p) {
    int x = p.x - 25;
    int y = p.y - 25;
    GradientPaint gradient =
        new GradientPaint(x, y, m_color, x + 50, y, Color.WHITE);         #B
    g2.setPaint(gradient);
    int[] xcoords = { x + 25, x, x + 50 };
    int[] ycoords = { y, y + 50, y + 50 };
    GeneralPath polygon =
        new GeneralPath(GeneralPath.WIND_EVEN_ODD, xcoords.length);
    polygon.moveTo(x + 25, y);
    for (int i = 0; i < xcoords.length; i++) {
      polygon.lineTo(xcoords[i], ycoords[i]);
    }
    polygon.closePath();
    g2.fill(polygon);
    BasicStroke wideStroke = new BasicStroke(2.0f);
    g2.setColor(Color.black);
    g2.setStroke(wideStroke);
    g2.draw(polygon);
  }

  public void setColor(Color color) {
    m_color = color;                                                        #C
  }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

#A remember assigned color
#B apply color to gradient
#C update assigned color

The paint frame bundle contains another implementation of the `SimpleShape` API: `org.foo.paint.DefaultShape`. This class lazily delegates to the real shape via the OSGi service registry, so it also needs to implement the new `setColor()` method. The correct implementation would follow the same approach used in `DefaultShape.draw()`. Namely: check that we have access to the real shape from the registry and, if we don't, request it. We're going to use a broken implementation instead and assume we always have access to the shape instance:

```
public void setColor(Color color) {
  m_shape.setColor(color);
}
```

This sort of mistake could be made by a new team member who doesn't know about the lazy delegation approach and assumes that `m_shape` has been initialized elsewhere. If the application just happened to call `draw()` early on then this bug could go unnoticed for a long time because `m_shape` would already be valid by the time the code reached `setColor()`. But one day, someone might reasonably change the application so it calls `setColor()` first, like in Listing 9.2 from the `ShapeComponent` class, and the bug will bite. (This example may seem a little contrived, but it's surprisingly hard to write bad code when you really want to!)

Listing 9.2 Triggering the missing initialization bug

```
protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;
  g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                      RenderingHints.VALUE_ANTIALIAS_ON);
  SimpleShape shape = m_frame.getShape(m_shapeName);
  shape.setColor(getForeground());                                    #A
  shape.draw(g2, new Point(getWidth() / 2, getHeight() / 2));
}
```

#A setColor called before draw

We now have our broken OSGi application, which will throw an exception whenever you try to paint shapes. Let's see if we can debug it using the JDK provided debugger, jdb [ref].

9.1.1 Debugging in action

The "Java Debugger" (also known as jdb) is a simple debugging tool that primarily exists as an example application for the Java Platform Debugger Architecture (JPDA) rather than a product in its own right. This means it lacks some of the polish and user-friendly features

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

found in most other debuggers. But jdb is still a useful tool, especially when debugging on production servers which have limited installation environments.

Debugging with JDB

We first need to build our broken example, once that is done we can start jdb:

```
$ cd chapter09/debugging-bundles

$ ant dist

$ jdb -classpath launcher.jar launcher.Main bundles

Initializing jdb ...
>
```

Jdb starts up, but it won't launch our application until we type "run":

```
> run
run launcher.Main bundles
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started:
>
```

You should see the updated paint window appear, just like Figure 9.1. All we had to do is call "jdb" instead of "java" and specify the class path and main class (jdb doesn't support the `-jar` option). We didn't have to tell jdb anything about our bundles or the OSGi framework; from jdb's perspective this is just another Java application. A quick side note: if you happen to see several I/O exceptions mentioning the "felix-cache", check that you haven't got any leftover debugged Java processes running. When you forcibly quit jdb using "Ctrl-C" it can

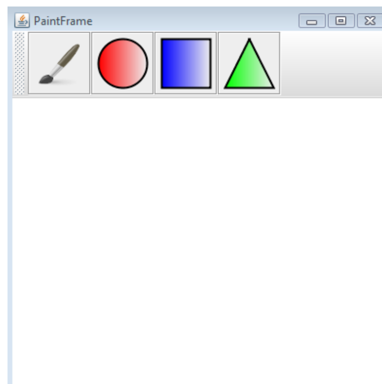


Figure 9.1 Updated paint example running under jdb

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

sometimes leave the debugged process running in the background, which in this case will stop new sessions from using the local “felix-cache” directory.

If you try to draw a shape in the paint window, jdb will report an uncaught exception in the AWT event thread:

```
Exception occurred: java.lang.NullPointerException (uncaught)
    "thread=AWT-EventQueue-0", java.awt.EventDispatchThread.run(),
        line=156 bci=152

AWT-EventQueue-0[1] where

  [1] java.awt.EventDispatchThread.run (EventDispatchThread.java:156)
```

This exception has percolated all the way up to the top of the AWT event thread and jdb doesn't give us an easy way to see where it was originally thrown. We can ask it to stop the application when this sort of exception occurs again, like so:

```
AWT-EventQueue-0[1] catch java.lang.NullPointerException

Set all java.lang.NullPointerException

AWT-EventQueue-0[1] resume

All threads resumed.
```

As soon as the application is resumed you will see an exception stack trace appear on the jdb console. This is not a new exception, it is simply the AWT thread printing out the original (uncaught) exception. The top of the exception stack confirms it was caused by our faulty code inside `DefaultShape`, which we know is contained inside the paint frame bundle. Notice that jdb doesn't give us a way to correlate the exception location with a particular JAR file. What if we didn't know which bundle contained this package? We could try to locate it using the console, but most framework consoles only let you see exported packages. For internal packages we would have to come up with a list of candidate bundles by manually checking the content of each bundle and comparing the exception location with the appropriate source. As we shall see in a moment, tracking a problem down to a specific bundle is much easier when you use an OSGi-aware debugger, such as the Eclipse debugger.

Returning to our broken example, try to paint again. Jdb will now detect and report the exception at the point at which it is thrown inside `setColor()`, but because we haven't attached any source files it won't show us the surrounding Java code:

```
Exception occurred: java.lang.NullPointerException
    (to be caught at: javax.swing.JComponent.paint(), line=1,043 bci=351)
        "thread=AWT-EventQueue-0", org.foo.paint.DefaultShape.setColor(),
            line=126 bci=5

AWT-EventQueue-0[1] list
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
Source file not found: DefaultShape.java
```

No problem, we just need to attach our local source directory:

```
AWT-EventQueue-0[1] use org.foo.paint/src

AWT-EventQueue-0[1] list

122         g2.drawImage(m_icon.getImage(), 0, 0, null);
123       }
124
125       public void setColor(Color color) {
126 =>      m_shape.setColor(color);
127       }
128     }
```

When we print the current value of `m_shape`, we can finally see why it failed:

```
AWT-EventQueue-0[1] print m_shape

 m_shape = null
```

If you're an experienced Java programmer this should all be very familiar, no special OSGi knowledge was required. But take another look at the command where we attached our source directory:

```
use org.foo.paint/src
```

This command has no knowledge of bundles or class versions, it merely provides a list of candidate source files for jdb to compare to debugged classes. Jdb only allows one version of a given source file to be used at any one time, which makes life difficult when debugging an OSGi application containing multiple bundle versions. You have to know which particular collection of source directories to enable for each debug session.

Debugging with Eclipse

Thankfully this is merely a limitation of jdb. If you use an IDE, such as Eclipse, which knows that multiple versions of a class can co-exist in the same JVM, then you don't have to worry about which source relates to which bundle version. The IDE manages that association for you as you debug your application. To see this in action generate Eclipse project files for the two paint examples from chapters 4 and 9:

```
$ cd ../../chapter04/paint-example

$ ant clean pde

$ cd ../../chapter09/debugging-bundles

$ ant clean pde
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Now import these two directories into Eclipse as existing projects. You should end up with ten new projects; half marked as version 4, the rest as version 6. To debug these bundles in Equinox click on the drop-down arrow next to the bug icon (circled at the top of Figure 9.2) and select "Debug Configurations…".

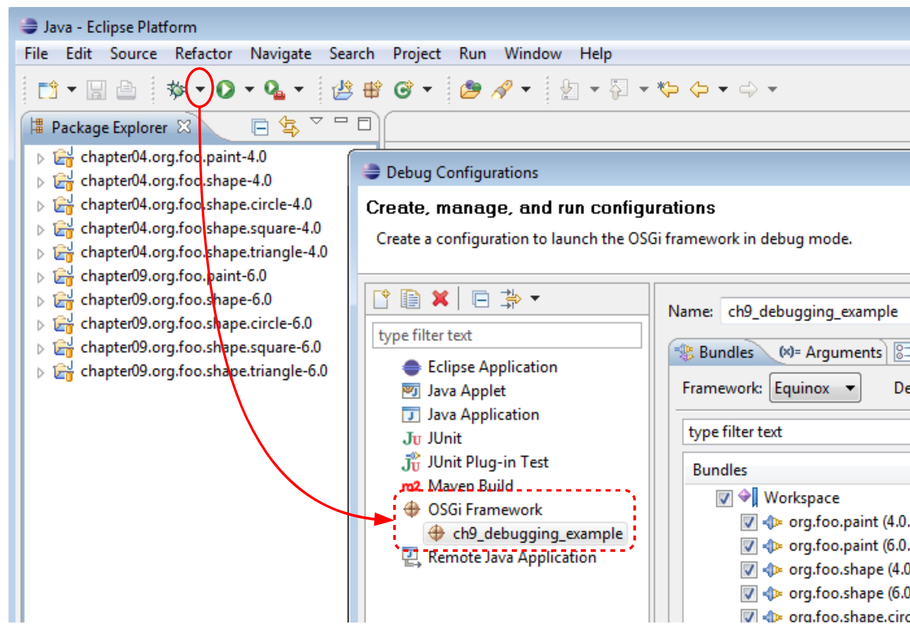


Figure 9.2 Configuring the Eclipse Debugger

This should open a dialog box similar to the one shown above. Now follow this list of instructions to configure a minimal Eclipse target platform for debugging the paint example:

6. Double-click on "OSGi Framework"
7. Change the name from "New_configuration" to "ch9_debugging_example"
8. Deselect "Include optional dependencies" as well as "add new workspace bundles"
9. Select "Validate bundles automatically"
10. Deselect the top-level "Target Platform"
11. Click on "Add Required Bundles"
12. Click on "Apply"

When you're happy with your selection press the "Debug" button to launch the debugger!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

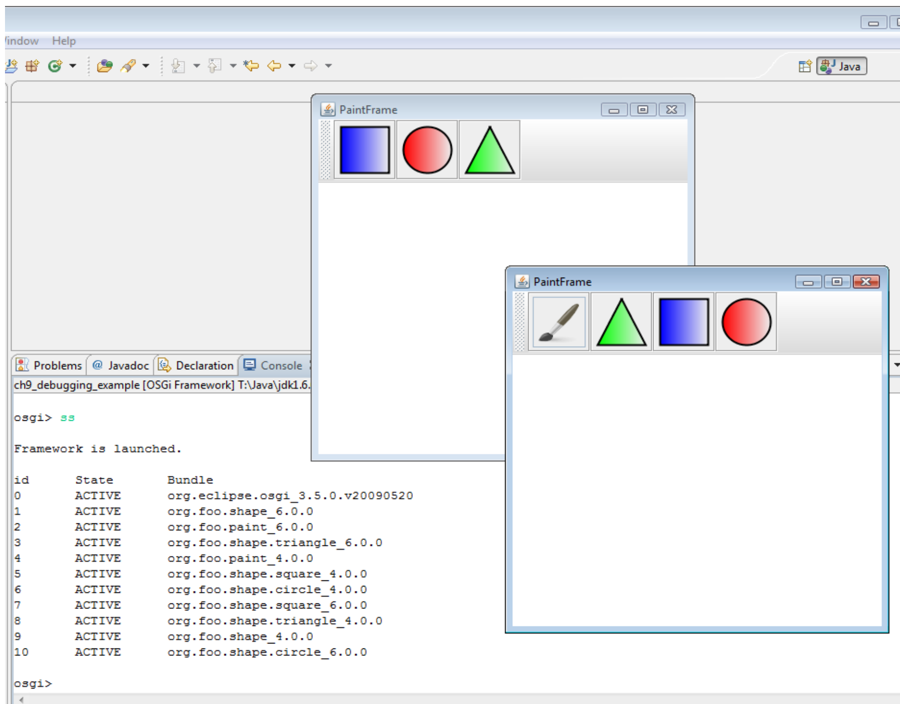


Figure 9.3 Debugging the paint example in Eclipse

Two different paint frames will appear, as depicted in Figure 9.3. This is because we have two versions of the code running simultaneously in the same JVM. Before we start to paint let's add a breakpoint so the debugger will stop when someone tries to use a null object

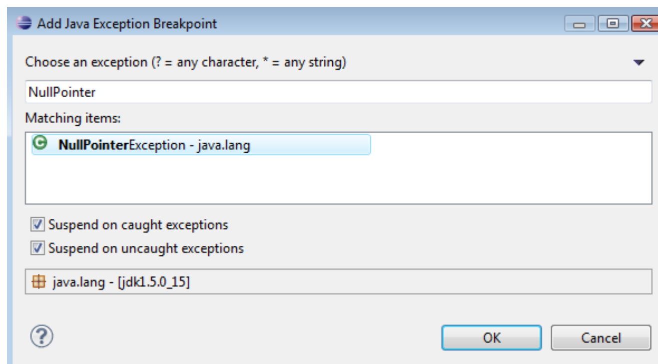


Figure 9.4 Watching for NullPointerExceptions

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

reference. Click on the “Run” menu and choose “Add Java Exception Breakpoint…”. This will open the dialog box shown in Figure 9.4. Select “java.lang.NullPointerException” and hit OK.

You should now have the two paint examples running in the Eclipse debugger. If you try to paint with the original version, which has just three shapes in its toolbar, everything will work as expected. But if you try to paint with the new version, the one with the paint brush in its toolbar, the debugger will stop (Figure 9.5).

Look closely at the title bar. It has correctly identified the affected source code is from `chapter09` even though there are multiple versions of this class loaded in the Java runtime. Once again, we see that the problem is caused by a null shape object. Using the Eclipse IDE, we can trace the exception back to the specific bundle project. We can also click on different frames in the stack trace to see what other bundles (if any) were involved. Compare this to

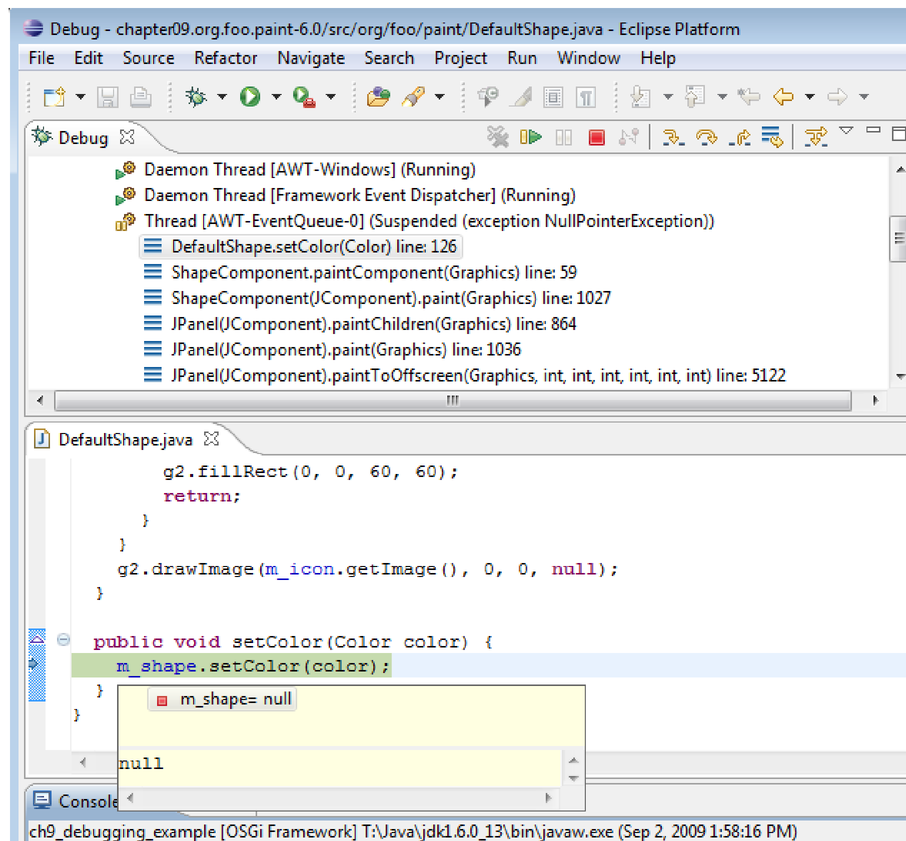


Figure 9.5 Exception caused by bad setColor() method

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

jdb, where it was difficult to tell which bundles were involved in a given stack trace without a good understanding of the source distribution.

We've successfully debugged an OSGi application with existing tools, from the basic jdb debugger to a fully-fledged IDE like Eclipse. But what do you do when you finally track down the bug? Do you stop your application, fix the code, and restart? What if your application takes a long time to initialize or if it takes hours to get it into the state that triggered the bug in the first place – surely there must be a better way!

9.1.2 Making things right with HotSwap

Thankfully there is and you might know it as "HotSwap". HotSwap is a feature of the Java 5 debugging architecture that lets you change the definition of a class at execution time without having to restart the JVM. The technical details behind HotSwap are outside of the scope of this book, what's more interesting to us is whether it works correctly with OSGi.

In order to use HotSwap you need to attach a native agent at startup to the low-level debugging hooks provided by the JVM. One such agent is attached whenever you run an application under jdb. While jdb provides a basic "redefine" command to swap in newly compiled classes, it won't work for our last example. Jdb refuses to redefine classes that have multiple versions loaded, because it can't determine which version should be redefined. But what about Eclipse? Can it help us update the right version of `DefaultShape`?

HotSwap with Eclipse

Back in section 9.1.1 we successfully used the Eclipse debugger to manage multiple versions of source code while debugging. Will Eclipse come to the rescue again and let us fix the broken `DefaultShape` implementation while leaving earlier working versions intact? If you still have the Eclipse debugger instance running you can skip onto the next paragraph. Otherwise, you'll need to re-launch the example by clicking on the drop-down arrow next to the bug icon (circled in Figure 9.2) and selecting "ch9_debugging_example". Trigger the exception once again by attempting to paint a shape.

You should have the paint example suspended in the debugger at the point of failure, as we saw back in Figure 9.5. Unlike jdb, which has to be told which classes to redefine, the Eclipse debugger automatically attempts to redefine any class whose source changes in the IDE (provided you have automatic builds enabled). This means all we need to do to squish this bug is change the `setColor()` method in the open `DefaultShape.java` window so that `m_shape` is initialized before we use it and save the file. For a quick solution we could simply copy and paste the relevant code from the `draw()` method, like in Listing 9.3:

Listing 9.3 Fixing the setColor() method

```
public void setColor(Color color) {
  if (m_context != null) {                                          #A
    try {
      if (m_shape == null) {                                        #B
        m_shape = (SimpleShape) m_context.getService(m_ref);        #C
```

```
            }
            m_shape.setColor(color);
        } catch (Exception ex) {}                                          #D
    }
}
```

#A confirm we're active
#B only get service once
#C access shape service
#D ignore missing shape

Copying code in this way is fine for a quick debugging session, but it would be better to extract the initialization code into a common method for use by both `draw()` and `setColor()` methods. Reducing code duplication makes testing and debugging a whole lot easier. For now we'll keep things simple: go ahead and paste the code from Listing 9.3 over the broken `setColor()` implementation. When you're ready, hit save to squish the bug!

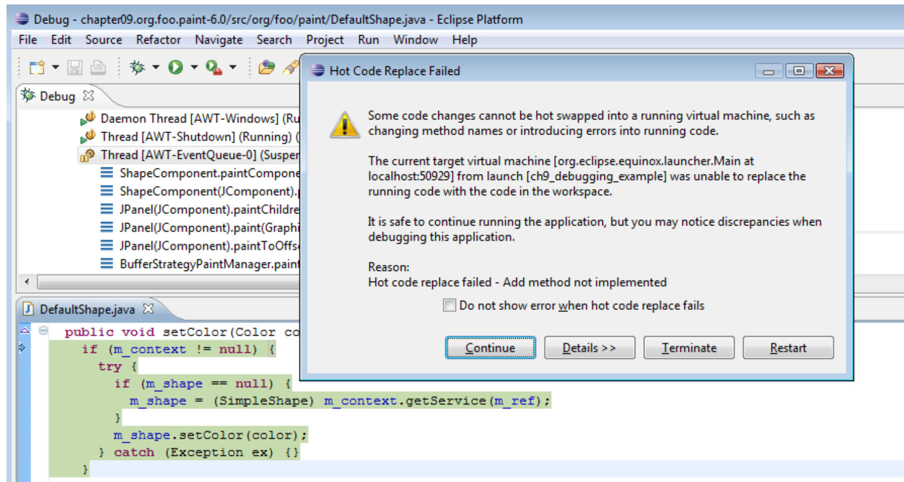


Figure 9.6 HotSwap failure updating DefaultShape

What just happened? Most, if not all of you, got an error message like the one in Figure 9.6 saying the JVM could not add a method to an existing class. This happened because Eclipse tried to update *both* versions of the `DefaultShape` class. While it was able to redefine the broken `setColor()` method in the version from this chapter, there is no such method in the `DefaultShape` class from chapter 4. Instead the debugger attempted to add the `setColor()` method to the old class, but adding methods is not supported by the current Sun implementation of HotSwap. Even worse, if we decide to ignore this error message and continue, we still get the same exception as before when painting shapes.

There are alternative implementations of HotSwap that do support adding methods. One such implementation can be found in the IBM JDK [ref]. If we debug the same example using

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

IBM Java 6 as the target runtime (remembering of course to first revert the `setColor()` method back to the broken version) we can successfully fix the problem without ever restarting the process. Figure 9.7 confirms that even after using HotSwap to squish the bug, both the old and new paint examples continue to work on the IBM JDK.

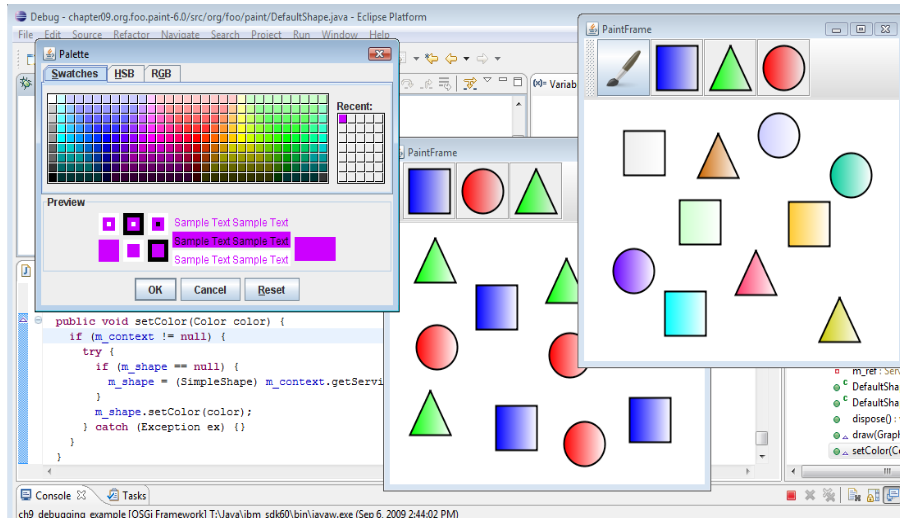


Figure 9.7 Successful HotSwap with the IBM JVM

While we eventually managed to use HotSwap to fix the problem in our bundle, this isn't exactly what we want because *all* versions of `DefaultShape` were updated. By chance this didn't affect our old paint example because we were adding a completely new method. It has no affect on the old application and just sits there unused. But what if we wanted to change a method that was tightly coupled to existing code? We could end up fixing one version only to find out we've broken all the others by unintentionally upgrading them with the new logic. This may not be a big deal during development, as you'll probably be focusing on one version at a time, but can we do better when debugging OSGi applications in the field?

HotSwap with JRebel

Yes we can do better; there is a JVM agent called JRebel (formerly known as JavaRebel [ref]) that behaves in a similar way to HotSwap, but has much better support for custom class loading solutions like OSGi. For those who don't know, a JVM agent is a small native library that attaches to the process on start-up and is granted low-level access to the Java runtime. Whenever you recompile a class, JRebel automatically updates the appropriate version loaded in the JVM without affecting any other versions of the class. This makes it very easy to develop, debug, and compare different releases of an application at the same time.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

So what are the downsides? The main downside is reduced performance due to the extra tracking involved. JRebel also needs to know how custom class loaders map their classes and resources to local files. It currently supports the Equinox OSGi implementation, but there's no guarantee it will work with other OSGi frameworks. Finally you need to add an option to the JVM command-line to use it, which is problematic in production environments that lock-down the JVM's configuration. In fact, some places won't let you use JVM agents at all because of the potential security issues involved. Agents have access to the entire process and can redefine almost any class in your application. Adding an agent to your JVM is like giving root access to a user in Linux. For these reasons, JRebel is usually best suited to development environments.

But what if you're working somewhere that forbids the use of debuggers or JVM agents, is there any other way we can update the broken bundle without restarting the whole process?

HotSwap the OSGi way

Update is the key word here. Remember back in section 3.7 where we discussed the update and refresh parts of the OSGi lifecycle, well we can use them here to deploy our fix without having to restart the JVM. To see this in action, we first need to revert the `setColor()` method of the local `DefaultShape` class back once again to the broken implementation:

```
public void setColor(Color color) {
  m_shape.setColor(color);
}
```

Next, completely rebuild our example:

```
$ ant clean dist
```

This time we're not going to use a debugger at all. We'll also add our command shell to the current set of bundles, so we can ask the framework to update the fixed bundle later.

```
$ ant add_shell_bundles

$ java -jar launcher.jar bundles
```

First, confirm you have the broken implementation installed by attempting to paint a shape (you should see an exception). Then fix the `setColor()` method of the `DefaultShape` class using the code from Listing 9.3 and rebuild the paint frame bundle in a new window:

```
$ cd chapter09/debugging-bundles/org.foo.paint

$ ant
```

We can now try updating our fixed bundle. Go back to the OSGi console and type:

```
-> update 6
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Where 6 is the ID of the paint frame bundle, as reported by the "bundles" command. When we issue the "update" command, the framework will update the bundle content by reloading the bundle JAR file from its original install location. It will also stop and restart the paint frame bundle, so you should see the paint frame window disappear and reappear. The paint example will now be using the fixed code, which means you can paint multicolored shapes as in Figure 9.8. Notice, we didn't need to follow the update with a refresh. This is because the paint frame bundle doesn't export any packages, so we know there are no other bundles hanging onto old revisions of the `DefaultShape` code.

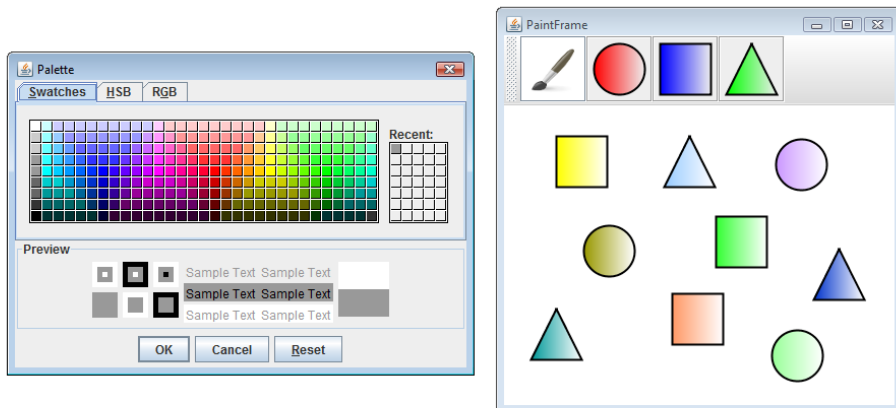


Figure 9.8 Painting with the fixed example

Unlike JRebel, the OSGi update process does not depend on a special JVM agent. It also doesn't have any significant effect on performance. These reasons together mean you could use the OSGi update process in a production environment. The downside is that we had to update and restart the entire bundle, potentially destroying the current state, rather than redefine a single class. If we wanted to keep any previously drawn shapes we would need to persist them somehow when stopping, and restore them when starting.

We've just seen how you can debug and fix problems in OSGi applications using everyday development tools such as jdb and Eclipse. We looked at more advanced techniques, such as HotSwap and JRebel, and finally used the tried and tested OSGi update process to fix a broken bundle. Hopefully these examples made you feel a bit more comfortable about debugging your own OSGi applications. In the next section, we will take a closer look at a set of problems you will eventually encounter when using OSGi: class loading issues.

9.2 Solving class loading issues

OSGi encourages and enforces modularity, which, by its very nature, can lead to class loading issues. Maybe you forgot to import a package or left something out when building a bundle. Perhaps you have a private copy of a class you're supposed to be sharing or forgot to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

make sure two tightly-coupled packages are provided by the same bundle. These are all situations which break modularity and can lead to various class loading exceptions. The right tools can help you avoid getting into these situations in the first place, but it is still worthwhile knowing what can happen and what the resulting problem signatures look like. In the following sections, we'll take you through a number of common class loading problems; what to look out for, what might be the cause, and how to solve them.

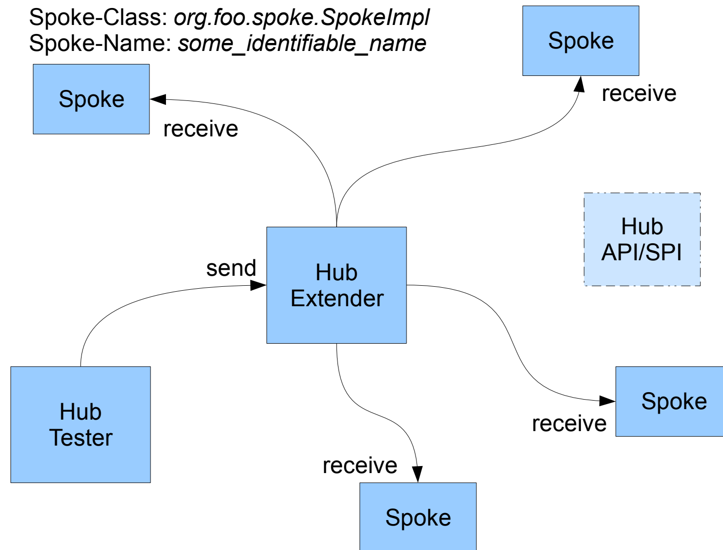


Figure 9.9 Simple hub-and-spoke message system

All the exceptions discussed in this section come from the same example application: a simple hub-and-spoke message system that uses the OSGi Extender Pattern (section 3.4) to load spoke implementations at execution time. The basic architecture is shown in Figure 9.9. The only thing that changes throughout this example is the content of the spoke implementation bundle; the API, hub extender, and test bundles remain exactly the same. By the end of this section you should understand how OSGi class loading can be affected by simple changes in content and metadata, and how you can diagnose and fix them when something goes wrong.

9.2.1 ClassNotFound vs NoClassDefFound

The first thing you should do when debugging a class loading exception is look and see if the exception is "ClassNotFound" or "NoClassDefFound". There is a subtle difference between these two types which will help you understand why the exception occurred and how to fix it.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

ClassNotFoundException

A `ClassNotFoundException` means the reporting class loader was not able to find or load the initial named class, either by itself or by delegating to other class loaders. There are three main reasons why this could occur in a Java application:

- There is a typo in the name passed to the class loader (very common),
- The class loader (and its peers) have no knowledge of the named class, or
- The named class is available, but is not visible to the calling code.

The third case, visibility, is where things get interesting. You know all about `public`, `protected`, and `private` access; but how many of you know what package-private means? Package-private classes are those without any access modifier before their `class` keyword. Their visibility rules are unique: in addition to only being visible to classes from the same package, they are also only visible to classes from the same *class loader*. Most Java programs have a single application class loader, so this last rule hardly ever comes up. OSGi applications contain multiple class loaders, but as long as each package is loaded by only one class loader it's effectively the same as before. The real problem arises with split packages (5.3.2), which span several class loaders. Package-private classes from a split package in one bundle are not visible to fellow classes in other bundles. This can lead to `ClassNotFoundExceptions` or `IllegalAccessExceptions` that wouldn't happen with a single application class loader. Figure 9.10 shows three different package-private scenarios; one classic and two involving split packages. Each scenario has subtly different class visibility.

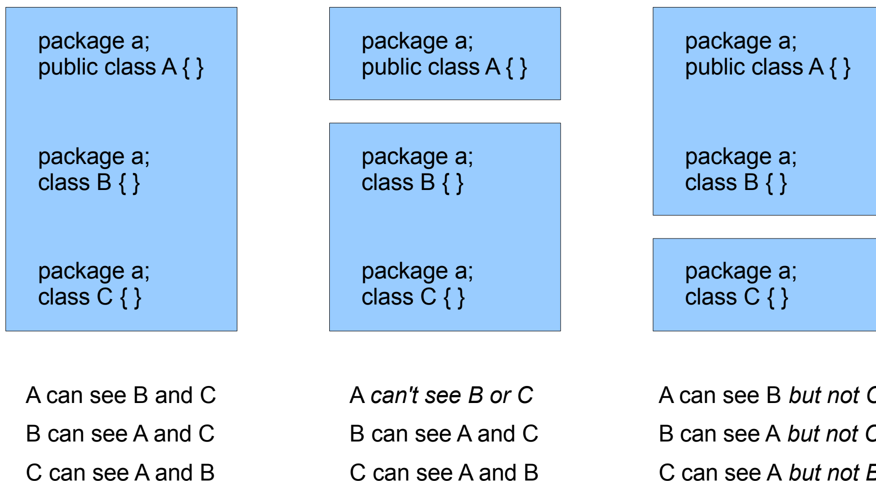


Figure 9.10 Split packages and package-private visibility

To see a common `ClassNotFoundException` situation, run the following example:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
./chapter09/classloading/PICK_EXAMPLE 1
```

This builds and deploys a spoke bundle that has incorrect extender metadata concerning its implementation class: it lists the name as `MySpokeImpl` instead of `SpokeImpl`. This is an easy mistake to make in applications configured with XML or property files because of the lack of type safety. The resulting exception gives the name of the missing class:

```
java.lang.ClassNotFoundException: org.foo.spoke.MySpokeImpl
```

You should use this information to check if the name is correct, the class is visible, and the package containing the class is either imported or contained inside the bundle. Most `ClassNotFoundExceptions` are easily solved by checking bundle manifests and configuration files. The hardest problems involve third-party custom class loaders; you inevitably need access to the class loader's source code to determine why it couldn't see a particular class, as well as having the patience to unravel the exception stack.

So that's `ClassNotFoundException`, but how is `NoClassDefFoundError` any different?

NoClassDefFoundError

Firstly, it's an error rather than an exception, which means applications are discouraged from catching it. Secondly, it means the initial class that started the current load cycle *was* found, but the class loader was not able to finish loading it because a class it depends on was missing. This can happen when a class is compiled against a dependent API, but the resulting bundle neither contains nor imports that package.

Continuing with our exceptional example, type:

```
./chapter09/classloading/PICK_EXAMPLE 2
```

This time the extender metadata in the spoke bundle is correct, but the bundle doesn't import the `org.foo.hub.spi` package containing the `Spoke` interface. The runtime begins to load the spoke implementation, but cannot find the named interface when defining the class:

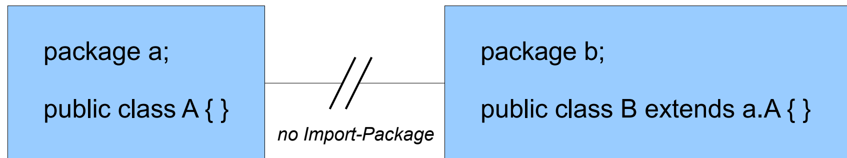
```
java.lang.NoClassDefFoundError: org/foo/hub/spi/Spoke
...
Caused by: java.lang.ClassNotFoundException: org.foo.hub.spi.Spoke
```

Debugging a `NoClassDefFoundError` involves tracing back through the dependencies of the class being loaded to find the missing link (or links). While the cause in this example is clear, developers often get side-tracked by assuming the initial class is at fault. The real culprit might be hidden right down at the bottom of the stack as the original cause of the exception. Once you know the real cause you can use the same problem solving approach used in `ClassNotFoundException` to fix the issue.

Figure 9.11 summarizes the difference between the two “missing class” exception types, together they make up many of the class loading issues you'll encounter when using OSGi.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Just remember: `ClassNotFoundException` means a class is missing, `NoClassDefFoundError` means one of its dependencies is missing.



bundleB.loadClass(“a.A”) → a.A✘ ClassNotFoundException

bundleB.loadClass(“b.B”) → b.B✔ → a.A✘ NoClassDefFoundError

Figure 9.11 Difference between ClassNotFound and NoClassDefFound

Unfortunately these two exceptions don't have a monopoly on confusing OSGi developers. A classic puzzle for people new to class loading goes something like this: you are given an object that says its type is `org.foo.Item`, but when you try to cast it to `org.foo.Item` you get a `ClassCastException`! What's going on?

9.2.2 Casting problems

How many of you would expect a `ClassCastException` from the following code?

```
ServiceTracker itemTracker =
    new ServiceTracker(bundleContext, "org.foo.Item", null);

itemTracker.open(true);                                              #1

Item item = (Item) itemTracker.getService();                         #2
```

At first glance it looks correct: we configure a service tracker to track services of type `org.foo.Item` and cast the discovered service, if any, to the same type. But notice how we open the tracker at (#1). Instead of calling the no-argument `open()` method as usual, we're passing in a boolean: `true`. This tells the service tracker to track *all* services whose type name matches the “org.foo.Item” string, not just the ones which are class loader compatible with our bundle (we discussed a similar situation back in section 4.5.1). If another bundle provides an `Item` service and happens to get the `org.foo` package from a different class space than us, you will see a `ClassCastException` at (#2).

How can this be? Recall from chapter 2 that class loaders actually form part of a type's identity at execution time, so the exact same class byte code loaded by two different class loaders is considered to be two distinct types. This makes all the difference, since OSGi uses

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

a class loader per bundle to support class space isolation in the same Java runtime. It also means you can get ClassCastExceptions when casting between types that look identical on paper.

To see this in practice, run the third example:

```
./chapter09/classloading/PICK_EXAMPLE 3
```

You should see a ClassCastException involving the Spoke class. This is because our spoke bundle contains its own private copy of org.foo.hub.spi, instead of importing it from the hub bundle. The spoke and hub end up using different class loaders for the same API class, which makes the spoke implementation incompatible with the hub:

```
java.lang.ClassCastException: org.foo.spoke.SpokeImpl
  cannot be cast to org.foo.hub.spi.Spoke
```

The fastest way to investigate these "impossible" ClassCastExceptions is to compare the class loaders for the expected and actual types. OSGi frameworks sometimes label their class loaders with the bundle identifier, so calling getClassLoader().toString() on both sides can tell you which bundles are involved. You can also use the framework console to find out who's exporting the affected package and who imports it from them. Use this to build a map of the different class spaces. The specific commands to use depend on the framework; at the time of writing this book the OSGi Alliance is still standardizing a command shell. On Felix, the "inspect package" command is the one to use. On Equinox you would use the "packages" or "bundle" commands. Once you understand the different class spaces, you can adjust the bundle metadata to make things consistent to avoid the ClassCastException. One approach might be to add "uses" constraints, which we first introduced at the end of chapter 2.

9.2.3 Using "uses" constraints

Cast your mind back to chapter 2, specifically the discussion about consistent class spaces in section 2.7.2. Bundles must have a consistent class space to avoid running into class-related problems, such as visibility or casting issues. When you have two tightly-coupled packages it is sometimes necessary to add "uses" constraints to make sure these packages come from the same class space. Perhaps you don't think you need all these "uses" constraints cluttering up your manifest, after all what's the worst that could happen if you remove them?

Let's find out by running the fourth example in our class loading series:

```
./chapter09/classloading/PICK_EXAMPLE 4
```

Yet again we get a class loading exception, except this time it happened inside the spoke implementation. The Java runtime noticed that we attempted to load two different versions of the Message class in the same class loader, in other words our class space is inconsistent!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
java.lang.LinkageError: loader constraint violation: loader (instance of
  org/apache/felix/framework/searchpolicy/ModuleImpl$ModuleClassLoader)
    previously initiated loading for a different type with name
      "org/foo/hub/Message"
```

How did this happen? Well our new spoke bundle has an open version range for the hub API, which means it can import any version after "1.0". It also provides a new "2.0" version of the `org.foo.hub` package that includes a modified `Message` interface. Now you might be wondering what this package is doing in our spoke bundle – maybe we're experimenting with a new design or perhaps it got included by mistake. How it got there is not really important. What *is* important is that we have a "2.0" version of `org.foo.hub` floating around without a corresponding "2.0" version of the `Spoke` SPI. Let's see how this affects the package wiring.

The hub extender and test bundles still have the original, restricted version range:

```
Import-Package: org.foo.hub;version="[1.0,2.0)",org.foo.hub.api;versio
 n="[1.0,2.0)",org.foo.hub.spi;version="[1.0,2.0)"
```

Thus, they get `Spoke` and `Message` from the original API bundle, but our spoke bundle has:

```
Import-Package: org.foo.hub;version="1.0",org.foo.hub.spi;version="1.0"
```

Which means it gets the original `Spoke` interface from the API bundle and the updated `Message` from itself. (Remember the framework always tries to pick the newest version it can.) This means the `Spoke` interface and our implementation see different versions of the `Message` interface, which causes the `LinkageError` in the JVM. Figure 9.12 shows the mismatched wiring.

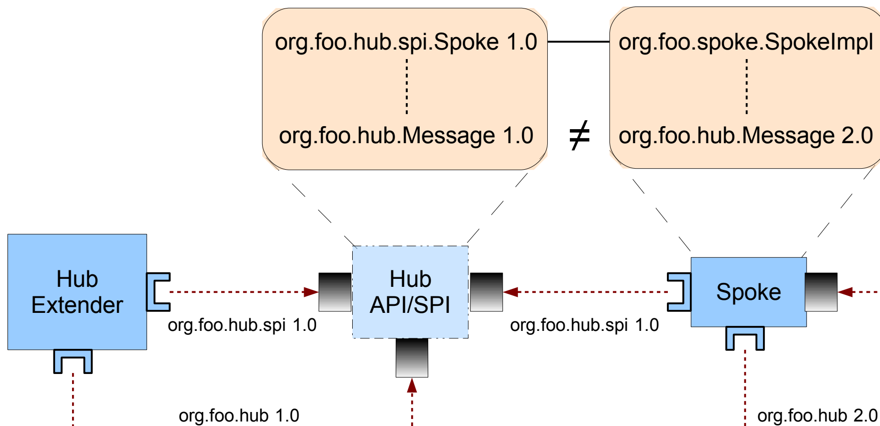


Figure 9.12 Mismatched wiring due to missing "uses" constraints

```
-nouses: ${no.uses}                                                  #A
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

#A ${no.uses} is true for example 4

Removing this re-enables bnd support for "uses" constraints. If we run the example again:

```
./chapter09/classloading/PICK_EXAMPLE 4
```

We no longer see any exceptions or linkage errors:

```
SPOKE org.foo.spoke.no_uses_constraints RECEIVED Testing Testing 1, 2, 3...
```

We just saw how "uses" constraints can help you avoid inconsistent class spaces and odd linkage errors, but what happens if they can't be satisfied? We can find out by tweaking the version range for `org.foo.hub` in the spoke bundle. By using a range of "[2.0, 3.0)" we leave only one matching exporter of `org.foo.hub`, the spoke bundle itself. But this breaks the "uses" constraints on the SPI package exported from the main API bundle, because it has a range of "`[1.0, 2.0)`" for `org.foo.hub`. These two ranges are incompatible, there is no way we can find a solution that satisfies both. The fifth example demonstrates the result:

```
./chapter09/classloading/PICK_EXAMPLE 5

Error starting framework: org.osgi.framework.BundleException:
  Unable to resolve due to constraint violation.
```

Unfortunately, the framework exception doesn't tell us which particular constraint failed or why. Determining why a solution wasn't found can be very time-consuming without help from the framework because the search space of potential solutions can be very large. Thankfully, Equinox has a "diag" command to explain which constraints were left unsatisfied. With Felix you can add more details to the original exception by enabling debug logging.

For example, if we change the last line in the PICK_EXAMPLE script to:

```
java "-Dfelix.log.level=4" -jar launcher.jar bundles
```

Then Felix prints the following message before the exception is thrown:

```
./chapter09/classloading/PICK_EXAMPLE 5

DEBUG: Constraint violation for 1.0 detected;
  module can see org.foo.hub from [1.0] and org.foo.hub from [2.0]
```

The message tells us the unsatisfied constraint is related to the `org.foo.hub` package, it also gives us the identifiers of the bundles involved. This is another reason why it's a good idea to use "uses" constraints. Without them you'd have to debug confusing class loading problems with no support from the framework. "Uses" constraints help avoid linkage errors to begin with and help the framework explain why certain sets of bundles aren't compatible.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

But it can only do this if the constraints are valid and consistent; which is why we recommend you always use a tool to compute them, such as bnd.

So far we've concentrated on what happens when your bundle metadata is wrong, but even a perfect manifest doesn't always guarantee success. Certain coding practices common to legacy code can cause problems in OSGi because they assume a flat, static class path. One practice worth avoiding is the use of `Class.forName()` to dynamically load code.

9.2.4 Staying clear of Class.forName

Suppose you're writing a module that needs to look up a class at execution time based on some incoming argument or configuration value. Skimming through the Java platform API you spot a method called `Class.forName()`. Give it a class name and it returns the loaded class – perfect, right? Its ongoing popularity suggests many Java programmers agree, but before sprinkling it throughout your code you should know it has a flaw: it does not work well in modular applications. It assumes the caller's class loader can see the named class, which we know is not always true when you enforce modularity.

How does this affect us as OSGi developers? Well, any class that you attempt to load using `Class.forName()` must either be contained, imported, or boot delegated by the bundle making the call. When you're loading from a selection of known classes this isn't such a big deal, but if you're providing a general utility (such as an aspect-weaving service), then there's no way to know which classes you might need to load. And even if you happen to know, you may decide to keep things flexible for the future. Those of you who remember our discussion on discovering imports from section 6.1.3 might think this sounds like a job for dynamic imports:

```
DynamicImport-Package: *
```

But dynamic imports only work when the wanted packages are exported. In addition, our bundle could get wired to many different packages in numerous client bundles. If any one of these bundles was refreshed our bundle would also end up refreshed, which in turn might affect the other bundles. Finally, we can only import one version of a package at any one time. If we want to work with non-exported classes or handle multiple versions of the same code concurrently, we need to find another way to access them.

Whenever you work with OSGi class loading, always remember there are well-defined rules governing visibility. Its not some arbitrary decision about who sees what. Every loaded class must be visible to at least one class loader. Our bundle might not be able to see the client class, but the client bundle certainly can. If we can somehow get ahold of the client class loader, we can use it to load the class instead of using our own class loader. This job is much easier if the method arguments already include a type or instance of a type that we know belongs to the client. In fact, let's see how easy it can be with the help of our sixth spoke implementation, shown in Listing 9.4:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Listing 9.4 Audited spoke implementation

```
public class SpokeImpl implements Spoke {

  String address;

  public SpokeImpl(String address) {
    this.address = address;
  }

  public boolean receive(Message message) {
    if (address.matches(message.getAddress())) {

      Class msgClazz = message.getClass();
      String auditorName = msgClazz.getPackage().getName() + ".Auditor"; #A

      try {
        Class auditClazz = Class.forName(auditorName);                    #B

        Method method = auditClazz.getDeclaredMethod(
            "audit", Spoke.class, Message.class);

        method.invoke(null, this, message);                               #C

        return true;

      } catch (Throwable e) {
        e.printStackTrace();
        return false;
      }
    }
    return false;
  }
}
```

#A assume same package
#B don't use forName!
#C call auditor method

This spoke assumes each `Message` implementation has an accompanying `Auditor` class in the same package and uses reflection to access it and log receipt of the message. The reason behind this design is not important, you might imagine the team wants to support both audited and non-audited messages without breaking the simple message API. What is important is that by using `Class.forName()` the spoke bundle assumes it can see the `Auditor` class. But we don't export our implementation packages, so when you run the sixth example you hopefully won't be too surprised to see an exception:

```
./chapter09/classloading/PICK_EXAMPLE 6

java.lang.ClassNotFoundException: org.foo.hub.test.Auditor
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

We know the `Auditor` sits alongside the `Message` implementation in the same package, so they share the same class loader (we don't have any split packages). We just need to access the `Message` implementation class loader and ask it to load the class like so:

```
Class auditClazz = msgClazz.getClassLoader().loadClass(auditorName);
```

Remove the `Class.forName()` line from the spoke implementation in Listing 9.4 and replace it with the line above. You can now run the sixth example without any problem:

```
./chapter09/classloading/PICK_EXAMPLE 6

Fri Sep 18 00:13:52 SGT 2009 - org.foo.spoke.SpokeImpl@186d4c1
  RECEIVED Testing Testing 1, 2, 3...
```

Class.forName() considered harmful!

Some of you may be wondering why we didn't use the longer form of `Class.forName()`, the method that accepts a user given class loader instead of using the caller's class loader. We don't use it because there is a subtle but important difference between these statements:

```
Class<?> a = initiatingClassLoader.loadClass(name);

Class<?> b = Class.forName(name, true, initiatingClassLoader);
```

First consider `loadClass()`: the initiating class loader is used to initiate the load request. It may delegate through several class loaders before finding one that has already loaded the class or can load it. The class loader that defines the class (by converting its bytecode into an actual class) is called the defining class loader. The result of the load request is cached in the defining class loader in case anyone else wants this class. Now consider `forName()`: while it behaves like `loadClass()` when looking for new classes, it caches the result in both the defining and initiating class loaders. It also consults the initiating loader cache before delegating any load request. So with `loadClass()` the resulting class could depend on your context, perhaps according to which module you're currently running in. But with `forName()` you get the same result irrespective of context. Because this extra caching might lead to unexpected results in dynamic environment such as OSGi, we strongly recommend you use `loadClass()` instead of `forName()`.

In our last example we found the client class loader by examining one of the arguments passed into our method and used that to look up the client's `Auditor` class. What if none of the method arguments relate to the client bundle? Perhaps we can use a feature specifically introduced for application frameworks in Java 2: the “Thread Context Class Loader”.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

9.2.5 Following the Context Class Loader

The Thread Context Class Loader (or TCCL) is as you might expect, a thread-specific class loader. Each thread can have its own TCCL, and by default a thread inherits the TCCL of its parent. Accessing the TCCL can be done with a single line of Java code:

```
ClassLoader tccl = Thread.currentThread().getContextClassLoader();
```

The TCCL is very useful when writing code that needs dynamic access to classes or resources, but must also run inside a number of different containers such as OSGi. Instead of adding a class loader parameter to each method call, you can instead use the code listed above to access the current TCCL. All the container needs to do is update the TCCL for each thread as it enters and leaves the container. When done properly this approach also supports nesting of containers, as shown in Figure 9.13.

{FIGURE}

Figure 9.13 Using TCCL with nested containers

Let's see how the TCCL can help us solve a class loading issue without affecting the API.

```
./chapter09/classloading/PICK_EXAMPLE 7
```

You should see an exception when the spoke attempts to load the Auditor class:

```
java.lang.ClassNotFoundException: org.foo.hub.test.Auditor
```

If you look at our seventh spoke implementation (Listing 9.5) you see it uses the TCCL:

Listing 9.5 Audited spoke implementation with TCCL

```
public class SpokeImpl implements Spoke {

  String address;

  public SpokeImpl(String address) {
    this.address = address;
  }

  public boolean receive(Message message) {
    if (address.matches(message.getAddress())) {

      Class msgClazz = message.getClass();
      String auditorName = msgClazz.getPackage().getName() + ".Auditor";

      try {
        Class auditClazz = Thread.currentThread()
            .getContextClassLoader().loadClass(auditorName);             #A
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
            Method method = auditClazz.getDeclaredMethod(
                "audit", Spoke.class, Message.class);

            method.invoke(null, this, message);

            return true;

          } catch (Throwable e) {
            e.printStackTrace();
            return false;
          }
        }
        return false;
      }
    }
```

#A use current TCCL

So as long as the TCCL is assigned properly by the container or the caller, this should work. The OSGi standard doesn't define what the default TCCL should be, it is left up to the framework implementers. This example uses Apache Felix, which leaves the default TCCL unchanged, in other words it will be set to the application class loader. Unfortunately the application class loader has no visibility of the Auditor class contained within the test bundle, which explains why we saw a `ClassNotFoundException`.

To avoid this exception we need to update the TCCL in the test bundle before sending the message. To be consistent we should also record the original TCCL and reset it after the call completes. This last step is very important if you want to nest or share containers inside the same process, as we saw in Figure 9.13. Now take a look at the test activator contained under "org.foo.hub.test", Listing 9.6 indicates the changes needed to set and reset the TCCL:

Listing 9.6 Setting and resetting the TCCL

```
public Object addingService(ServiceReference reference) {
  ClassLoader oldTCCL = Thread.currentThread().getContextClassLoader();   #A

  try {
    Thread.currentThread().setContextClassLoader(
        getClass().getClassLoader());                                     #B

    Hub hub = (Hub) ctx.getService(reference);
    hub.send(new TextMessage(".*", "Testing Testing 1, 2, 3..."));

  } catch (Throwable e) {
    e.printStackTrace();
  } finally {
    Thread.currentThread().setContextClassLoader(oldTCCL);                #C
  }
  return null;
}
```

#A record old TCCL
#B update TCCL

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

#C reset old TCCL

With these three changes we can re-run the test without any class loading problems:

```
./chapter09/classloading/PICK_EXAMPLE 7

Fri Sep 19 00:13:52 SGT 2009 - org.foo.spoke.SpokeImpl@186d4c1
  RECEIVED Testing Testing 1, 2, 3...
```

That wraps up our discussion of class loading problems. We were able to use the same example code to show a wide range of different exceptions that you might encounter when developing OSGi applications. Hopefully this will provide you with a foundation for any future class loading investigations. If you can relate a particular exception with one of the examples here then hopefully the associated solution will also help fix your problem.

Unfortunately class loading is not the only problem you might encounter when working with OSGi, but the next topic we will look at is indirectly related to class loading. OSGi enforces modularity with custom class loaders. An OSGi application will contain several class loaders, each one holding onto a set of resources. Unused class loaders should be cleared as bundles are uninstalled and the framework refreshed, but occasionally a rogue reference keeps a class loader and its associated resources alive. This can turn into a memory leak.

9.3 Tracking down memory leaks

Memory leaks can occur in OSGi applications just like any other Java application. All you need is something like a rogue thread or static field hanging onto one end of a spaghetti ball of references, stopping the garbage collector from reclaiming the objects. In a desktop Java application you might not notice any memory leaks because you don't leave the application running for long. As soon as you restart the JVM your old application with its ever-growing heap of objects is gone and you get a brand new empty heap to fill.

OSGi applications on the other hand typically have longer lifetimes, an uptime of many months is not unreasonable. In fact, one of the strengths of OSGi is that you are able to install, update, and uninstall bundles without having to restart the JVM. While this is great for maximizing uptime, it does mean you have to be very careful not to introduce memory leaks in your bundles. You cannot always rely on the process being occasionally restarted. Furthermore, updating a bundle introduces a new class loader to hold the updated classes. If there is anything holding onto objects or classes from the old class loader, then it won't be reclaimed and your process will use more and more class loaders each time the bundle is updated or re-installed.

Often times, class loader leaks can often be more problematic than simple object leaks, because several Java runtimes (like Sun's HotSpot JVM) place classes in a separate heap with a much smaller limit than the main object heap. Imagine how quick this so-called “PermGen” heap could fill up when each bundle update adds dozens more classes without unloading any class. While any leak is a cause for concern, depending on your requirements

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

not all leaks may warrant investigation. You might not even notice certain leaks if they only add a few bytes to the heap every now and again. So what is the best way to test for leaks in an OSGi application?

9.3.1 Analyzing OSGi heap dumps

[DOCUMENT memory-leaks EXAMPLE]

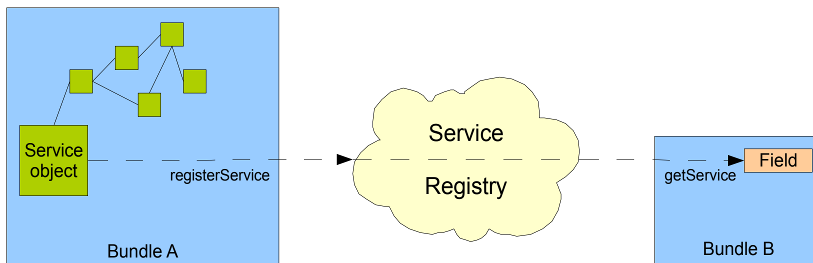
9.3.2 Monitoring bundle resources

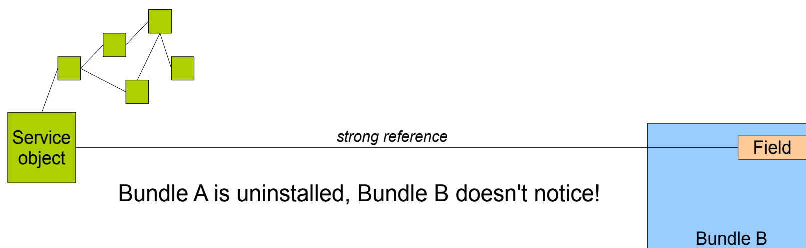
[DOCUMENT monitoring CONCEPTS]

9.4 Dangling services

In addition to the everyday leaks Java developers have to be careful of, the OSGi framework introduces a new form of memory leak to trap the unwary: dangling services. But what exactly do we mean by dangling?

Cast your mind back to section 4.3.1 where we showed why it was a bad idea to access a service instance once and store it in a field. This was because you wouldn't know when this service was unregistered by the providing bundle. Your bundle would continue to keep a strong reference to the original service instance and its entire graph of references long after the providing bundle had been updated or uninstalled. You would also be keeping alive the class loaders of any classes used by this instance. Like many memory leaks you could end up with a significant amount of potential space being kept alive by one single field. Clearing this field would free everything up and allow your application to continue its uptime.





©Ma um:

Figure 9.14 Classic dangling service

9.4.1 Finding a dangling service

In an ideal world your application won't resemble a haystack! Often you will have some idea of where the leak might be because of the bundles involved. For example, if bundle A leaks when it is updated and you know that it's only used by bundles X and Y, then you can concentrate your search on those three bundles. This is another benefit of modularity: by enforcing module boundaries and interacting indirectly via the service registry you reduce the contact points between modules. You no longer have to read through or instrument the entire code base for potential references because different concerns are kept separate from one another. But regardless of how much code you have to look through, there are a couple of techniques you can use to narrow the search, ranging from high-level queries to low-level monitoring.

Querying the framework

You can perform high-level monitoring by using facilities built into the OSGi framework to track service use. The `Bundle` API has a method called `getServicesInUse()` to tell you which services the OSGi framework believes a given bundle is actively using at any one time. Remember from Chapter 4, this is done by tracking calls to `getService()` and `ungetService()`. Unfortunately, many developers and even some service-based frameworks do not call `ungetService()` when they are done with a service, which can lead you to think there is a leak where there isn't one. This approach also doesn't detect when a direct reference to the service escapes from the bundle into some long-lived field. You can also use the `getUsingBundles()` method from the `ServiceReference` API to perform a reverse check and find out which bundles are using a given service, but this too doesn't account for incorrectly cached instances.

Monitoring with JVMTI

Low-level monitoring is possible using the JVM Tools Interface [ref] or JVMTI for short. JVMTI is a native API that provides several ways to interrogate, intercept, and introspect aspects of the JVM such as the Java heap, locks, and threads. There are already open-source agents that can analyze the heap to find leak candidates. It should be possible to take these generic agents and enhance them with knowledge about OSGi specifics, so they can watch for references to OSGi service instances on the Java heap and determine which bundle is responsible for holding onto them.

Just as we saw when debugging, it's one thing to find out why something is happening. Being able to do something about it (in this case protect against it) is even more important.

9.4.2 Protecting against dangling services

One of the simplest ways to protect against dangling services is to let a service component framework like Declarative Services manage services for you. Component frameworks are discussed in detail in Chapter 11, for now you might like to think of them as watchful parents that protect their children from the harsh realities of the world. But even component

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

frameworks may not be able to help against rogue clients that stubbornly refuse to relinquish references to your service. We somehow need to give these bundles a reference that we can clear ourselves, without requiring their co-operation.

One way to do this is by using a delegating service object. A delegating service object is basically a thin wrapper that implements the same set of interfaces as the original service. It contains a single reference to the real service implementation that can be set and cleared by methods only visible to our registering bundle. By registering this delegating object with the service registry instead of the real service implementation we stay in control. Because client bundles are unaware of the internal indirection, they cannot accidentally keep a reference to the underlying service. As Figure XX shows, we can decide to sever the link at any time:

Figure XX Delegating service object

While you could manually create delegating service objects upfront, this would only make sense for small numbers of services. For large systems you'd want a generic service that accepts a set of interfaces at execution time and returns the appropriate delegating service object. There's also an overhead involved in both memory and performance, so you should really only use a delegating service object when you really don't trust client bundles to do the right thing or your service uses so many resources that even a single leak could be dangerous.

9.5 Summary

We started off this chapter with a practical guide to debugging OSGi applications using the console debugger (jdb) and an advanced IDE (Eclipse). We then moved onto specific issues that you may encounter while working with OSGi, including *seven* class loading problems:

- ClassNotFoundException
- NoClassDefFoundException
- ClassCastException
- Missing 'uses' constraints
- Mismatched 'uses'
- Class.forName issues
- TCCL loading issues

This was followed by a couple of related resource discussions:

- Memory / resource leaks
- Dangling OSGi services

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The next chapter should be a welcome break from all of this low-level debugging and testing. Look out for fresh, high-level concepts as we discuss managing OSGi applications!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

10
Component Models

So far in this book we have shown you how to develop applications that run using the core OSGi framework, which includes three key layers: modularization, lifecycle and services. In the chapter 2 we mentioned that component orientated programming actually shares some common characteristics with modularization. Also in chapter 4 we mentioned that a service model can work along side a component model. So there's obviously some level of synergy between OSGi and component technologies. During this chapter we'll explore various types of component models and their usage in an OSGi environment. By the end of this chapter you will know how to migrate existing component applications to OSGi or build new OSGi applications with a minimum of fuss.

Component models have become incredibly popular in Java development over the past decade, and there are a vast number of different schemes that can be used, including Enterprise Java Beans, Spring Beans, Google Guice, SCA, Fractal, Avalon, etc. During the course of this chapter we are going to look at three different component models that are designed to work specifically in an OSGi environment; Declarative Services, Blueprint Container and iPojo. The first two are defined as part of the OSGi compendium specification, the third is an extremely interesting project hosted at the Apache Felix project. We will reuse our paint example converting it to use each of these component models and in the process of doing so we will discuss the various unique features of each model.

At the end of this chapter we will see it is possible for all three of these component models to interoperate in the same framework along side standard OSGi services. We feel this demonstrates the extreme flexibility provided by the OSGi specification. Given a common classloading and services model with enforced modularity boundaries different developers can pick and choose the programming model they wish to use. Whilst still

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

allowing their code to work seamlessly at runtime with code written by other developers using potentially very different programming models.

The key aspect of all component technologies is that they describe the functional building blocks of your application. These building blocks are typically business objects which publish particular interfaces and consume other interfaces provided by other components. When using a component model you code your business objects using a certain pattern defined by the component model. Table 10.1 provides a quick summary of the types of component models and the patterns they employ.

Table 10.1 Types of component models

Type	Description
Type I	Early component models such as EJB 2.0 required the developer to implement defined interfaces, or implement defined method signatures e.g. initialize(), setDependency(Service service) or destroy() etc.
Type II	The next generation of component models abstracted the component lifecycle to an external configuration file – usually XML – which defined the lifecycle methods to call on the Java objects.
Type III	Most recently component models tend to use a sprinkling of annotations to mark out the lifecycle methods to call

Component benefits

The requirement to code to a certain pattern is actually a bit of a constraint, as you are bounded by the rules of the component framework, so why do we do it? Well the key point is that by conforming to a pattern and providing a description of the components you wish to construct a third party framework can be tasked with the actual work of constructing and managing the business objects that make up your application. This provides a number of benefits:

- Redundancy - removal of boiler plate code
- Composability – common patterns allow components to be plugged together
- Uniformity – making code easier to navigate for new developers

Component downsides

This sounds very useful, but are there any downsides to using a component framework? Well yes there are always issues that have to be weighed up in any architectural decision. Table 10.2 details the issues you should consider when choosing whether to employ a component model, note these issues apply whether or not you are using OSGi but it is worth reminding ourselves.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Table 10.2 Problems associated with usage of component models

Problem	Description	Analysis
Bloat	Some component frameworks are relatively heavy so for small applications they may not be appropriate	How complex are your application dependencies? Is the extra functionality provided by a component framework required?
Diagnosis	Debugging service dependency problems requires a new set of tools to figure out what is going on when your services are not published as expected	We'll show you in this chapter some of the tools available, it is up to you which you think is most appropriate
Side file syndrome	Build or runtime problems causes by component configuration becoming stale with respect to Java source code can be frustrating to debug	IDE tooling can definitely help here by providing refactoring support and early analysis. Again we'll try to point out relevant tooling during this chapter where appropriate

But overall we feel that component models are a significant benefit to OSGi development as they simplify a lot of the low level logic for dealing with services that you would otherwise have to manage yourself.

10.1 Component models and OSGi

In this section we will look at the specific benefits of using component models in an OSGi environment and look at the general architecture of a component frameworks within OSGi. This will provide us with context for the rest of the chapter such that we can focus on the unique features of each component implementation.

10.1.1 Why components?

The most immediate benefit of moving to a component model in OSGi is the simplification it brings to handling multiple service dependencies i.e. the removal of redundant boiler plate code. Recall in chapter [ref] we showed you various mechanisms for looking up services from the OSGi BundleContext, either direct lookup via the getServiceReference method of BundleContext or via indirect notification via the ServiceListener interface or the ServiceTracker utility class. These mechanisms provide you with the flexibility to allow you to tailor how your application responds to the availability of service dependencies to a high degree of detail. However in many cases you will find yourself repeating the same basic set of service dependency code.

Consider a trivial example where a class `FooImpl` depends on service `Bar` and should only publish it's service when `Bar` is available, or to say it another way `FooImpl` has a 1..1 cardinality dependency on a `Bar` service. This scenario is shown in figure 10.1.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

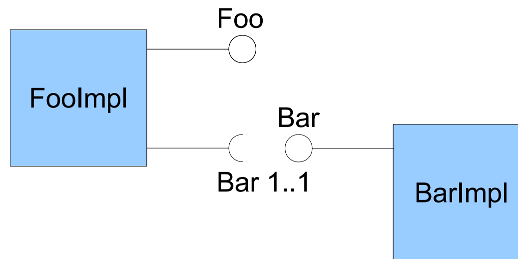


Figure 10.1 Trivial service dependency between two components Foo and Bar

Listing 10.1 shows the code that achieves this scenario using a ServiceTracker pattern.

Listing 10.1 Service tracker providing 1..1 dependency on a service

```
class BarTracker extends ServiceTracker {
  private final FooImpl foo;
  private final BundleContext ctx;
  private LinkedList<Bar> found = new LinkedList<Bar>();
  private ServiceRegistration reg;

  BarTracker(FooImpl foo, BundleContext ctx) {
    super(ctx, Bar.class.getName(), null);
    this.foo = foo;
    this.ctx = ctx;
  }

  @Override
  public Object addingService(ServiceReference reference) {
    Bar bar = (Bar) super.addingService(reference);
    found.add(bar);                                                    #1
    if (foo.getBar() == null) {                                        #2
      foo.setBar(bar);                                                 #3
      reg = ctx.registerService(Foo.class.getName(), foo, null);       #4
    }
    return bar;
  }

  @Override
  public void removedService(ServiceReference reference, Object service) {
    found.remove(service);
    if (foo.getBar() == service) {                                     #5
      if (found.isEmpty()) {
        reg.unregister();                                              #6
        foo.setBar(null);                                              #7
        reg = null;
      }
      else {
        foo.setBar(found.getFirst());                                  #8
      }
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
      }
      super.removedService(reference, service);
    }
  }
```

This service tracker checks if this is the first Bar service it has found at #2. If so it calls the FooImpl.setBar() method at #3 prior to registering the service in the BundleContext at #4. If more than one Bar service is found then backups are stored at #1. If the original Bar service is removed (#5) then one of the backups is set in it's place at #8. If no backup is available the service is unregistered at #6 and the setBar method is called with a null parameter at #7.

Now you may be looking at this code and thinking "Gee that looks pretty complicated" which is especially true if you also consider that this only covers the 1..1 cardinality case – we need a different set of logic to deal with 0..1, 0..n and 1..n cardinalities. Then consider how complex things get if you have a service that has multiple dependencies – i.e if FooImpl has a 1..1 dependency on Bar and a 1..n depencency on a Baz service. The logic at #4 and #6 to publish or unpublish the Foo service to the BundleContext now needs to take account of whether all dependencies are satisfied.

You could obviously implement your own abstracted wrapper classes around these use cases but in actual fact this is exactly what the various OSGi component models do on your behalf. So unless you are very keen to implement your own version of a component model you are more than likely better off reusing one of these models.

Let's convince you of the benefits of using a component model by showing you the code you need to write to achieve the uber complex Foo, Bar, Baz dependency case we alluded to above using one of the component models we will cover in this chapter – declarative services. The Java code needed to implement this is shown in listing 10.2.

Listing 10.2 Multiple dependencies with declarative services

```
public class FooImpl implements Foo {
  private volatile Bar bar;                                            #1
  private List<Baz> bazList = Collections.synchronizedList(new
LinkedList<Baz>());                                                    #2

  protected void setBar(Bar bar) {                                     #3
    this.bar = bar;
  }

  protected void addBaz(Baz baz) {                                     #4
    bazList.add(baz);
  }

  protected void removeBaz(Baz baz){                                   #5
    bazList.remove(baz);
  }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

This class declares two internal references to the Bar and Baz services at #1 and #2 respectively and provides accessor methods to set these references at #3, #4 and #5. “So where is the OSGi code in this example?” I hear you ask. Well there isn't any! The only non trivial bits of code in this example are the use of the volatile keyword and the synchronized wrapper class which we use to protect ourselves from the dynamic nature of services in an OSGi environment. So how *do* we get the services to and from the BundleContext? The answer is via an external xml configuration file which conforms to the Declarative Services specification as shown in listing 10.3.

Listing 10.3 Multiple dependencies with declarative services

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
  <scr:implementation class="org.foo.example.FooImpl" />           #1

  <scr:reference                                                   #2
    name="bar"                                                     #2
    interface="org.foo.example.Bar"                                #2
    cardinality="1..1"                                             #2
    policy="dynamic"                                               #2
    bind="setBar"                                                  #2
    unbind="setBar"/>                                              #2

  <scr:reference                                                   #3
    name="baz"                                                     #3
    interface="org.foo.example.Baz"                                #3
    cardinality="1..n"                                             #3
    policy="dynamic"                                               #3
    bind="addBaz"                                                  #3
    unbind="removeBaz"/>                                           #3

  <scr:service>
     <scr:provide interface="org.foo.example.Foo"/>                #4
  </scr:service>

</scr:component>
```

This configuration file tells the declarative services framework to:

- Create an instance of our FooImpl class at #1
- Search for one instance of the Bar service and call the setBar method when one is found at #2
- Search for one or more instances of the Baz service and call the addBaz method when one arrives and the removeBaz method when one is removed at #3
- Publishes the FooImpl to the BundleContext under the Foo interface when one Bar and at least one Baz service are connected to the FooImpl object

Hopefully you will appreciate that this significantly reduces the amount of boilerplate code you will need to use in your applications. Though the xmlaphobes among you are probably

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

starting to get one of *those* headaches. In general we agree that though there are question marks over the *POJOness* of components written with annotations they are in general more expressive and less prone to code rot. But in fact the choice of meta model is even less critical in the long term, OSGi places no restriction on the format of the meta model used to share services. In our last example in this chapter we will show how it is possible to mix component technologies some that use XML and another that uses annotations in a single application. This allows for infinite flexibility in the long term – you can pick and choose the model that suits you best.

Hang on we've introduced a new term here "meta model" without specifying what we mean by this (collective slap on the wrist). Let's look now at the general architecture of a component framework within an OSGi context.

10.1.2 Component generalizations

Almost all component frameworks in OSGi follow the same basic pattern:

```
     (Component Extender)
   + (Java Code + Meta Model + Component Indicator)
  => Runtime
```

In the above discussion on the benefits of component models in OSGi we have seen an example of the *Java Code* and *Meta Model* in the form of the FooImpl class and the foo.xml component definition respectively. Obviously the use of XML as the meta model is purely an arbitrary (though common) choice. Another popular route is to combine the java code and the meta model through the use of annotations. But what about the other parts of this formula?

Well in general component frameworks within OSGi follow the extender pattern which we covered in chapter [ref] when adding lifecycle hooks into the paint program. As such the *Component Extender* is a bundle or set of bundles that work together to publish and consume services on behalf of the the client component bundles. These extender bundles typically look for some form of *Component Indicator* within the bundles – either a manifest header or some well known file patterns to decide whether they should consider this bundle for extension.

Given all of these ingredients the component extender could be said to act like a catalyst in a chemical reaction – as depicted in figure 10.2. Without it the various component bundles will sit inert within the OSGi framework. Though they provide the description of what to do, they do not actively manage their own lifecycles. But when a component provider is installed and started all sorts of services and processes will spring into life within the OSGi framework.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

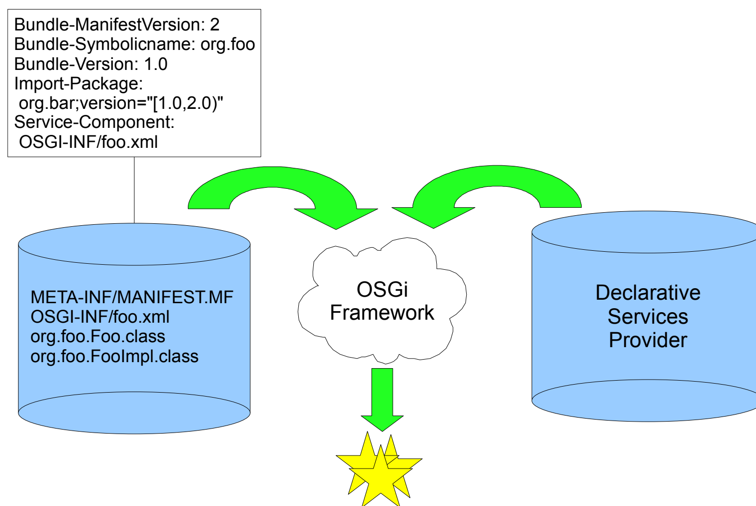


Figure 10.2 Mixing bundles together can have some spectacular results!

10.1.3 Painting with components

During the rest of this chapter we are going to look three different component models, Declarative Services, Blueprint Container and iPojo. With each of these component frameworks we will convert them from the procedural Java code of BundleActivators, ServiceRegistrations and ServiceListeners to the applicable meta model for that component framework. In each case our components will follow a common pattern as shown in figure 10.3.

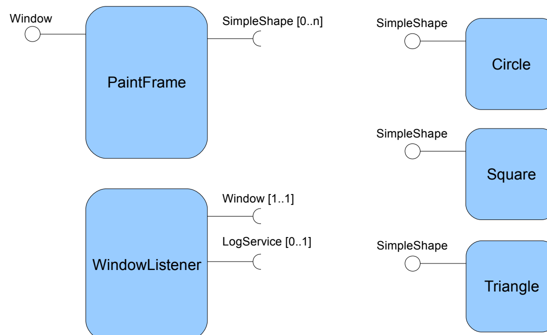


Figure 10.3 Components used in modified paint application.

Each of these components is packaged in it's own bundle (though this is not strictly necessary). The PaintFrame exposes a Window service and consumes 0..n SimpleShape services and the shape components each export a single SimpleShape interface

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The WindowListener has a 1..1 dependency on the Window service and shuts down the framework when the window it is bound to closes, it also has an optional 0..1 dependency on the LogService to log a message when the window is closed. In our procedural paint application the function of the window listener was realized as an inner class registered in the bundle activator. This class listened for the window closing event and shutdown the OSGi framework by stopping bundle zero (as we discussed in chapter [ref]). In our componentized paint program the WindowListener binds to published java.awt.Window services and registers itself as a listener for the window closing event. In a modular java environment we might imagine that many window objects could be published from a single JVM as such we use an attribute and filter pattern to limit the set of windows that our WindowListener component binds to.

The code for this chapter is found in the following directories:

- Declarative Services – $oia/code/chapter10/paint-example-ds
- Blueprint – $oia/code/chapter10/paint-example-bp
- iPojo – $oia/code/chapter10/paint-example-ip

If you compile and run each of these examples you will see that they each behave exactly as the original procedural OSGi code from chapter [ref]. But what's going on underneath the hood? Let's turn our attention to the first component model on our list, declarative services.

10.2 Declarative Services

Declarative Services is the first component model defined by the OSGi alliance, it was added to the OSGi specification in release 4.1 and it defines a lightweight approach to managing service dependencies. The basis of the declarative services specification is to address three main areas of concern, which we discuss below in table 10.3.

Table 10.3 The declarative services raison d'etre.

Area of concern	**Discussion**
Startup Time	Whilst the lifecycle layer of OSGi provides a mechanism to dynamically register and consume services, with many bundles each having activators the initialization time of each bundle adds to the initialization time of the entire application.
Memory Footprint	A service model allows for consumers to be decoupled from their implementations however the downside is that a service provider has no easy way to find out in advance whether it's service is actually required. Registering services implies the creation of many classes and objects to support the service and each of these requires memory to be consumed – if these services are never

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

	used then this is wasted resource that could have been used for other application functions.
Complexity	We've already discussed the amount of boiler plate code that is required to handle complex service dependency scenarios, any amount of boilerplate code represents a risk in terms of software maintenance - more moving parts imply more potential bugs or more work during upgrades.

10.2.1 Building declarative services components

Let's start by considering how to convert the circle bundle to use declarative services (the square and triangle follow the exact same pattern). If you inspect the contents of the circle bundle you will see it now has the following contents:

```
META-INF/MANIFEST.MF
OSGI-INF/
OSGI-INF/circle.xml
org/
org/foo/
org/foo/shape/
org/foo/shape/circle/
org/foo/shape/circle/Circle.class
org/foo/shape/circle/circle.png
```

The first thing you will notice here is that there is no BundleActivator class contained in this bundle, however if we inspect the services published by the circle bundle you will see that it is exporting a service with the SimpleShape API:

Listing 10.4 Confirming the SimpleShape API is published as a service

```
-> ps                                                              #1
START LEVEL 1
   ID   State         Level  Name
[   0] [Active     ] [    0] System Bundle (2.0.1)
[   1] [Active     ] [    1] circle (4.0)
...
-> services 1                                                      #2

circle (1) provides:
--------------------
component.id = 0
component.name = circle
objectClass = org.foo.shape.SimpleShape
service.id = 9
simple.shape.icon = circle.png
simple.shape.name = Circle
```

At #1 we list the bundles installed within the framework using the felix shell command "ps" then having found the bundle id of our circle component we list it's services at #2. So

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

how did this get here if there is no BundleActivator? The clue we need is located in the MANIFEST.MF file for this bundle, notice that it contains a new manifest entry:

```
Service-Component: OSGI-INF/circle.xml
```

This manifest entry is defined by the declarative services specification and is used to advertise the location of an xml component definition file within the bundle. When a bundle is installed within the OSGi framework that contains this manifest entry the declarative services provider bundle will be triggered to act as extension provider and add ServiceListeners and register services on that bundles behalf based on the policy set out in this component definition file.

In declarative services the convention is to put these component definition files in the directory OSGI-INF/<component-name>.xml but in fact they can go anywhere within the bundle. It is possible to include multiple component files in a single bundle either by comma delimiting the set of configuration files that should be registered or by placing all files in a single directory and using a wild card pattern, or indeed a mixture of the two:

```
Service-Component: OSGI-INF/foo.xml,OSGi-INF/baz.xml
```

Or

```
Service-Component: OSGI-INF/*.xml
```

Segue

FRAGMENTED COMPONENTS

It is also possible to place component definition files in bundle fragments (which we covered in chapter [ref]). In this case only the host bundle's Service-Component manifest header is read, though the xml configuration may reside in a separate bundle fragment. A possible use case for doing this would be if you wanted to support several different component configuration options and choose at deployment time which is actually instantiated.

Publishing services with declarative services

The following code snippet shows the declaration used to publish our Circle as a service to the OSGi bundle context under the SimpleShape API (with attributes specifying the name and icon that the Paint component should use to display this component):

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0">
  <property name="simple.shape.name" value="Circle" />                 #1
  <property name="simple.shape.icon" value="circle.png" />             #2
  <scr:implementation class="org.foo.shape.circle.Circle" />           #3
  <scr:service>
     <scr:provide interface="org.foo.shape.SimpleShape"/>              #4
  </scr:service>
</scr:component>
```

In this file we define two properties for our circle component at #1 and #2. Properties may be used to configure a component (which we will look at shortly) but importantly they are also automatically added as attributes to services published from components. We define

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

the implementation class for this component at #3 which must be a class that is visible to the bundle classpath where this component file is located. At #4 we define that this component exports a SimpleShape service interface, let's now look at our Circle class shown below in listing 10.5

Listing 10.5 Circle class used in the declarative services paint example

```
package org.foo.shape.circle;

import java.awt.*;
import java.awt.geom.Ellipse2D;
import org.foo.shape.SimpleShape;

public class Circle implements SimpleShape {
  public void draw(Graphics2D g2, Point p) {
    ...
  }
}
```

Here you will note that the circle class is exactly the same as that defined in the procedural OSGi services paint program. Segue

Component Properties

In the case of our circle component we specified the service attributes using the declarative services construct of component properties. This follows the same pattern of property propagation that we saw in chapter [ref] where by any configuration property not prefixed with a "." is copied onto the service. In the our case we statically declared the properties within our component xml file using <property> element. But in fact this is the last place that declarative services looks for component properties. In fact component properties may be sourced from:

13. Properties passed to the ComponentFactory.newInstance method
14. Properties retrieved from the ConfigurationAdmin service using component name as the PID.
15. Properties defined in the component description xml file.

Properties defined in 1, override properties of the same name in 2 and 3 and likewise properties defined in 2 override properties defined in 3. This precedence behavior allows a developer to setup the component with default component properties but allows another user or system administrator to change the configuration at runtime to suite his or her needs.

DECLARATIVE SERVICES HAVE SIMPLE ATTRIBUTE TYPES

One important difference between this approach compared to the procedural version used in chapter [ref] (apart from the obvious lack of java activation code) is that the declarative services specification is only able to publish services with "simple" java types (String (default), Long, Double, Float, Integer, Byte, Character, Boolean, Short). In our

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

procedural version of this paint example we added the image that represented the shape as an attribute of the service using the following code:

```
dict.put(SimpleShape.ICON_PROPERTY, new
ImageIcon(this.getClass().getResource("circle.png")));
```

This is does mark a breaking change between a declarative services model and a "pure" OSGi service model so is something you should be aware of when choosing a component model. However in the vast majority of situations this limitation is actually not that big a deal, though we will return to this topic later in the chapter.

Consuming services with declarative services

So that seems fairly straight forward, but then our circle component does not consume any services, it only publishes them. To see how we use external services let's now turn our attention to the Paint frame. Again in this case the paint bundle contains a similar `OSGI-INF/paint.xml` component definition shown in listing 10.6:

Listing 10.6 Declarative services definition of Paint frame component

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
  name="paint"                                                        #1
  immediate="true">                                                   #2

  <scr:implementation class="org.foo.paint.PaintFrame" />             #3

  <scr:reference                                                      #4
    interface="org.foo.shape.SimpleShape"                             #5
    cardinality="0..n"                                                #6
    policy="dynamic"                                                  #7
    bind="addShape"                                                   #8
    unbind="removeShape"/>                                            #9

  <scr:service>
     <scr:provide interface="java.awt.Window"/>                      #10
  </scr:service>

  <scr:property name="name" value="main"/>                           #11
</scr:component>
```

There's quite a lot going on in this code listing but as with our shape components this paint component declares a new component called "paint" at #1 and publishes a service at #10 with the service attribute defined at #11 backed by an implementation class defined at #3.

In fact the first new element we want to draw your attention to is at #4. Here we define a "reference" to an external service which we will locate via the OSGi bundle context and we tell the declarative services framework at #5 that we are interested in finding services published to the OSGi registry with the SimpleShape interface.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

For those of you familiar with other dependency injection frameworks such as Spring Beans this should feel pretty familiar, at #8 it states that when found the declarative services framework should injected the discovered service into our PaintFrame implementation by calling the addShape method on the PaintFrame. Also as we are in an OSGi environment – where services can come and go – we also define an method at #9 that will be called should that service be removed from the OSGi bundle context.

Binding method signatures

The declarative services specification defines the following method signatures for binding methods:

1. void <method-name>(ServiceReference);
2. void <method-name>(<parameter-type>);
3. void <method-name>(<parameter-type>, Map);

The first form parses in the ServiceReference to the service which allows a component to find out which services are available within the framework but not necessarily use them. This pattern should be used in conjunction with the ComponentContext which we will see a little later and can be used to implement an extremely light weight service model where services are only bound when absolutely necessary.

The second form should look familiar to most programmers who have used some form of dependency injection framework, here the service is pulled from the OSGi bundle context on behalf of the component and injected into the component via the binding method. Here the component developer may choose to store a reference to the service however they *must* be very careful to dereference that service when the corresponding unbind method is called to prevent memory leakage.

Finally the third form behaves much like the second form except that additionally the service attributes of the injected service are also provided to the binding method. As we use the attributes of our service to carry the name of the shape and it's icon it is this form that we will use. Let's look at the addShape method on our PaintFrame shown below in listing 10.7:

Listing 10.7 Method used to inject the SimpleShape service into the PaintFrame

```
void addShape(SimpleShape shape, Map attrs) {
  final DefaultShape delegate = new DefaultShape(shape);
  final String name = (String) attrs.get(SimpleShape.NAME_PROPERTY);      #1
  final Icon icon = new ImageIcon(shape.getClass().getResource((String)
attrs.get(SimpleShape.ICON_PROPERTY)));                                     #2

  synchronized( m_shapes ) {
    m_shapes.put(name, delegate);                                          #3
  }
  ...
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The declarative services framework calls the addShape method when any SimpleShape service is published to the OSGi registry, passing in the service and the map of service attributes with which the service was published. At #1 we read the name attribute of the simple shape as we did in chapter [ref] in the procedural OSGi version of the paint application. At #2 we load the ImageIcon that represents the simple shape component. As we mentioned earlier when we looked at the circle component the declarative services framework is only able to publish services with simple attribute types so in this version of the paint component we have to explicitly load the resource via the shape objects classloader. Finally at #3 we store a reference to the shape service in an internal map so it can be utilized later on by the UI thread.

When the shape component is unpublished from the OSGi registry the removeShape method (shown below in listing 10.8) is called automatically by the declarative services framework.

Listing 10.8 Method used to tell the PaintFrame that a SimpleShape service is removed

```
void removeShape(SimpleShape shape, Map attrs) {
  final String name = (String) attrs.get(SimpleShape.NAME_PROPERTY);    [1]

  DefaultShape delegate = null;

  synchronized( m_shapes ) {
    delegate = (DefaultShape) m_shapes.remove(name);                    [2]
  }

  ...
}
```

Again we use the form of binding method that supplies the service attributes as a map. We use the name attribute to figure out which component has been removed at #1 and deference the object from the internal map at #2.

METHOD ACCESSIBILITY

You may have noticed that the binding methods we defined for our paint component are set at the default "package" level visibility. In fact the declarative services specification states the following with regard to method visibility:

- public – access permitted
- protected – access permitted
- package – access permitted if method declared in implementation class or any superclass within the same package
- private – access permitted if method declared in implementation class

Even if you are using an interface based approach and the interface does not expose the binding methods then the you may still wish to protect your binding methods as this will

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

prevent external code using reflection to inject service bindings (assuming the Java security manager is enabled – we will look at security in an OSGi context in chapter [ref]).

So far we have seen how to create components and publish and consume services, however there is one use case we have not yet explored. What if your component is not a singleton within the framework and instead holds state and must be created multiple times (possibly with slightly different configurations)? Declarative services supports this model through the concept of a component factory which we will look at next.

FACTORY COMPONENTS

A trivial example of a factory component is a component that stores an authenticated session – in this case the component must be created uniquely for each consumer in order to avoid sharing authentication credentials.

In order to create a factory component you simply need to declare the component using the factory=<factory.identifier> attribute on the top level component declaration:

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
factory="session.factory" name="session">
```

This defines our component as a factory component client code that wishes to use this service should do so using the following declarative services reference:

```
<reference
  name="session"
  interface="org.osgi.service.component.ComponentFactory"
  target="(component.factory=session.factory)"
  cardinality="1..1"
  policy="static"
  bind="addSessionFactory"
  unbind="removeSessionFactory"/>
```

Blah blah

Listing 10.9 Binding to a component factory

```
package org.foo.session.client;
import org.foo.session.Session;
import org.osgi.service.component.ComponentFactory;

public class SessionClient {
  private Session session;

  void addSessionFactory(ComponentFactory factory) {
    session = (Session) factory.newInstance(null);
    session.login(null);
  }

  void removeSessionFactory(ComponentFactory factory) {
    session.logout();
    session = null;
  }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

blah blah

10.2.2 Declarative services component lifecycle

Having defined our component implementations, services and service references the next area of the design we need to consider is their lifecycle within the OSGi framework: When are components created? When are they destroyed? Are there any callback events at these stages? Can you access the BundleContext if there is no BundleActivator? We'll deal with each of these questions in this section.

Creating components

In chapter [ref] we introduced you to the life cycle of OSGi bundles, in essence bundles are installed, then resolved, then activated. Declarative services defines a similar life cycle for components where they are enabled, then satisfied, then activated. For a bundle activation implies that the BundleActivator start method has been called. In the context of declarative services an activated component implies the implementation class has been constructed (using the default no args object constructor) and if it publishes a service this is registered in the OSGi bundle context.

The lifecycle of components within a bundle extends but is also coupled to the lifecycle of the bundle itself. Once a bundle containing declarative service components has been started components within it are eligible to be activated. If a bundle is stopped then all activated components within it are automatically deactivated. Here a deactivated component implies that any service is unregistered from the OSGi bundle context and the implementation object is dereferenced making it eligible for garbage collection by the JVM.

Let's dive into some code to see what this means in practice in our paint application. Firstly we'll look at the PaintFrame class in listing 10.10:

Listing 10.10 Lifecycle related code from the declarative services PaintFrame class

```
public PaintFrame() {                                                      #1
  super("PaintFrame");
...
}
...
void activate(Map properties) {                                            #2
  Integer w = (Integer) properties.get(".width");                          #3
  Integer h = (Integer) properties.get(".height");                         #3

  int width = w == null ? 400 : w;
  int height = h == null ? 400 : h;

  setSize(width, height);                                                  #4

  SwingUtils.invokeAndWait(new Runnable() {
    public void run() {
      setVisible(true);
    }
  });
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
}

void deactivate() {                                                      #5
  SwingUtils.invokeLater(new Runnable() {
    public void run() {
      setVisible(false);
      dispose();
    }
  });
}
...
}
```

At #1 we define the default no args constructor for our component class, declarative services components classes *must* define a public no args constructor. At #2 we define a method "activate" that is a callback method invoked by the declarative services framework when the component is activated. At #5 we define a corresponding deactivate method that is called when the component is deactivated.

In fact the name "activate" and "deactivate" are simply defaults, if you wish to use a different pattern or are migrating legacy code then it is possible to define the name of these callback methods via attributes on the <component> xml declaration. For example in the following code snippet we redefine the activate and deactivate methods to be start and stop respectively:

```
<component name="org.foo.example"
           activate="start"
           deactivate="stop">
```

DECLARATIVE SERVICES ACTIVATION FAQ

In declarative services the activate and deactivate methods are optional so if your component has no need to track it's activation state then you can simply leave them out. Also if you use the default "activate" and "deactivate" method names then there is no need to define these in the component xml declaration.

So you may be wondering about the Map parsed into the activate method at #2 which we use to configure the size of the component at #3 and #4? In fact the lifecycle callbacks in declarative services accept a number of different argument types that give the component context within the wider OSGi framework. We'll look at this next.

Lifecycle method signatures

The lifecycle callback methods follow the same method accessibility rules laid out earlier in section [ref] for binding methods. They may also accept zero or more of the following argument types (order is not important):

- ComponentContext – The component instance will be passed the Component Context for the component configuration.
- BundleContext – The component instance will be passed the BundleContext of the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

component's bundle.

- Map – The component instance will be passed an unmodifiable Map containing the component properties.

In addition to this set of arguments the deactivate method may also accept an argument of type:

- int or Integer – The component instance will be passed the reason the component configuration is being deactivated. Where the integer code maps to one of the following reasons:
  - 0 – Unspecified.
  - 1 – The component was disabled.
  - 2 – A reference became unsatisfied.
  - 3 – A configuration was changed.
  - 4 – A configuration was deleted.
  - 5 – The component was disposed.
  - 6 – The bundle was stopped.

Here we have introduced you to a new interface the ComponentContext. What is this and what function does it supply? We'll look at this next.

Using the ComponentContext

Blah blah

```
public interface ComponentContext {
  public Dictionary getProperties();
  public Object locateService(String name);
  public Object locateService(String name, ServiceReference reference);
  public Object[] locateServices(String name);
  public BundleContext getBundleContext();
  public Bundle getUsingBundle();
  public ComponentInstance getComponentInstance();
  public void enableComponent(String name);
  public void disableComponent(String name);
  public ServiceReference getServiceReference();
}
```

Lookup strategy

Now you may be surprised to here that the only the interface attribute is mandatory on the reference element.

Blah blah

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Modified

The last lifecycle method defined by the declarative services specification is the modified callback event. This occurs if the components configuration is updated in the ConfigurationAdmin service. This presents the component with the option to update it's internal state without requiring the entire component to be deactivated and reactivated.

The method signature and accessibility rules for this callback method are the same as the activate method. Unlike the activate and deactivate methods there is no default method name for this lifecycle callback so if you need to deal with this use case you must define the method name in the <component> element using the "modified" attribute as shown in the following code snippet:

```
<component name="org.foo.example"
           modified="modified">
```

We've now seen how to build declarative services components and how to support lifecycle events, but in order to complete the picture we need to look at how declarative services interacts with a running OSGi framework, we'll do this next.

10.2.3 Declarative services components as part of a framework

Note at the start of section [ref] we said that starting a bundle containing components implied the components were "eligible to be activated". In fact whether a component is activated or not depends on a number of external factors:

1. Is the component enabled?
2. Is any required configuration set in the ConfigurationAdmin?
3. Are all mandatory (1..X) service references satisfied?

Let's look at each of these in turn.

Enabled

The enabled flag on a component is a mechanism to control the life cycle of individual components decoupled from the bundle life cycle. Whether a component is enabled is initially controlled via an xml attribute "enabled" of the <component> element where the default value for this is "true" if not specified. It is possible to programatically toggle the enabled state of a component via the ComponentContext interface.

A simple use case to which this can be applied is to reduce start up time. Essentially a bundle containing a number of components may enable one master component within the declarative services xml file and use this master component to toggle the enabled state of the other components in this bundle based on runtime requirements.

Configuration policy

We've seen that it is possible to configure a declarative services component by specifying an entry in the ConfigurationAdmin service with PID corresponding to the name of our declarative services component. This configuration overrides any configuration specified in the xml document and provides a way to tweek the behavior of a component at runtime.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

In fact declarative services allows the component developer to state that this configuration policy is `optional`, `require`(d) or `ignore`(d). The default configuration policy is optional implying that the declarative services framework will append (and override) any default configuration supplied by the component with configuration specified via the ConfigurationAdmin service.

However in some cases it may be that there is no sensible default value that can be given to a configuration value. A classic example of this type of configuration is that of a data base URL (there is no such thing as a default database URL so specifying one in the component xml would not make any sense). In this case it is possible to add an attribute to the <component> element stating that this component must be configured before it can be activated. The following code snippet shows how this is done:

```
<component name="org.foo.example"
           configuration-policy="require">
```

This states that the org.foo.example configuration must be registered in the ConfigurationAdmin before the component can be activated. Declarative services also allows the opposite behaviour, via `configuration-policy="ignore"` this allows the component developer to lock their component configuration at build time such that it cannot be overridden by the ConfigurationAdmin service. A potential use case for this is if the component developer wishes to lock a sensitive configuration value that the developer wishes not to hard code into the application but none the less is effectively hard coded at deployment time – such as a buffer size to prevent out of memory errors.

Mandatory services

A mandatory service reference is one that has cardinality 1..1 or 1..n, i.e. these represent tightly coupled service dependencies, for example an HttpServlet component might have a mandatory 1..1 dependency on an HttpService provider – without an HttpService to publish the servlet there's not much point in creating the servlet as there is no useful work it can do. Conversely the same HttpServlet may have an optional 0..1 dependency on a LogService – without the LogService no log events will happen but this is not critical to the application – it can still serve content to end users. Mandatory services must be available in the OSGi bundle context before the component can be activated.

Circular dependencies

Here we should draw to your attention to how declarative services handles the issue of circular dependencies between components. When designing components it is possible to get yourself into a situation where component A needs to access a service from component B and component B also needs to access a service from component A.

If both of these dependencies are mandatory then there is no way for a declarative services framework to break the cycle. Service A will not be published until component A

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

is activated and this will only happen after service B is bound. If component B is also waiting until service A is bound then the components are effectively deadlocked.

These sort of problems can be tricky to spot and often creep in my accident in large software projects. Here the declarative services specification states that the declarative services provider should detect these cyclic dependencies and log an error message with the OSGi LogService – which at least gives you a warning that there is a problem.

However it is possible to break this cycle by making one of the service dependencies optional. If we say that component A is mandatory for component B but component B is optional for component A, this allows the A component to be activated and published first which then satisfies component B which is then subsequently wired into component A.

That must be it right? What other tricks can declarative services possibly have up it's sleeve with respect to life cycle? Well it turns out there is one more...immediate and delayed components.

Immediate vs delayed components

Many components such as the shape components in our paint example exist solely to provide a function to other parts of the OSGi framework. As such if there are no other bundles deployed to the OSGi framework that consume these services then there is no need to expend resources in creating classes or threads.

Delayed components represent an elegant way of conserving JVM resources until the point at which a component is really needed – i.e. when another object within the OSGi framework is going to call a method on the service published by that component. In fact the default behavior for components that provide services within a declarative services framework is to do so as delayed components. If we look again at the component declaration of our paint frame we see that it specifies the immediate=”true” attribute:

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
  name="paint"
  immediate="true">
```

This turns off this “delayed” behavior and forces the declarative services framework to construct our paint frame before another bundle requests it's service interface. For services that do not provide a service then it is an error to set immediate to false (as they would never be instantiated) and they are implicitly defined as immediate components so the declarative services framework instantiates them as soon as the component is activated.

Declarative services achieves this delayed behavior by registering a ServiceFactory (which we looked at in chapter [ref]) in the OSGi bundle context as a proxy to the service on behalf of the component. This allows declarative services to delay all resource creation – even classloading – until the moment that the service is loaded by another bundle. In short delayed components have an extra lifecycle phase: enabled, then satisfied, then *registered*, then active.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

10.2.4 Complex service dependencies in declarative services

We've shown you how to make a trivial dependency on another service using the declarative services specification. But what if you have more complex service dependencies, for example: How are mandatory or optional services specified? How can services be filtered based on service attributes? How does declarative services help with dealing with service dynamics?

To illustrate some of these aspects of declarative services let's take a look at the WindowListener component of our modified paint program. This component has a mandatory dependency on a service published to the OSGi bundle context with the `java.awt.Window` API with the attribute `name=main` and an optional dependency on the OSGi `LogService` (to log an info message if a window close event causes the WindowListener to shutdown the OSGi framework). Listing 10.11 provides a look at the declarative services XML file that defines this component:

Listing 10.11 WindowListener with optional LogService dependency

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
  name="windowlistener">
  <scr:implementation class="org.foo.windowlistener.WindowListener" />

  <scr:reference
    name="window"
    interface="java.awt.Window"
    policy="static"                                                    #1
    cardinality="1..1"                                                 #2
    target="(name=main)"                                               #3
    bind="bindWindow"
    unbind="unbindWindow"/>
  <scr:reference
    name="logService"
    interface="org.osgi.service.log.LogService"
    policy="dynamic"                                                   #4
    cardinality="0..1"                                                 #5
    bind="bindLog"
    unbind="unbindLog"/>
</scr:component>
```

At #2 and #5 we state the cardinality of the service dependencies, possible values for this attribute are 0..1, 0..n, 1..1 or 1..n (where 0.. dependencies are optional and 1.. dependencies are mandatory). At #3 we specify an LDAP filter that will be applied to any Window services that are found in the OSGi bundle context. Finally at #1 and #4 we specify the binding policy for our references which will be explained in the following discussion.

Target reference properties

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

We saw earlier that component properties can be used to configure the service attributes published by the component. In fact component properties can also be used to configure reference filters at runtime. In order to do this the property name must be equal to the name of the reference appended with ".target". In our case this implies we could override the window target using a property of the form:

```
<property name="window.target" value="(name=other)" />
```

This will bind our window listener to windows attributed with the name=other identifier. Doing this directly in the declarative services xml is obviously of relatively low value, but if you remember the discussion on component properties these values can also be set at runtime via the ConfigurationAdmin service or via the ComponentFactory.newInstance method which brings into play a whole set of interesting use cases.

As you can see filtering services based on attributes is incredibly easy and at first glance dealing with optional services appears equally trivial. However there are some subtle mechanics that you need to be aware of particularly around the area of the policy attribute used in listing 10.11 which we'll deal with in the following side bar.

Dynamic or static policy?

A reference may be declared with either of two policy values: dynamic or static. But what does this mean? Well a dynamic policy means that the component will be notified whenever the service comes or goes. Where as with a *static* policy the service is injected once and not changed until the component is reactivated.

In essence if you use a dynamic policy then your component class needs to be able cope with the possible threading issues of a service coming or going whilst the component is running. If you use a static policy then you don't need to worry about the threading issues but your component will only see one view of the services published in the OSGi registry whilst it is activated. Or to say it another way if a reference is static and the component is satisfied (active) added services are not considered until the component is reactivated for other reasons.

Let's consider some examples, of a 0..1 dependency on a service Foo from a component Bar. In this case with a static policy: If Foo is missing when Bar is activated then it will be null until that component is reactivated. If instead Foo is available when Bar is activated and subsequently goes away the service will initially be bound, then when it is removed the Bar component will first be deactivated then the relevant unbindMethod is called then the component is reactivated with the new reference.

Listing 10.12 shows relevant lines from the declarative services WindowListener class:

Listing 10.12 WindowListener with optional LogService dependency

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
public class WindowListener extends WindowAdapter {
...
  private AtomicReference<LogService> logRef = new
AtomicReference<LogService>();                                          #1

  protected void bindLog(LogService log) {
    logRef.compareAndSet(null, log);                                    #2
  }

  protected void unbindLog(LogService log) {
    logRef.compareAndSet(log, null);                                    #3
  }
...
  private void log(int level, String msg) {
    LogService log = logRef.get();                                      #4
    if ( log != null ) {                                                #4
      log.log(level,msg);
    }
  }
}
```

Here we store the LogService reference using a java.util.concurrent.AtomicReference object at #1 and set it at #2 and #3 in our binding methods. We use an AtomicReference to protect ourselves from threading issues related to the service being bound or unbound whilst being accessed from the AWT UI thread. We also need to be wary of the fact that the LogService may in fact not be bound as it is optional – so at #4 we check whether the service is bound and log a message if so. The use of a wrapper method to achieve this is one possible mechanism – a more advanced solution could chose to use null objects to protect other areas of code from this runtime aspect of the design.

This concludes our look at the declarative services framework, we shall now turn our attention to the next component model on our list, that of the Blueprint specification.

10.3 Blueprint Services

Probably the most widely used component model in use in the world today is that of Spring Beans. In Blueprint Services the OSGi alliance has taken the base XML bean definition from Spring and standardized it in the 4.2 release of the OSGi specification in such a way that it works in an OSGi context. This specification is based heavily on the Spring Dynamic Modules framework from SpringSource but there are also several planned implementations of this specification from other vendors, including Apache Geronimo, and the recently announced Eclipse Gemini project.

There can be only one?

One could argue that the fact that the OSGi specification defines more than one component definition is confusing. However there is method in this apparent madness. Firstly both specifications are in the main (see discussion later in section [ref])

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

interoperable at runtime so both can be used interchangeably in a running framework. Secondly each specification caters for subtlety different use cases:

- Declarative services focusses on building a light weight components with quick startup time
- Blueprint focusses on building a rich component model which is familiar to current Java developers

XXX

Blah blah

Blueprint architecture

The blueprint specification defines a component in terms of a number of elements each of which is backed by an underlying manager provided by the blueprint container. Each blueprint definition can contain zero or more or any of these managers, these managers are listed in table 10.4.

Table 10.4 Blueprint container managers

Manager	Description
Bean	Provides implementation classes with same basic semantics as Spring beans, i.e.: • construction via reflective construction or static factory methods • support for singletons or prototype instances • injection of properties or constructor arguments • lifecycle callbacks for activation and deactivation
Reference	Provides a single service from the OSGi registry based on interface and optional filters of the available service attributes
Reference List	Provides one or more services from the OSGi registry based on interface and optional a filter of the available service attributes
Service	Provides a mechanism to manage any OSGi service registrations made on behalf of the component.
Environment	Provides access to the OSGi framework and the Blueprint container, including the bundle context of the component.

Blah blah

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

10.3.1 Building blueprint components

In this section we will explore how the blueprint specification can be used to build our paint application. As with declarative services let's start by looking first at the circle bundle. Again this bundle contains no activator code, but it does contain a circle.xml file which is shown below in listing 10.13.

Listing 10.13 Blueprint definition of the circle component

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">           #1
  <bean id="circle" class="org.foo.shape.circle.Circle" />               #2

  <service id="shape" interface="org.foo.shape.SimpleShape" ref="circle">#3
    <service-properties>
      <entry key="simple.shape.name" value="Circle"/>                    #4
      <entry key="simple.shape.icon">                                    #5
        <bean class="org.foo.shape.circle.IconFactory"
              factory-method="createIcon"/>                              #6
      </entry>
    </service-properties>
  </service>
</blueprint>
```

As you can see this uses a different xml syntax to that of declarative services. However those of you who have built applications using Spring beans should be at least partially familiar with this syntax. You should recognize the <bean> element at #2 where it used to define our Circle component and at #6 where it is used to specify a value for a service attribute. Equally you should be familiar with the <entry> at #4 and #5 element which is used in Spring beans to define the entries of map objects and here to define the entries of our service interface. The new features in this example which you will not be as familiar with are:

- The top level element at #1 <blueprint> (with a new namespace) compared to the <beans> from a classic spring application.
- The <service> element at #3 which is uses the "ref" element to publish the bean with id "circle" to the OSGi bundle context with a set of nested service attributes.

At first site there only appear to be syntactic differences between the declarative services version of this component and this blueprint component. However there is one big differentiator, the ability to define complex attribute objects – namely the use of another blueprint pattern – the factory bean – to create a javax.swing.ImageIcon attribute at #6. Below you can see the code required to implement our factory bean:

```
package org.foo.shape.circle;

import javax.swing.ImageIcon;

public class IconFactory {
  public static ImageIcon createIcon() {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    return new ImageIcon(IconFactory.class.getResource("circle.png"));
  }
}
```

The factory pattern allows us to create a class to call the non trivial code required to create an ImageIcon from the XML model. The factory bean pattern allows the blueprint service to create objects with non trivial constructors and use them within the component as services, parameters or service attributes. The final piece of the puzzle is to notice that the blueprint circle bundle contains a new manifest header:

```
Bundle-Blueprint:  OSGI-INF/circle.xml
```

As with declarative services this header provides the *component indicator* needed by the blueprint extender bundle to spot that this bundle is a blueprint bundle and start to manage it's lifecycle. Blueprint also supports a number of other mechanisms for supplying the service interface, the <interfaces> element and the “auto-export” attribute. To demonstrate these features consider the following trivial bean and class definition:

```
<bean id="fooImpl"  class="FooImpl"/>
public class FooImpl implements Foo { ... }
```

Given this definition then the following service definitions are equivalent:

```
<service id="foo">
  <interfaces>
    <value>com.acme.Foo</value>
  </interface>
</service>
<service id="foo" interface="com.acme.Foo" ref="fooImpl"/>
<service id="foo" auto-export="interfaces" ref="fooImpl"/>
```

The first form is a long hand form of service definition which allows a blueprint component to export more than one interface for a given bean. The second form is the short form that explicitly exports a bean using a single interface. Finally the last form is a short cut that allows the service manager to reflectively calculate the interfaces under which the bean should be registered. The possible values of the auto export attribute are:

- disabled – No auto-detection of service interface names is undertaken, the interface names must be explicitly declared. This is the default mode.
- interfaces – The service object will be registered using all of its implemented public Java interface types, including any interfaces implemented by super classes.
- class-hierarchy – The service object will be registered using its actual type and any public super-types up to the Object class (not included).
- all-classes – The service object will be registered using its actual type, all public super-types up to the Object class (not including), as well as all public interfaces implemented by the service object and any of its super classes.

Hopefully that all seems pretty straight forward? Let's now look at the PaintFrame component to see how to reference the services published by our shape components.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

10.3.2 Referencing services in blueprint

Blueprint provides two patterns of binding to services published in the OSGi registry these are: Provided Objects or Reference Listeners. Provided objects will be familiar to those of you who have developed Spring components in the past – the service or a list of services (depending on the cardinality requirements of your application) is injected as a bean property or as a constructor argument. Conversely reference listeners provide callback methods that the blueprint container may notify when services come and go from the OSGi bundle context.

The a, b, c's of blueprint references

To demonstrate these differences let's look at a few trivial cases. Firstly we will define a trivial service interface A with a method doit for client code to call:

```
public interface A {
  void doit();
}
```

Pretty straightforward I hope you'll agree, now let's look at a simple client:

```
public class B {
  private A a;
  public void setService(A a) { this.a = a }
  public void someAction() { a.doit(); }
}
```

In this example we have a class B that depends on a service A that is injected via a bean property method setService. In blueprint we can express this relationship as:

```
<reference id="a" interface="A"/>
<bean id="b" class="B">
  <property name="service" ref="a"/>
</bean>
```

Given this declaration the Blueprint container injects a java.lang.reflect.Proxy that backs onto the OSGi bundle context to find the underlying A service when it is registered by another bundle. An alternative form of this is shown below where a class C has a dependency on the service A which this time is injected via a constructor argument.

```
public class C {
  private A a;
  public C(A a) { this.a = a }
  public void someAction() { a.doit(); }
}
```

In this case the blueprint xml to express this relationship looks like this:

```
<reference id="a" interface="A"/>
<bean id="c" class="C">
  <argument ref="a"/>
</bean>
```

In both cases each client class is injected with a proxy that hides the dynamic aspects of the OSGi lifecycle. When service A is registered in the bundle context then the proxy is created. But hang on! What happens when service A is unregistered? Well in this case blueprint specifies that the proxy should block method calls with a timeout until a new service becomes available to take it's place. If no service becomes available after the timeout

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

then the proxy throws the runtime exception `org.osgi.service.blueprint.container.ServiceUnavailableException`. We'll look at the exact mechanics of this a little later.

What about if our client code aggregates services? The following example shows a class D that aggregates many A services registered in the OSGi bundle context:

```
public class D {
  private List<A> list;
  public void setServices(List<A> list) { this.list = list }
  public void someAction() {
    for ( A a : list ) {
      a.doit();
    }
  }
}
```

In this case our class D is injected with a provided object list that aggregates the services bound in the OSGi bundle context. Changes to the OSGi bundle context are reflected in the list, new services are appended to the end of the list and old services are removed. You may be looking at this code and wondering what happens the underlying services backing this list change during the course of the iteration. In fact the blueprint specification protects against this by ensuring that the `hasNext` and `getNext` operations are "safe" with respect to changes in the OSGi bundle context – that is if a service is removed when `hasNext` has already been called then a dummy object is returned that throws an `ServiceUnavailableException` when a method is called instead of throwing a `ConcurrentModificationException`. The blueprint xml to define this is as follows:

```
<reference-list id="a" interface="A"/>
<bean id="d" class="D">
  <property name="services" ref="a"/>
</bean>
```

Here the xml element <reference/> has been replaced by a new element <reference-list/>. Now compare these provided object patterns to the reference listener pattern of receiving service bind an unbind events as shown in the following example:

```
public class E {
  private volatile A a;
  public void addService(A a) { this.a = a }
  public void removeService(A a) { this.a = null }
  public void someAction() {
    A a = this.a;
    if ( a != null ) a.doit();
  }
}
```

In this case the class E receives a callback via the addService and removeService methods when the A service is registered or unregistered in the OSGi bundle context and the body of the someAction method must protect against the fact that the service may be null. We can express this relationship in blueprint xml as follows:

```
<bean id="e" class="E"/>
<reference id="a" interface="A">
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    <reference-listener
      bind-method="addService"
      unbind-method="removeService"
      ref="e"/>
  </reference>
```

So each of these patterns gives us some flexibility in how to define our code but what are the benefits or costs of each?

PROVIDED OBJECTS VS REFERENCE LISTENERS

Provided objects mask the dynamic aspects of services in an OSGi framework and as such can simplify the task of dealing with services in code. However provided objects are static so cannot be used with stateful services as you will not know when the underlying services comes or goes.

Painting with blueprint

Let's now return to our paint example, listing 10.14 provides the definition of our the paint frame using the blueprint xml syntax:

Listing 10.14 Blueprint definition of Paint frame component

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="paintFrame" class="org.foo.paint.PaintFrame"                 #1
        init-method="activate"                                           #2
        destroy-method="deactivate"/>                                    #3

  <reference-list id="shape"
                  interface="org.foo.shape.SimpleShape"
                  availability="optional">                               #4
    <reference-listener                                                  #5
      bind-method="addShape"
      unbind-method="removeShape"
      ref="paintFrame"/>                                                 #6
  </reference-list>

  <service id="window"
           interface="org.foo.windowlistener.api.Window"                 #7
           ref="paintFrame"/>
</blueprint>
```

At #1 we define our paint component and at #2 and #3 we specify the lifecycle methods that should be called by the blueprint container after it's properties have been injected. As with declarative services we'll leave the discussion of the lifecycle aspects of Blueprint till the next section (section [ref]). At #4 we use the <reference-list> element to ask the Blueprint container to monitor the OSGi context for any SimpleShape services it finds there. Nested within this reference-list element there is a <reference-listener> element used at #5 to tell blueprint to inject the service via the bind and unbind methods addShape and removeShape on the bean defined at #6 via the ref attribute which points back to our paintFrame bean.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Finally at #7 we use the <service> element to publish the paintFrame bean as a service to the OSGi registry. One point to note here is the need to publish the service as a java interface where as standard OSGi and declarative services allow (though don't recommend) you to publish a service with a java class as the service interface.

You may wonder why blueprint only allows java interfaces as service interfaces? Well the reason comes down to the mechanism via which it injects references into a bean. In fact when a service is bound to a bean from the OSGi registry in blueprint it is not the service itself which is bound but rather a java.lang.reflect.Proxy to that service and java proxies must be defined with an interface vs a class. This has impact on our paint application of forcing us to define a new interface class to provide the java.awt.Window methods we wish to use in our window listener component:

```
package org.foo.windowlistener.api;

import java.awt.event.WindowListener;

public interface Window {
  void addWindowListener(WindowListener listener);
  void removeWindowListener(WindowListener listener);
}
```

In our blueprint xml we defined the bind methods addShape and the unbind method removeShape – the code for this looks basically the same as the declarative services example (shown below in listing 10.15) but with one minor difference:

Listing 10.15 Binding method used in Blueprint component

```
public void addShape(SimpleShape shape, Map attrs) {
  final DefaultShape delegate = new DefaultShape(shape);
  final String name = (String) attrs.get(SimpleShape.NAME_PROPERTY);
  final Icon icon = (Icon) attrs.get(SimpleShape.ICON_PROPERTY);         #1
  m_shapes.put(name, delegate);

  SwingUtils.invokeAndWait( new Runnable() {
    public void run() {
      //...
    }
  });
}
```

At #1 we are given the Icon object directly vs having to look this up from the shape's classloader. As with declarative services the method signatures for these callback methods can take a number of forms which we look at in the following callout.

REFERENCE LISTENER METHOD SIGNATURES

In blueprint a reference listener callback can have any of the following signatures (note they must be public):

- public void <method-name>(ServiceReference)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

- public void <method-name>(? super T)
- public void <method-name>(? super T,Map)

We've now seen how to define component implementations, how to publish services and how to consume service references. The next area of the blueprint specification we should look at is how blueprint deals with the component life cycle.

10.3.3 Blueprint component lifecycle

As with declarative services blueprint supports the notion of: mandatory and optional service references; eager and lazy component activation; and lifecycle callbacks on bean classes.

Mandatory and optional service references

As with declarative services components it is possible to define service references in blueprint as either mandatory or optional. Any mandatory service references must be satisfied before a component can be enabled. Once a component is enabled any service managers associated with the blueprint component can register their associated service interfaces in the OSGi bundle context.

> **BLUEPRINT SERVICE REFERENCES ARE HYSTERICAL?**
>
> Unlike in declarative services if a mandatory service reference is removed from the bundle context then this does not cause the unregister of the blueprints services. Instead the services remain published in the OSGi registry until the bundle containing the blueprint definition is stopped. In this case calling methods on the published service may result in the component object receiving ServiceUnavailableExceptions from the proxy if the service reference is a provided object or an empty list if the provided object comes from a reference list. The behavior in a component object which uses the reference listener pattern is undefined but could result in NullPointerExceptions if the code is not properly protected. All in all the blueprint specification of mandatory is a very weak form of dependency checking – you have been warned!

However there are cases such as logging where the existence of the service is indeed optional and shouldn't prevent a service from being registered. Consider the following code snippet from our blueprint windowlistener xml file.

```
<reference id="log" interface="org.osgi.service.log.LogService"
availability="optional">
  <reference-listener
    bind-method="bindLog"
    unbind-method="unbindLog"
    ref="listener"/>
</reference>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Here we've labeled the reference as optional using the “availability” attribute. Possible values of this attribute are “optional” and “mandatory”. The default value of this is mandatory unless otherwise specified.

Registration Listener

The registrationListeners represent the objects that need to be called back after the service has been registered and just before it will be unregistered. The listenerComponent must be a Target object; it is the target for the following callbacks:

- registrationMethod – The name of the notification method that is called after this service has been registered.
- unregistrationMethod – This method is called when this service will be unregistered.

Damping and grace periods

Up until now barring some syntactic and minor semantic differences blueprint seems to behave in a very similar fashion to declarative services. However there are two big differentiators in the blueprint lifecycle compared to that of declarative services, namely damping and grace periods, we will look at these next.

When is a mandatory service reference not mandatory? Well according to blueprint it is not mandatory during a software upgrade. As we have already mentioned blueprint service references are actually proxies to the underlying OSGi bundle context. When a service is removed from the bundle context the blueprint proxy goes into a timeout wait to allow a new service to be published to satisfy the call.

This timeout process is known as damping and allows for bundle updates without large unpublish/publish waves rippling through the framework. Of course one may also achieve the same behavior by marking a service as optional – but then your application code has to check that the service reference is not null.

SERVICEUNAVAILABLEEXCEPTION

If the timeout passes and no new service is found to replace the missing service reference the blueprint proxy will throw a org.osgi.service.blueprint.container.ServiceUnavailableException which is a runtime exception so unchecked in your application code. Client code should therefore be defensively coded to deal with runtime exceptions gracefully – but then this is true in general and not just in blueprint.

To configure this behavior you may specify the “timeout” attribute on a service reference which specifies the number of milliseconds to wait until the service reappears. The timeout value must be equal or larger than zero and a timeout of zero implies that the service reference should wait indefinitely. From our window listener component we can see this in action:

```
<reference id="window" interface="org.foo.windowlistener.api.Window"
timeout="1">
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The grace period is a timeout that ensures that at some point during component initialization a service has been registered in the OSGi bundle context for each mandatory service reference. This allows a component that uses the provided object pattern of service references to be optimally wired together. The grace period behaviour is configured via manifest headers in the blueprint bundle:

```
Bundle-SymbolicName: com.acme.foo;
blueprint.graceperiod:=true;
blueprint.timeout:= 10000
```

In this example we state that the grace period should be used and that the blueprint container should wait 10 seconds for required services to be published to the OSGi bundle context before publishing services for this component. It is possible to disable this grace period behaviour by setting blueprint.graceperiod=false. In this case the blueprint container will not wait for any mandatory service references to be satisfied before publishing the services for any blueprint component contained in this bundle.

Lazy vs eager initialization

As with declarative services blueprint components are lazy by default and all classloading is delayed until a service published by the blueprint component is requested from the OSGi bundle context or an eager manager is activated. A blueprint manager is declared as eager or lazy using the "eager" attribute on the xml element which accepts a boolean argument, for example:

```
<bean id="foo" class="Foo" eager="true" />
<reference id="bar" interface="Bar" eager="false" />
<service id="baz" interface="Baz" eager="false" />
```

One potential gotcha when using lazy beans is if a service that references the bean specifies the auto-export attribute which we looked at in section [ref] – in this case the blueprint container must activate the underlying bean to perform classloading to calculate the service interface via reflection.

Blueprint defaults

In declarative services the default behavior of components is defined in the compendium specification. In blueprint the default behaviour of the component can be specified via a number of attributes on the <blueprint /> element:

- default-activation – The default for the activation attribute on a manager. The default for this attribute is eager.
- default-availability – The default availability of the service reference elements. The default for this attribute is mandatory.
- default-timeout – The default for the reference element timeout attribute. The default for this attribute is is 30000, or 5 minutes.

Segue

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

10.3.4 Advanced blueprint

Scope

A bean manager has a recipe for the construction and injection of an object value. However, there can be different strategies in constructing its component instance, this strategy is reflected in the scope. The following scopes are architected for this specification:

- singleton – The bean manager only holds a single component instance. This object is created and set when the bean is activated. Subsequent requests must provide the same instance. Singleton is the default scope. It is usually used for core component instances as well as stateless services.
- prototype – The object is created and configured a new each time the bean is requested to provide a component instance, that is, every call to getComponentInstance must result in a new component instance. This is usually the only possible scope for stateful objects. All inlined beans are always prototype scope.

Service manager scope

ServiceFactory is always singleton. Otherwise, if the manager that provided the service object has prototype scope, a new object will be provided for each bundle. A singleton manager will be shared between all bundles.

Environment manager

```
<bean id="listener" class="org.foo.windowlistener.SystemListener">
  <property name="bundleContext" ref="blueprintBundleContext" />
</bean>
```

blah blah

```
private BundleContext m_context;
...
public void setBundleContext(BundleContext context) {
  m_context = context;
}
```

blah blah

Manager as Value

Each manager can be the provider of component instances that act as object values. When a manager is used in an object value, then that is the manager asked to provide a component instance. The managers are specified in manager on page657. The simple example is a bean. Any inlined bean can act as an object value. For example:

```
<list>
<bean class="com.acme.FooImpl"/>
</list>
```

Some managers have side effects when they are instantiated. For example, a service manager will result in a ServiceRegistration object but it will also register a service.

```
<map>
<entry key="foo">
<service interface="com.acme.Foo">
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
<bean class="com.acme.FooImpl"/>
</service>
</entry>
</map>
```

Metadata

Blah, blah

Type converters

Blah, blah

This concludes our look at the Blueprint specification, we will now turn our attention to the last component model on our list, iPojo from the Apache Felix project.

10.4 Apache Felix iPOJO

Outside of the OSGi alliance there are a number of different component models that have been developed or ported to work in an OSGi environment including:

- Google Guice – Peaberry
- Felix iPojo
- Scala Modules

As such iPojo is yet another component model for OSGi however we believe it warrants special attention due to the novel approach it takes which significantly simplifies the task of building a service orientated components within OSGi. The biggest differentiators of iPojo over declarative services and blueprint container are: it's use of byte code weaving; Java annotations; and it's extensible Handlers API. Let's look at each of these quickly to get an overview before we show you how this applies to our paint application.

Bytecode weaving

Byte code weaving is a mechanism to instrument existing Java byte code with new byte code instructions either at compile time or runtime. Many popular libraries use this technique including AspectJ, XXX and YYY. This pattern allows developers to focus on their business logic without worrying about domain specific issues, such as data synchronization or networking protocols which can be slotted in later. iPojo uses byte code weaving at compile time to handle the complex threading issues related to managing and accessing dynamic services in an OSGi environment.

Annotations vs XML

Both Declarative Services and Blueprint Container each employ a separate XML file embedded within the bundle to describe the meta model of their components. There are a number of problems with this model, the main one being the issue of keeping this file up to date with respect to changes to the underlying java code. IPojo supports the XML model of component declaration but also importantly introduces a number of Java annotations that express the component aspects of a Java class right there in the code.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

HANDLERS

The first thing to note about building components in iPojo is that the component definition is creating a template of a component that can be instantiated. In this sense every iPojo component is equivalent to a declarative services component factory. A component is encapsulated in a so called "container" that manages the connectivity of the Java class that provides the implementation to the outside world. The container is extensible via a pluggable API called a Handler as shown in figure 10.4:

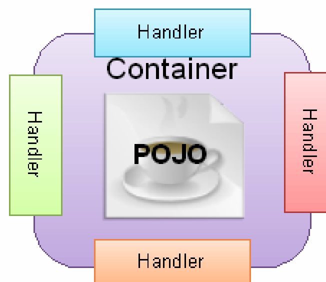


Figure 10.4 iPojo components are an aggregation of Handlers that are managed at runtime as a unit referred to as a container.

In fact all of the typical component functions of such as service publication, wiring in service dependencies, making lifecycle callbacks and configuring components are handled via a default set of Handlers that are always attached to a component definition. In general a handler should typically manage one specific functional concern, for example handling persistence characteristics of a component or providing audit of method invocations, this allows complex component definitions to be composed via aggregation.

10.4.1 Building iPojo components

For every architectural aspect of an iPojo component there is both an XML representation and an equivalent Java annotation. As this chapter has already been pretty XML heavy so far we'll change the pace by looking at the annotated view of iPojo for information on the XML syntax you can refer to the iPojo documentation found at [ref].iPojo defines the following annotations described in table 10.5:

Table 10.5 iPojo annotations and their uses

Annotation	Description
@Component	Class level annotation declaring class as a component class

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

@Provides	Class level annotation declaring that this class provides a service
@Requires	Field level annotation declaring that this field is a service which should be injected from the OSGi bundle context
@ServiceProperty	Field level annotation declaring that this field should be added as a service property to this components services
@Property	Field level annotation declaring that this field should be configured from component properties
@Updated	Method level annotation declaring a callback method invoked when configuration is complete
@Bind	Method level annotation declaring a callback method invoked when a service is available to be injected from the OSGi bundle context
@Unbind	Method level annotation declaring a callback method invoked when a service previously bound to this component is removed from the OSGi bundle context.
@Validate	Method level annotation declaring a callback method invoked when all handlers report the valid state
@Invalidate	Method level annotation declaring a callback method invoked when one or more handlers report the invalid state

Building iPojo components

iPojo uses byte code manipulation to instrument java class files with functionality to work in an OSGi services environment. The simplest way to achieve this is via a build time step to parse your business objects and their component definitions. To do this iPojo provides integrations with Ant, Maven and Eclipse (the ant task is shown below for our circle bundle).

```
<taskdef name="ipojo"
         classname="org.apache.felix.ipojo.task.IPojoTask"
         classpath="${lib}/felix/org.apache.felix.ipojo.ant-1.4.0.jar" />
<ipojo input="${dist}/${ant.project.name}-${version}.jar"
       metadata = "OSGI-INF/circle.xml"/>
```

Once this build step has been completed you can verify that the manipulation has taken place by inspecting the manifest headers of your bundle. It will now contain a header:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

iPOJO-Components that provides a declaration of every component defined by this bundle (the example below is taken from our manipulated circle bundle).

```
iPOJO-Components: instance { $component="org.foo.shape.circle.Circle"
 }component { $classname="org.foo.shape.circle.Circle" $public="true"
 $immediate="true" $name="org.foo.shape.circle.Circle" provides { prop
 erty { $field="m_name" $name="simple.shape.name" }property { $field="
 m_icon" $name="simple.shape.icon" }}manipulation { interface { $name=
 "org.foo.shape.SimpleShape" }field { $type="javax.swing.ImageIcon" $n
 ame="m_icon" }field { $type="java.lang.String" $name="m_name" }method
  { $name="$init" }method { $arguments="{java.awt.Graphics2D,java.awt.
 Point}" $name="draw" }}}
```

This header may look quite verbose – compared to the headers defined for declarative services, and blueprint container. In fact this header is a compact representation of all the component meta data which allows the iPojo framework to optimize start up time by delaying all classloading until the very last moment.

As with our other component models there are three main architectural tasks that a component developer wishes to achieve with iPojo: publishing services; consuming services; and configuring the component.

Providing services

Let's look at the Java code used to define our circle component shown below in listing 10.16:

Listing 10.16 iPojo declaration of circle component type using annotations

```
import org.apache.felix.ipojo.annotations.Component;
import org.apache.felix.ipojo.annotations.Provides;
import org.apache.felix.ipojo.annotations.ServiceProperty;
...
@Component(immediate=true)                                            #1
@Provides                                                             #2
public class Circle implements SimpleShape {

  @ServiceProperty(name=SimpleShape.NAME_PROPERTY)                    #3
  private String m_name = "Circle";

  @ServiceProperty(name=SimpleShape.ICON_PROPERTY)                    #4
  private ImageIcon m_icon = new
ImageIcon(this.getClass().getResource("circle.png"));

  public void draw(Graphics2D g2, Point p) {
    ...
  }
}
```

At #1 we define our circle class as an iPojo component, the immediate flag causes iPojo to create the component whether or not a service reference has been requested elsewhere in the framework. At #2 we state that the component provides a service, in this case leaving

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

iPojo the task of registering the component under all implemented interfaces (namely SimpleShape in this case). At #3 and #4 we use the ServiceProperty annotation to state that the value of the fields m_name and m_icon should be published as the service attributes of this component. Note as annotations are part of the Java source code it is possible to use the static constant fields for the attribute names where as in declarative services and blueprint container we had to externalize these names as there expanded names – this greatly reduces the risks of code rot due to changing attribute names.

Immediate components and service properties

Just as with declarative services and blueprint container, iPojo delays classloading for as long as possible by using a ServiceFactory to register the service interface in the OSGi bundle context. However this delayed loading has an unexpected side effect when using the @ServiceProperty annotation. As discussed above the @ServiceProperty causes iPojo to monitor the field values of our component object and add the attributes to the service registration.

However if the component is a delayed component (which is the default behaviour) then iPojo does not have a field to monitor so the service is initially registered with no attributes. Then when the service is requested the component class is constructed which causes the field values to be added to the service.

In order to rectify this situation components that use the @ServiceProperty should declare themselves as immediate components such that iPojo eagerly constructs the component class and therefore adds the correct attributes to the service registration.

In listing 10.16 we used a short cut form of the @Provides annotation that tells iPojo to export the component using all implemented interfaces of the component class, including any interfaces specified in the inheritance tree. However it is also possible to export a specific interface or an abstract or concrete class as the service interface by specifying the provided interface as shown in the following code snippet:

```
@Provides(specifications=java.awt.Window.class)
public class PaintFrame extends JFrame implements MouseListener,
MouseMotionListener {
```

Let's now turn our attention to the other side of the equation, how iPojo connects components to services from the OSGi bundle context. iPojo defines two mechanisms of injecting services into a component class: at method level via @Bind and @Unbind and at field level via @Required. Both mechanisms share a common attribute of “binding policy” which controls how and when a component sees it dependencies bound. Binding policy is an enumeration containing the following values (where the default value is `dynamic` if not specified):

- `static` - Static dependencies cannot change without invalidating the component

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

instance, so injected services typically do not change at run time and service departures typically result in the component instance being destroyed and potentially recreated

- `dynamic` - Dynamic dependencies can change without invalidating the component instance, so injected services can change at run time, but do not change with respect to service priority changes (i.e., they *do not* automatically switch if a higher priority service appears)
- `dynamic-priority` - Dynamic priority dependencies can change without invalidating the component instance and *do* dynamically update based on service priority rankings at run time

We'll first turn our attention to method level injection as this is most similar to the mechanisms we found in Declarative Services and Blueprint container.

Service bindings

iPojo defines two annotations @Bind and @Unbind that can be applied to methods with any of the following signatures:

1. void <method-name>() - no args
2. void <method-name>(ServiceReference ref) – the service reference from the OSGi bundle context
3. void <method-name>(Service svc) – the service from the OSGi bundle context
4. void <method-name>(Service svc, ServiceReference ref) – as above
5. void <method-name>(Service svc, Map properties) – the service from the OSGi bundle context and a map of it's service attributes
6. void <method-name>(Service svc, Dictionary properties) – the service from the OSGi bundle context and a java.util.Dictionary of it's service attributes

In cases 1 and 2 it is necessary to state the service interface to which this method applies using the `specification` parameter to the annotation but in all other cases iPojo will infer the service interface from the method signature. Let's look at some examples, firstly the bind methods in our iPojo window listener class shown in listing 10.17.

Listing 10.17 Bind and unbind methods from the iPojo WindowListener

```
@Bind(filter="(name=main)")                                               #1
protected void bindWindow(Window window) {                                #2
  m_log.log( LogService.LOG_INFO, "Bind window" );
  window.addWindowListener(this);
}

@Unbind(filter="(name=main)")                                             #3
protected void unbindWindow(Window window) {                              #4
  m_log.log( LogService.LOG_INFO, "Unbind window" );
  window.removeWindowListener(this);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
}
```

Here we simply annotate the bind and unbind methods right there in the Java code at #1 and #3 and iPojo infers that it should create ServiceListeners on our behalf using java.awt.Window as the service interface from the method declarations at #2 and #4. As in our previous examples with declarative services and blueprint container we also state that this service should be published with the name=”main” service attribute. So this specifies a mandatory (1..1) dependency on a service published with a service attribute. How about services with multiple cardinalities as in our shape components? Well let's look at the iPojo declaration of the PaintFrame shape binding methods shown in listing 10.18:

Listing 10.18 Bind and unbind methods from the iPojo PaintFrame

```
@Bind(aggregate=true,id="shape")                                        #1
public void addShape(SimpleShape shape, Map attrs) {                    #2
  final DefaultShape delegate = new DefaultShape(shape);
  final String name = (String) attrs.get(SimpleShape.NAME_PROPERTY);
  final Icon icon = (Icon) attrs.get(SimpleShape.ICON_PROPERTY);

  m_shapes.put(name, delegate);

  SwingUtils.invokeAndWait( new Runnable() {
    public void run() {
      ...
    }
  });
}

@Unbind(aggregate=true,id="shape")                                      #3
public void removeShape(SimpleShape shape, Map attrs) {                 #4
  final String name = (String) attrs.get(SimpleShape.NAME_PROPERTY);

  DefaultShape delegate = null;

  synchronized( m_shapes ) {
    delegate = (DefaultShape) m_shapes.remove(name);
  }
}
```

Here we define a bind and unbind method for our SimpleShape services at #1 and #3 in order to prompt iPojo to bind more than one service from the OSGi bundle context we specify the aggregate=true parameter on the annotations which tells iPojo that our PaintFrame component aggregates SimpleShape services from the OSGi bundle context.

ID'S

Note we have specified an id=shape parameter on these binding methods where as we did not do this for our previous window binding. What does this mean? Well in fact iPojo automatically infers bind and unbind pairs based on method names:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

if the id is already defines in a "@requires " or "@unbind" annotation, it adds this method as a bind method of the already created dependency. (optional, default= no id, compute an id if the method name begin by "bind" (for instance "bindFoo" will have the "Foo" id))

Requiring services

All the component frameworks we have covered in this chapter provide tools to simplify the task of accessing OSGi services, via dependency injection semantics based on java reflection. However accessing services is only one part of the challenge, another is dealing with the dynamic nature of services in an OSGi framework. If services can come and go at any point a service consumer must protectively code against this. One simple way to do this is to use mandatory dependencies such that a component is not itself accessible unless all of it's dependencies are satisfied. However this only works if the XXX

However they do nothing to simplify the task of dealing with the dynamic nature of services in an OSGi environment.

```
@Requires(optional=true)
private LogService m_log;
...
```

Blah blah

```
@Override
public void windowClosed(WindowEvent evt) {
  try {
    m_log.log( LogService.LOG_INFO, "Window closed" );
    m_context.getBundle(0).stop();
  } catch (BundleException e) {
  }
}
```

Nullable, Blah blah

Blah blah

Byte code weaving/synchronization

optional=true

Creating components

```
<?xml version="1.0" encoding="UTF-8"?>
<iPOJO>
  <instance component="org.foo.shape.circle.Circle"/>
</iPOJO>
```

ConfigurationAdmin, org.apache.felix.ipojo.Factory, @Instance (iPojo 1.6)

10.4.2 iPojo component lifecycle

Callback methods

```
@Validate
protected void activate()
{
  SwingUtils.invokeAndWait(new Runnable() {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    public void run() {
      setVisible(true);
    }
  });
}
```

Blah, blah

```
@Invalidate
protected void deactivate()
{
  SwingUtils.invokeLater(new Runnable() {
    public void run() {
      setVisible(false);
      dispose();
    }
  });
}
```

Invalidate is called after unbind – so need to be wary of null services!

Framework context

```
@Component(immediate=true)
public class SystemListener extends WindowAdapter {

  private BundleContext m_context;

  public SystemListener(BundleContext context) {
    m_context = context;
    log( LogService.LOG_INFO, "Created " + this );
  }
```

Blah blah

10.4.3 iPojo Handlers

An external handler is identified by a namespace. This namespace will be used by developers to refer to the external handler (when he configures its component type) and by iPOJO to instantiate the handler object.

org.apache.felix.ipojo.MethodInterceptor

org.apache.felix.ipojo.FieldInterceptor

org.apache.felix.ipojo.InstanceStateListener

org.apache.felix.ipojo.FactoryStateListener

org.apache.felix.ipojo.Handler

org.apache.felix.ipojo.PrimitiveHandler

Blah blah

10.5 Mix and match

Benefits:

Choose the right model for your module and others can work to their own environments. Demonstrate deploying ds, bp, ipojo in same framework

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Feature	Declarative Services	Blueprint	IPojo
Dependency injection			
Callback injection	Yes	No	Yes
Constructor injection	No	Yes	No
Field injection	No	No	Yes
Setter injection	Yes	Yes	Yes
Proxy injection	No	Yes	No
List injection	No	Yes	Yes
Nullable injection	No	No	Yes
Lifecycle			
Callbacks (activate/deactivate)	Yes	Yes	Yes
Factory pattern	Yes	Yes	Yes
Lazy initialization	Yes	Yes	Yes
Damping	No	Yes	No
Field synchronization	No	No	Yes
Configuration			
Property Configuration	No	Yes	Yes
Field Configuration	No	No	Yes
Configuration Admin	Yes	No	Yes
Services			

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Custom attribute type	No	Yes	Yes
Lazy initialization via ServiceFactory	Yes	Yes	Yes
Model			
XML	Yes	Yes	Yes
Java Annotations	No	No	Yes

Gotcha's

DS only supports simple service attributes

BP parses in a proxy so be careful with calls such as service.getClass() as this will not return the result you expect – namely service.getClass().getResource does not work as expected!!

BP damping can cause some odd hangs

10.6 Summary

Component models can simplify the task of dealing with the OSGi framework and each add useful capabilities including lazy initialization, management of complex service dependencies, XXX and YYY, that you may otherwise end up coding yourself in your applications.

- Declarative services is the simplest framework offering the key tools needed to build dynamic service environment but few extra frills
- Blueprint is very rich and is a useful step if you are migrating an existing Spring application to an OSGi environment but it also has some *odd* design decisions namely the fact that the removal of services on mandatory references does not cause published services to be unpublished
- iPojo offers the most rounded and complete framework for building a dynamic services based application but it's use of byte code engineering may turn off some users and it is not a specification so you are reliant on a single (all be it open source) implementation

We've also demonstrated that it is possible to mix and match component frameworks in a single OSGi framework in the end you can pick the component model that suits you and this will not effect other parts of the application. With these tools you will be able to build rich dynamic applications in an OSGi framework. In the next chapter we will look at how to launch and/or embed an OSGi framework within an existing Java process.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

11
Launching and Embedding an OSGi Framework

We've spent a lot of time talking about creating, deploying, and managing bundles and services. Interestingly, we can't do anything with these unless we have a running OSGi framework. For such an important and necessary topic, we've spent very little time discussing how precisely to achieve it. Not only is it necessary, but by learning to launch the framework, we'll have the ability to create custom launchers tailored to our application's needs. It even opens up new use cases, where we can actually use an instance of an OSGi framework inside an existing application or even embedded inside a bunde. Very interesting stuff.

In this chapter we'll learn everything we need to know about launching the OSGi framework. To help us reach this goal, we will dissect the generic bundle launcher we've been using to run the book's examples. We'll also refactor our paint program to show how to embed a framework instance inside an existing application. Let's get going.

11.1 Standard launching and embedding

As we mentioned back in chapter 3, we face a dilemma when we want to use a bundle we have created. We need a `BundleContext` object to install our bundle into the framework, but the framework only gives a `BundleContext` object to an installed and started bundle.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

So, we are in a chicken-and-egg situation where we need to be an installed and started bundle in order to install and start our bundle. We need some way to bootstrap the process.

Traditionally, OSGi framework implementations from Apache Felix, Equinox, and Knopflerfish devised implementation-specific means for dealing with this situation. This typically involved some combination of auto-deploy configuration properties for each framework implementations' custom launchers and/or shells with textual or graphical interfaces. These mechanisms worked reasonably well, but weren't portable across framework implementations.

With the release of the OSGi R4.2 specification, the OSGi Alliance has defined a standard framework launching and embeddin API. While this is not a major advance in and of itself, it does help us create applications that are truly portable across framework implementations.

You might be wondering if this is really necessary or all that common. The truth is, it is fairly common since your application may have specific configuration requirements that cannot be easily addressed by the default launcher of a framework implementation. Surprisingly, even the need to create an instance of a framework to use within another application is not that uncommon, because in some legacy situations are are not free to rewrite everything as a bundle. Previously, if either of these use cases applied to you, then you had to tie your application to a specific framework implementation by using its custom API for launching it.

Now, R4.2 compliant frameworks share a common API for creating, configuring, and starting the framework. Let's dive into its details.

11.1.1 Framework API overview

As we previously mentioned, at execution time the OSGi framework is internally represented as a special bundle, called the system bundle, with bundle identifier zero. This means active bundles are able to interact with the framework using the standard `Bundle` interface, which we reiterate in Listing 11.1.

Listing 11.1 Standard `Bundle` interface

```
public interface Bundle {
  int getState();
  void start() throws BundleException;
  void start(int options) throws BundleException;
  void stop() throws BundleException;
  void stop(int options) throws BundleException;
  void update() throws BundleException;
  void update(InputStream input) throws BundleException;
  void uninstall() throws BundleException;
  BundleContext getBundleContext();
  long getBundleId();
  URL getEntry(String path);
  Enumeration getEntryPaths(String path);
  Enumeration findEntries(String path, String pattern, boolean recurse);
  Dictionary getHeaders();
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    Dictionary getHeaders(String locale);
    long getLastModified();
    String getLocation();
    ServiceReference[] getRegisteredServices();
    URL getResource(String name);
    Enumeration getResources(String name) throws IOException;
    ServiceReference[] getServicesInUse();
    Map getSignerCertificates(int signersType);
    String getSymbolicName();
    Version getVersion();
    boolean hasPermission(Object permission);
    Class loadClass(String name) throws ClassNotFoundException;
  }
```

While this provides an internal framework API for other bundles, it doesn't really help externally when we want to create and start framework instances. So when the R4.2 specification looked to address this situation, the logical place to start was with the `Bundle` interface. The `Bundle` interface is a good starting point for interacting with the framework, but it isn't completely sufficient. To address the missing pieces, the R4.2 specification defines a new `Bundle` subtype, called `Framework`, which is captured in the following snippet:

```
  public interface Framework extends Bundle {
    void init() throws BundleException;
    FrameworkEvent waitForStop(long timeout) throws InterruptedException;
  }
```

All R4.2 compliant framework implementations will now implement the `Framework` interface. Since it extends `Bundle`, this means framework implementations now look like a bundle externally as well as internally via the system bundle.

NOTE

While this new API represents the framework instance internally and externally as a `Bundle` object, the specification does not require the internal system bundle object to be the same as the external `Framework` object. Whether this is or is not the case is framework implementation dependent.

As you can see, the `Framework` interface is a simple extension, so we don't have too much new API to learn. In the following subsections, we will fully explore how to use this API to configure, create, and control framework implementations in a standard way.

11.1.2 Creating a framework instance

While it is great to have a standard interface for framework implementations, we can't instantiate an interface so we need a way to get a concrete implementation class. It is not possible for the OSGi specification to define a standard class name, so it adopted the standard Java approach for specifying service provider implementations in JAR files, which is `META-INF/services`.

In this case, `META-INF/services` refers to a directory entry in a JAR file. Just like a JAR's `META-INF/MANIFEST.MF` file contains metadata about the JAR file, so does its `META-`

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

INF/services directory. More specifically, it contains metadata about the service providers contained in a JAR file. Here the term "service" is not referring to an OSGi service, but to well-known interfaces and/or abstract classes in general. All in all, the concept here is very similar to the OSGi service concept.

The META-INF/services directory in a JAR file contains service provider configuration files, which refer to a concrete implementation class for a given service. Concrete service implementations are connected to their abstract service type via the name of the file entry in the directory, which is named after the fully qualified service it implements. For example, a service implementation for the java.io.spi.CharCodec service would be named:

```
META-INF/services/java.io.spi.CharCodec
```

Where the contents of this file is the name of the concrete implementation class, such as:

```
org.foo.CustomCharCodec
```

To find and create service providers, service provider configuration files are retrieved like any normal resource using the standard service name as the name of the resource file. Once a concrete type is obtained from the content of the file, the associated class can be loaded and instantiated.

The OSGi specification leverages this mechanisms to provide a standard way to get the concrete framework implementation class. However, rather than directly retrieving a framework implementation class, OSGi defines a simple framework factory service as follows:

```
public interface FrameworkFactory {
  Framework newFramework(Map config);
}
```

This interface provides a simple way to create new framework instances and pass a configuration map into them. As a concrete example, the Apache Felix framework implementation has the following entry in its JAR file declaring its service implementation:

```
META-INF/services/org.osgi.framework.launch.FrameworkFactory
```

The content of this JAR file entry is the name of the concrete class implementing the factory service:

```
org.apache.felix.framework.FrameworkFactory
```

Of course, these details are only for illustrative purposes, since we only need to know how to get a framework factory service instance. The standard way to do this in Java 6 is to use java.util.ServiceLoader. We obtain a ServiceLoader instance for framework factories like this:

```
ServiceLoader<FrameworkFactory> factoryLoader =
    ServiceLoader.load(FrameworkFactory.class);
```

Using the ServiceLoader instance referenced by factoryLoader, we are able to iterate over all available OSGi framework factory services using like this:

```
Iterator<FrameworkFactory> it = factoryLoader.iterator();
```

In most cases, we only really care if there is a single provider of the factory service, so we can just invoke it.next() to get the first available factory and use FrameworkFactory.newInstance() to create a framework instance. If you are not using

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Java 6, you can also use the `ClassLoader.getResource()` method as illustrated in Listing 11.2.

Listing 11.2 Retrieving a FrameworkFactory service manually

```
private static FrameworkFactory getFrameworkFactory() throws Exception {
  URL url = Main.class.getClassLoader().getResource(
    "META-INF/services/org.osgi.framework.launch.FrameworkFactory");      #1
  if (url != null) {
    BufferedReader br =
      new BufferedReader(new InputStreamReader(url.openStream()));        #2
    try {
      for (String s = br.readLine(); s != null; s = br.readLine()) {
        s = s.trim();
        if ((s.length() > 0) && (s.charAt(0) != '#')) {
          return (FrameworkFactory) Class.forName(s).newInstance();       #3
        }
      }
    } finally {
      if (br != null) br.close();
    }
  }
  throw new Exception("Could not find framework factory.");               #4
}
```

The `getFrameworkFactory()` method in Listing 11.2 is not as robust as it could be, but it is sufficient to get the job done. At (#1) it queries for the standard service provider configuration file. If it finds one, it reads the contents of the file at (#2). Within the loop, it searches for the first line not starting with '#' (i.e., the comment character) and assumes that line contains the name of the concrete class it should instantiate at (#3). The method will throw an exception if an errors occur during the above or at (#4) if a factory provider could not be found. Fairly simple and this one method will work for all R4.2 compliant frameworks; we will use this for our generic launcher in section [ref 12.2]. Next we will look into how we use the factory service to configure a framework instance.

11.1.3 Configuring a framework

Once we have a framework factory service, we are able to create an instance of `Framework`. Typically, we do not use a framework instance as is, but we want to configure it in some way, such as setting the directory where the framework should store cached bundles. This is why `FrameworkFactory.newInstance()` takes a `Map`, so we can pass in configuration properties for the created framework instance.

NO CONFIGURATION REQUIRED

You do not have to pass in a configuration when creating a factory, `null` is an acceptable configuration. The OSGi specification says framework implementations must use reasonable defaults; however, it does not explicitly define all of them. This means some defaults are implementation specific. For example, by default the Apache Felix framework

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

caches installed bundles in a `felix-cache/` directory in the current directory, while the Equinox framework uses `configuration/org.eclipse.osgi/bundles/` in the directory where the Equinox JAR file is located. So be aware, you will not necessarily get the same behavior unless you explicitly configure it.

Prior OSGi specifications defined a few standard configuration properties, but until the framework factory API, there was no standard way to set them. As part of the R4.2 specification process, several new standard configuration properties have been introduced. Table 11.1 introduces some of the new standard configuration properties and a few of the existing ones.

Table 11.1 Some standard OSGi framework configuration properties

Property name	New	Meaning
org.osgi.framework.storage	R4.2	A file system path to a directory, which will be created if it does not exist. If this property is not set, a reasonable default is used.
org.osgi.framework.storage.clean	R4.2	Specifies if and when the storage area for the framework should be cleaned. If no value is specified, the framework storage area will not be cleaned. Currently, the only possible value is `onFirstInit`, which causes the framework instance to clean the storage area the first time it is used.
org.osgi.framework.system.packages		Using standard `Export-Package` syntax, specifies a list of class path packages to be exported from the system bundle. If not set, the framework must provide a reasonable default for the current VM.
org.osgi.framework.system.packages.extra	R4.2	Specifies a list of class path packages to be exported from the system bundle in addition to those from the previous system packages property.
org.osgi.framework.startlevel.beginning	R4.2	Specifies the beginning start level of the framework.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

org.osgi.framework.bootdelegation		Specifies a comma-delimited list of packages with potential wildcards to make available to bundles from the class path without `Import-Package` declarations (e.g., `com.sun.*`). By default, all `java.*` packages are boot delegated. We recommend avoiding this property.
org.osgi.framework.bundle.parent	R4.2	Specifies which class loader is used for boot delegation. Possible values are `boot` for the boot class loader, `app` for the application class loader, `ext` for the extension class loader, and `framework` for the class loader of the framework. The default is `boot`.
org.osgi.framework.library.extensions	R4.2	A comma separated list of additional library file extensions that must be used when searching for native code.
org.osgi.framework.command.execpermission	R4.2	Specifies an optional OS specific command to set file permissions on a bundle's native code.

All of the properties listed in Table 11.1 can be put into a `Map` and passed into the `FrameworkFactory.newInstance()` method to configure the resulting framework instance; property names are case insensitive. We will not go into the precise details of all the standard configuration properties, so consult the R4.2 specification if you want details not covered here. With this knowledge, we know how to configure and instantiate a framework instance, so now we can look at how to start it.

11.1.4 Starting a framework instance

Once we have a `Framework` instance from `FrameworkFactory`, starting it is quite easy, just invoke the `start()` method inherited from the `Bundle` interface. The `start()` method implicitly initializes the framework by invoking the `Framework.init()` method, unless you explicitly initialize it beforehand. If the `init()` method was not invoked prior to calling `start()`, then it is invoked by `start()`. We can relate these methods to the framework lifecycle transitions, similar to the normal bundle lifecycle:

- `init()` transitions the framework instance to the `Bundle.STARTING` state.
- `start()` transitions the framework instance to the `Bundle.ACTIVE` state.

The `init()` method gets the framework ready, but does not actually start executing any bundle code yet; it performs the following steps:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

7. Framework event handling is enabled.
8. The security manager is installed if it is enabled.
9. The framework start level is set to 0.
10. All cached bundles are reloaded and their state is set to `Bundle.INSTALLED`.
11. A `BundleContext` object is created for the framework.
12. All framework-provided services are made available (e.g., Package Admin, Start Level, etc.).
13. The framework enters the `Bundle.STARTING` state.

The `start()` method actually starts the framework instance and performs the following steps:

1. If the framework is not in the `Bundle.STARTING` state, then the `init()` method is invoked.
2. The framework sets its beginning start level to the configured value, which causes all reloaded bundles to be started in accordance with their activation policy and start level.
3. The framework's state is set to `Bundle.ACTIVE`.
4. A framework event of type `FrameworkEvent.STARTED` is fired.

You may wonder why the `init()` method is necessary and why all the steps just aren't performed in the `start()` method? There are some cases where you may want to interact with the framework instance before re-starting cached bundles, but some interactions can only happen via the framework's `BundleContext` object. Since bundles (including the framework) do not have a `BundleContext` object until they have been started, `init()` is necessary to transition the framework to the `Bundle.STARTING` state so you can acquire its context with `Bundle.getBundleContext()`.

To summarize, in the simple case, just call `start()`. But if you want to do some actions before all of the cached bundles re-start, then call `init()` first to do what you need to do followed by a call to `start()`. Once the framework is active, subsequent calls to `init()` and `start()` have no effect. Next we will look at how we shutdown a running framework.

11.1.5 Stopping a framework instance

As you might guess, stopping an active framework simply involves invoking the `stop()` method inherited from the `Bundle` interface. This method asynchronously stops the framework on another thread, so the method returns immediately to the caller. If you want to know when the framework has actually finished shutting down, you should call `Framework.waitForStop()` after calling `stop()`, which blocks the calling thread until shutdown is complete.

The following steps are performed when stopping a framework:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

1. The framework's state is set to `Bundle.STOPPING`.
2. All installed bundles are stopped without changing each bundle's persistent activation state and according to start levels.
3. The framework's start level is set to 0.
4. Framework event handling is disabled.
5. The framework's state is set to `Bundle.RESOLVED`.
6. All resources held by the framework are released.
7. All threads waiting on `Framework.waitForStop()` are awakened.

WAIT MEANS WAIT

Calling `waitForStop()` does not start the framework shutdown process, it simply waits for it to occur. If you want to stop the framework, you must call `stop()` first.

The `waitForStop()` method takes a timeout value in milliseconds and returns a `FrameworkEvent` object whose type indicates why the framework stopped; these values include:

- `FrameworkEvent.STOPPED` – The framework was stopped.
- `FrameworkEvent.STOPPED_UPDATE` – The framework was updated.
- `FrameworkEvent.ERROR` – An error forced the framework to shutdown or an error occurred while shutting down.
- `FrameworkEvent.WAIT_TIMEDOUT` – The timeout value has expired before the framework stopped.

Once the framework has successfully stopped, it can be safely discarded or reused. To start the framework again, simply call `start()` or `init()/start()`. The normal startup process will commence, except the bundle cache will not be deleted again if the storage cleaning policy is `onFirstInit`, since it only applies to the first time the framework is initialized. Otherwise, you can stop and re-start the framework as much as you like.

That's all there is to creating and launching frameworks with the standard framework launching and embedding API in R4.2. Let's explore our newfound knowledge by examining our generic bundle launcher.

Launching versus embedding

Why is this called the framework launching and embedding API? The term "launching" is largely self explanatory, but the term "embedding" is less clear. What is the difference between the two? The conceptual difference is launching refers to creating and starting a framework instance in isolation, while embedding refers to creating and starting a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

framework instance within (i.e., embedded in) another application. Technically, there is very little difference between the two, since creating, configuring, and starting a framework instance with the API is the same in either case.

The main technical differences are in your objectives. When you launch a framework, all functionality is typically provided by installed bundles and there is no concern about the outside world. However, when you embed a framework, you often have functionality on the outside you want to expose somehow on the inside or vice versa. Embedding a framework instance has some additional constraints and complications we will discuss later in this chapter.

11.2 Launching the framework

The general steps for launching the framework are fairly straightforward; we want to set the desired configuration properties, create a framework instance using the configuration properties, start the framework instance, and install some bundles. These are the same basic steps our generic bundle launcher uses, which we'll introduce in the following subsections by breaking the example into short code snippets. The complete source code for the generic launcher is in the `code/launcher/` directory of the companion code.

11.2.1 Determining which bundles to install

As we've seen throughout the book, our generic bundle launcher installs and starts all bundles contained in a directory specified as a command-line argument. The launcher is composed of a single class, called `Main`, which is defined in the snippet in Listing 11.3.

Listing 11.3 Main class definition for generic bundle launcher

```
public class Main {
  private static Framework fwk;                                         #1

  public static void main(String[] args) throws Exception {
    if (args.length < 1 || !new File(args[0]).isDirectory()) {          #2
      System.out.println("Usage: <bundle-directory>");
    } else {
      File[] files = new File(args[0]).listFiles();                     #3
      Arrays.sort(files);                                               #3
      List jars = new ArrayList();                                      #3
      for (int i = 0; i < files.length; i++)                            #3
        if (files[i].getName().endsWith(".jar"))                        #3
          jars.add(files[i]);                                           #3
      ...
```

The static member variable at (#1) will hold the framework instance we are going to create. At (#2) we verify that a directory was specified as a command line argument. If a directory was specified, we get the files contained in it and save all files ending with ".jar" into a list to be processed later.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

11.2.2 Shutting down cleanly

We cannot always guarantee that our launcher process will exit normally, so it is a good idea to try to ensure our framework instance cleanly shutdowns. Depending on the framework implementation, you could end up with a corrupted bundle cache if you do not shut down cleanly. The code snippet in Listing 11.4 adds a shutdown hook to our JVM process to cleanly shut down our framework instance.

Listing 11.4 Using a shutdown hook to cleanly stop the framework

```
...
if (jars.isEmpty()) {
  System.out.println("No bundles to install.");
} else {
  Runtime.getRuntime().addShutdownHook(new Thread() {              #1
    public void run() {
      try {
        if (fwk != null) {                                         #2
          fwk.stop();                                              #3
          fwk.waitForStop(0);                                      #4
        }
      } catch (Exception ex) {
        System.err.println("Error stopping framework: " + ex);
      }
    }
  });
  ...
```

The JVM shutdown hook mechanism requires a `Thread` object to perform necessary actions during process exit; we supply a thread at (#1) to cleanly stop the framework. When our shutdown thread executes, we verify that a framework instance was created at (#2) and, if so, we stop it at (#3). Since shutting down the framework happens asynchronously, the call to `fwk.stop()` will return immediately. We call fwk.waitForStop() at (#4) to make our thread wait for the framework to completely stop. It is necessary to have our thread wait, otherwise there is a race condition between the JVM process exiting and our framework actually stopping.

Using a shutdown hook is not strictly necessary. The process is in an awkward state during shutdown and not all JVM services are guaranteed to be available. There is also the potential for deadlock and hanging the process. In short, it is a good idea to try to cleanly shut down the framework, but be aware of the potential pitfalls and do as little work as possible in the shutdown hook.

11.2.3 Configuring, creating, and starting the framework

So, we know which bundles we want to install and registered a shutdown hook to cleanly stop the framework; all we need now is a framework instance. The following snippet shows how we create it:

```
...
Bundle mainBundle = null;                                          #1
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
try {
  List bundleList = new ArrayList();                             #2
  Map m = new HashMap();                                          #3
  m.putAll(System.getProperties());                               #4
  m.put(Constants.FRAMEWORK_STORAGE_CLEAN, "onFirstInit");        #5
  fwk = getFrameworkFactory().newFramework(m);                    #6
  fwk.start();                                                    #7
  ...
```

First we have some odds and ends. At (#1) we create a variable to hold a reference to our “main” bundle, which is a bundle with a `Main-Class` entry in its manifest file; we'll come back to this concept in a couple sections so we won't go into any details now. After that we create a list to hold the bundles we successfully install at (#2).

We get to the actual setup for the framework instance at (#3), where we create a configuration map for it. For our generic launcher, we copy the system properties in the configuration map as a convenience at (#4) and only set one configuration property at (#5), which is to clean the bundle cache on first initialization. In most cases, you likely won't want to do this, but for the purposes of the book examples this makes sense to make sure we always start up with a clean framework instance. We get the framework factory service and use it to create a framework instance using the configuration map at (#6). To get the framework factory service, we use the `getFrameworkFactory()` method we introduced in Listing 11.2. Lastly, we start the framework at (#7).

11.2.4 Installing the bundles

Now we have a configured and started framework instance. Since we configured the framework to clean its bundle cache on first initialization, we know our framework has no bundles installed in it. We need to remedy that. The following code snippet shows how we install the bundles contained in the directory specified on the command line:

```
...
BundleContext ctxt = fwk.getBundleContext();                     #1
for (int i = 0; i < jars.size(); i++) {
  Bundle b = ctxt.installBundle(                                 #2
    ((File) jars.get(i)).toURI().toString());
  bundleList.add(b);                                             #3
  if (b.getHeaders().get("Main-Class") != null) {                #4
    mainBundle = b;                                              #4
  }
}
...
```

At (#1), we get the `BundleContext` object associated with the system bundle; this is possible because the `Framework` object extends `Bundle` and represents the system bundle. We loop through the JAR files contained in the specified directory and install them at (#2) using the system bundle context; any exceptions will cause the launcher to fail. After we install a bundle, we add it to a list of installed bundles at (#3) and probe to see if its manifest contains a `Main-Class` header at (#4), which we'll use later. If there is more than one bundle with a `Main-Class` header, we use the last one we discover.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

11.2.5 Starting the bundles

We installed all of the bundles, but they aren't doing anything yet. We need to start them all. We can accomplish this in a simple loop over all installed bundles, invoking `start()` on each one, like this:

```
...
for (int i = 0; i < bundleList.size(); i++) {
  if (!isFragment((Bundle) bundleList.get(i))) {                #1
    ((Bundle) bundleList.get(i)).start();
  }
}
...
```

You may be wondering why we just didn't start each installed bundle right after starting it? It is better to install and start bundles in two passes: one pass for installing and one pass for starting. This helps alleviate ordering issues when it comes to dependency resolution. If you install and start a bundle immediately, it may fail to resolve since it may depend on some bundle that it not yet installed. By installing all of the bundles first, we stand a better chance of successfully resolving the bundles when we activate them.

Notice also at (#1), we don't simply call start on all bundles; instead, we only call start on bundles that are not fragment bundles. Fragments cannot be started and will throw an exception if we try to do so, which is why we avoid it. How do we know a bundle is a fragment? This simple approach works:

```
private static boolean isFragment(Bundle bundle) {
  return bundle.getHeaders().get(Constants.FRAGMENT_HOST) != null;
}
```

We just check to see if the bundle's manifest headers contain the `Fragment-Host` header. If so, it must be a fragment and we don't want to start it.

11.2.6 Starting the main bundle

We've installed and started all the bundles contained in the specified directory. In most cases, this would be good enough. For the examples in this book, however, we need one more step. In chapter 2, we showed how we could use the module layer all by itself to modularize our paint program. In that example, none of our bundles contained a `BundleActivator`, since activators are part of the lifecycle layer. In such an scenario, we need some way to start our application, so we decided to use the standard Java `Main-Class` JAR file manifest header as a way to define a "main bundle" from which we will load the "main class" and execute it `static void main()` method.

NOT AN OSGI CONVENTION

The notion of a main bundle with a main class is not an OSGi convention or a standard. We just defined this approach for this book to show that it is possible to use the OSGi modularity layer to modularize OSGi-unaware applications.

Listing 11.5 shows how we load the main class and invoke its `main()` method.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Listing 11.5 Invoking the main class from the main bundle

```
...
if (mainBundle != null) {                                        #1
  final String className =
    (String) mainBundle.getHeaders().get("Main-Class");          #2
  if (mainClassName != null) {
    final Class mainClass = mainBundle.loadClass(className);     #3
    try {
      Method method = mainClass.getMethod(                       #4
        "main", new Class[] { String[].class });
      String[] mainArgs = new String[args.length-1];             #5
      System.arraycopy(args, 1, mainArgs, 0, mainArgs.length);  #5
      method.invoke(null, new Object[] { mainArgs });            #6
    } catch (Exception ex) {
      System.err.println("Error invoking main method: "
        + ex + " cause = " + ex.getCause());
    }
  } else {
    System.err.println("Main class not found: " + mainClassName);
  }
}
...
```

If we have a main bundle at (#1), then we need to invoke its main class' main() method; we won't necessarily have a main bundle if the bundles do have activators. First, we get the name of the class from the Main-Class manifest header at (#2). Using this name, we load the class from the main bundle at (#3). At (#4) we use reflection to get the `Method` object associated with the main class' `main()` method. We make an array to contain any additional command-line arguments passed into the launcher after the specified directory at (#5). Finally, we use reflection to invoke the `main()` method at (#6), passing in any command-line arguments.

11.2.7 Waiting for shutdown

At this point, our launcher should have our bundled application up and running. So what's left to do? Not much really, just sit around and wait for it to finish, like this:

```
          ...
          fwk.waitForStop(0);                                    #1
          System.exit(0);                                        #2
        } catch (Exception ex) {
          System.err.println("Error starting framework: " + ex);
          ex.printStackTrace();
          System.exit(0);
        }
      }
    }
  }
```

We call `Framework.waitForStop()` at (#1), which does not stop the framework, it just waits for it to stop somehow. Why do we do this at all? Why not just let the calling thread run off the end of our main method, similar to what we do with Swing applications?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Unlike Swing applications, which result in a non-daemon thread getting starting for Swing event delivery, we don't have any guarantee when starting an OSGi framework that any non-daemon threads will be created. For those not familiar with concept of daemon threads, it is just a fancy way of saying a background thread. For the Java VM, if only daemon threads are present, it will end the VM process. So we need to explicitly wait for the framework to stop because we know the main thread is non-daemon and will keep the VM process alive.

For similar issues, we call `System.exit()` at (#2) to end the VM process. If we didn't call `exit()` here and someone started a non-daemon thread that wasn't properly stopped, then the VM process would never exit. This is similar Swing applications, which require an explicit call to `exit()` since the Swing event thread is non-daemon.

That's all there is to it. We've successfully created a completely generic launcher that will work with any OSGi R4.2 framework implementation. To use this launcher with an arbitrary framework implementation, just put it on the class path with the launcher and you are good to go. But what about situations where you cannot convert your entire application into bundles? In that case, you may want to embed a framework instance inside your application. We'll look into that next.

11.3 Embedding the framework

In some situations it is not possible to convert your entire application into bundles where your whole application runs on top of the OSGi framework. This could happen in legacy situations where conversion into bundles is prohibitively expensive or even is situations where there is resistance or uncertainty about converting your entire application. Even in these sorts of situations, it is possible to leverage OSGi technology for specific needs. For example, it is not uncommon for Java-based applications to provide a "plugin" mechanism for extensibility purposes. If your application has a plugin mechanism or you are thinking about adding one, an embedded OSGi framework can do the trick.

You might be thinking, "Wouldn't I be better off just creating my own simple plugin mechanism in this case?" Typically, the answer is no. The dynamic class loading aspects of plugin mechanisms are difficult to get correct. Over time you will likely find you need to add more advanced features, like support for library sharing, side-by-side versions, or native libraries, at which point you start to get into really complicated territory and start reinventing the wheel. By using OSGi, all of this is taken care of for you so you can concentrate on implementing your application's core functionality. If you are concerned about the size of an OSGi framework, remember that they were intended to run in embedded devices, so most implementations are not too hefty. In addition, you get the benefit of having a known standard which makes it easier for your plugin developers and provides opportunity for reuse of existing bundles.

Embedding an OSGi framework instance into an application may sound pretty exotic, but thanks to the standard framework launching and embedding API, it is largely the same as simply launching the framework. There are, however, some differences and a few issues you

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

need to understand. In the remainder of this section we will discuss these issues as well as present an example of embedding and framework instance into an application.

11.3.1 Inside versus outside

The main issue around embedding a framework instance into an application is the distinction between being on the "inside" of the framework versus being on the "outside" of the framework. The bundles deployed into the embedded framework live in a nice insulated world and know nothing about the outside. Conversely, the application lives in an external rough and tumble world. Figure 11.1 illustrates the situation.

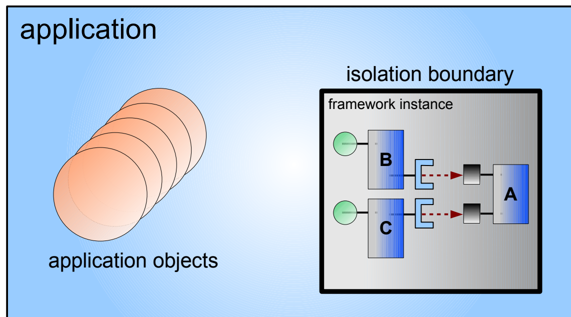


Figure 11.1 The embedded framework instance forms an isolation boundary between the bundles on the inside and the application objects on the outside.

It is possible to traverse the isolation boundary provided by the framework, but the inside/outside distinction places some constraints on how the application can interact with installed bundles and vice versa.

AVOID BEING ON THE OUTSIDE

The best approach for dealing with the inside/outside divide is to eliminate it by converting your entire application to bundles. If you are on the fence about this issue, it is possible to start with an embedding approach and later convert the rest of your application to bundles. But if you have a choice up front, start with all bundles.

So, if you decide to embed a framework instance, what are some of the things you will likely need to do with it? You will likely want to:

1. Interact with and manage the embedded framework instance.
2. Provide services to bundles and use services from bundles.

Let's look at what we need to do in each of these cases.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Interacting with the embedded framework

We actually already know how to interact with embedded framework instances; through the standard launching and embedding API. When we create an instance of an R4.2 compatible framework implementation, we get an object which implements the `Framework` interface. As we saw previously, this interface gives us access to all of the API necessary to control and inspect the framework instance. The framework instance represents the system bundle and provides us with a passage from to the inside of the framework, as depicted in Figure 11.2.

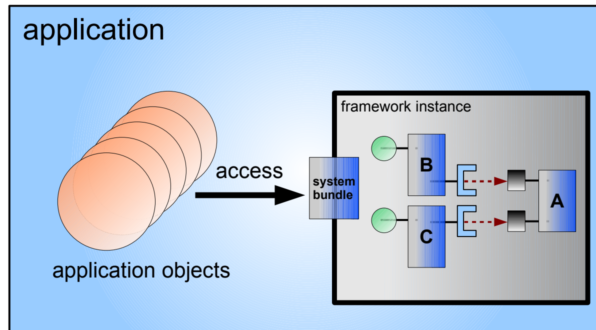


Figure 11.2 A framework instance represents the system bundle and provides the means to manage the framework instance as well as interact with deployed bundles.

From the system bundle we can start and stop the framework as well as deploy, manage, and interact with bundles. If you are using an embedded framework instance as a plugin mechanism in your application, you will use this API to deploy plugin bundles by loading them from a directory or providing a GUI for user access, for example. It is also through this API that we can provide services to bundles and use services from bundles.

Providing services and using bundle services

Luckily, there is no new API to learn when it comes to providing application services to embedded bundles or using services from them. We learned about providing and using services in chapter 4 and all of that knowledge applies here. The only real difference is we are going to use the system bundle to do everything, since the application has no bundle associated with it.

Since we need a `BundleContext` to register or find services, we simply use the `BundleContext` associated with the system bundle. We can get access to it by calling `getBundleContext()` on the framework instance. From there, registering and using services is pretty much the same as if the application were a bundle. Simple, right? As you might expect, there is one main constraint.

COMMON CLASSES REQUIRED

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

> An application embedding a framework instance can only interact with contained bundles using objects whose class definition is the same for both the application and bundles.

By default, the application on the outside and the bundles on the inside only share core JVM packages, so it would be possible for the application and bundles to interact using objects from classes defined in core JVM packages. For example, the it would be possible to provide or use `java.lang.Runnable` services, since we know the application and the bundles use a common class definition for `Runnable`. This works out fairly well if everything you need is in a core JVM package, but this won't typically be the case.

Luckily, there is a rudimentary way to share packages from the application to the contained bundles via framework configuration. The launching and embedding API defines two previously mentioned configuration properties for this purpose:

- `org.osgi.framework.system.packages`
- `org.osgi.framework.system.packages.extra`

The former defines the complete set of class path packages exported by the system bundle, while the latter defines an additional set of class path packages which is appended to the former set. Typically, you will only want to use the latter property, since the specification requires the framework to set a reasonable default for the former. For an example, consider if we were going to created a version of our paint program that used an embedded framework instance. In that case, we would likely want to put the `SimpleShape` interface on the class path so we could share a common definition between the application and the bundles. In that case, we would configure our framework instance like this:

```
Map m = new HashMap();
m.put(Constants.FRAMEWORK_SYSTEMPACKAGES_EXTRA, "org.foo.shape");
fwk = getFrameworkFactory().newFramework(m);
fwk.start();
```

The syntax to use when specifying the property is exactly the same as for the `Export-Package` manifest header, which means we can specify additional packages by separating them with commas and we can also include version information and attributes.

> **NECESSARY, BUT NOT SUFFICIENT**
>
> It is necessary to specify this configuration property to share class path packages with bundles, but it is not sufficient to do only this. You must also ensure that the specified packages are actually available on the class path when you start your application. This is accomplished in the standard way.

The need to perform this configuration is an extra step for the application, but from the bundle's perspective it is business as usual. The bundle's simply need to specify the package on their `Import-Package` manifest header, like normal, and the framework will give them access to the package following normal OSGi rules.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

So, what about the situation where you do not have a common class available from the class path? Since it is not possible for the application to import packages from bundles, there isn't much you can do here. The main option is to resort to reflection, which is possible since OSGi service lookup can be performed by the class name. Of course, you should use `BundleContext.getAllServiceReferences()` instead of `BundleContext.getServiceReferences()`, since the framework will potentially filter results if it determines you do not have access to the published service type. This will give you access to the ServiceReference which you can use to get access to the service object so you can invoke methods on it using reflection.

If you had a situation where you had different definitions of the service class on the outside and inside, you could even try to get fancy and use dynamic proxies to bridge the two types in a generic way. But this is likely overkill.

11.3.2 Who's in control?

If you're going to pursue the embedded framework route, you may run into a few other issues related to who is expecting to be control. Generally speaking, the OSGi framework assumes it is in control of the JVM on which it is running. If you are embedding a framework, it is likely you don't want it to be in control or at least want it to share control with the application in which you are embedding it. It is not uncommon to run into issues related to JVM singleton mechanisms, like URL and content handler factories or security.

Singleton mechanisms like these are only intended to be set once at execution time. OSGi framework implementations need to be responsible for initializing these mechanisms to properly implement specification functionality. When a framework is embedded in another application, often the application assumes it is in control of these singletons. The OSGi specification doesn't specifically address these aspects of framework embedding, so how implementations deal with it is undefined. Some frameworks, like Apache Felix, go to lengths to try to do the "right thing," but the right thing often depends on the specific use case. If you run into issues in these areas, you'll have to consult the documentation or support forums for your specific framework implementation.

Another area where issues arise is in the use of the thread context class loader. If you are not familiar with this concept, each thread in Java has a class loader associated with it, which is the thread context class loader. The context class loader provides a backdoor mechanism to subvert the normal, strict hierarchical class loading of Java. Application servers and various frameworks use this mechanism to deal with class loading dependencies that cannot be shoehorned into hierarchical class loading. Unfortunately, this half-baked attempt at dealing with class loading dependencies doesn't mesh well with OSGi modularity.

Explain example and potential workarounds...

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

11.3.3 Embedded framework example

For a simple illustration of framework embedding, we'll take the service-based paint program from chapter 4 and convert it into a stand-alone application with an embedded framework instance. Since the service-based paint program was completely composed of bundles, we need to transform it into a Java application. Our new stand-alone paint program will use an embedded framework instance as a plugin mechanism by which it can deploy custom shape implementations. Figure 11.3 shows the before and after state.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

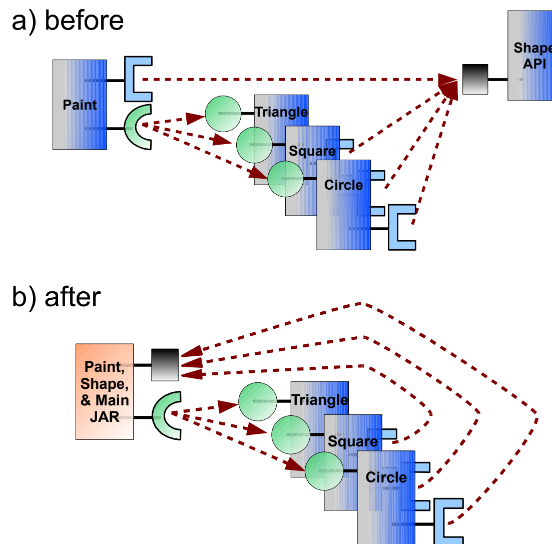


Figure 11.3 a) The service-based paint program is composed of five bundle sharing packages and services. b) The stand-alone paint program combines the core paint program, shape API, and launcher into a single JAR file which will provide share the API with the bundles and use their services.

For the stand-alone paint program, the shape bundles don't need to be changed at all. So what does need to be changed? The original service-based paint program didn't need a launcher, since the bundle activator in the paint bundle served this purpose. For the stand-alone paint program, we need a launcher that will create the paint frame and the framework instance and wire everything together. Additionally, since the paint program needs a common class definition to interact with bundles implementing shapes, we must move the shape API into the stand-alone application so the application and bundles can use the same `SimpleShape` service interface definition. Note that Figure 11.3 depicts the application as a quasi bundle with an exported package and service dependencies. This is just for illustrative purposes, the application is just a normal JAR file. The structure of the modified paint program source code is as follows:

```
org/foo/paint/
  DefaultShape.java
  Main.java
  PaintFrame.java
  ShapeTracker.java
  ShapeComponent.java
  underc.png
org/foo/shape
  SimpleShape.java
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

What's the design of our stand-alone paint program? Recall the original design of the paint program, the main paint frame was designed in such a way to be injected with shape implementations. This approach had the benefit of allowing us to limit dependencies on OSGi API and to concentrate our OSGi-aware code to the shape tracker. In keeping with these design principles, we will do most of the work in the launcher `Main` class, which will create the embedded framework instance, deploy the shape bundles, create the paint frame, and bind the paint frame to the embedded shape services.

It may be fairly obvious to you at this point that this already sounds pretty similar to the generic framework launcher we created in the previous section. You'd be correct. Using the framework in an embedded way is not all that different, other than the issues we outlined previously. As a result, the launcher code for our stand-alone paint program will bear a striking resemblance to our generic launcher. The different aspects it illustrates are:

1. Sharing code from the class path to bundles.
2. Using services on the outside.
3. Providing services to the inside.

This last aspect doesn't have an analogue in the original service-based paint program, but we include it to demonstrate that it is possible to provide services from the outside. As before, we'll break the launcher into small snippets and describe each one. Let's get started.

PERFORMING THE MAIN TASKS

Since our paint program is no longer a bundle, we replace its bundle activator with a `Main` class. The main tasks this class performs are easy to discern from the `main()` method in Listing 11.6.

Listing 11.6 Stand-alone paint program main method

```
public class Main {
  private static Framework fwk;
  private static PaintFrame frame = null;
  private static ShapeTracker shapeTracker = null;

  public static void main(String[] args) throws Exception {
    addShutdownHook();                                              #1
    fwk = createFramework();                                        #2
    publishTrapezoidService();                                      #3
    createPaintFrame();                                             #4
  }
  ...
```

The performed functionality is a combination of our generic launcher and the old bundle activator: adding a shutdown hook (#1), creating a framework instance (#2), creating a paint frame (#4). The only new task is publishing an external trapezoid shape service (#3), which we'll see is pretty much the same as publishing a normal service. Let's continue to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

look into the details. Since adding a shutdown hook is basically identical to what we did in the generic launcher, we'll skip that step and go directly to creating the framework instance.

CONFIGURING AND CREATING THE FRAMEWORK

The `createFramework()` method follows fairly closely to our launcher, so we can go over the details fairly quickly. The method starts, like the launcher, with discovering which bundles it should install into the framework instance as depicted in the following snippet:

```
...
private static Framework createFramework() throws Exception {
  File[] files = new File("bundles").listFiles();                    #1
  Arrays.sort(files);
  List jars = new ArrayList();
  for (int i = 0; i < files.length; i++)
    if (files[i].getName().endsWith(".jar"))                         #2
      jars.add(files[i]);                                            #2
...
```

Here we get the contents of the bundles directory in the current directory at (#1) and add all contained JAR files to a list at (#2). This is rather simplistic, but it is sufficient for this example. Now we can create the framework instance and deploy the discovered bundles. The snippet in Listing 11.7 shows these steps.

Listing 11.7 Creating the framework instance and deploying discovered bundles

```
  ...
  try {
    List bundleList = new ArrayList();
    Map m = new HashMap();
    m.putAll(System.getProperties());
    m.put(Constants.FRAMEWORK_STORAGE_CLEAN,                          #1
      Constants.FRAMEWORK_STORAGE_CLEAN_ONFIRSTINIT);                 #1
    m.put(Constants.FRAMEWORK_SYSTEMPACKAGES_EXTRA,                   #2
      "org.foo.shape; version=\"4.0.0\"");                            #2
    fwk = getFrameworkFactory().newFramework(m);                      #3
    fwk.start();                                                      #3
    BundleContext ctxt = fwk.getBundleContext();                      #4
    for (int i = 0; i < jars.size(); i++) {                           #5
      Bundle b = ctxt.installBundle(                                  #5
        ((File) jars.get(i)).toURI().toString());                     #5
      bundleList.add(b);                                              #5
    }                                                                 #5
    for (int i = 0; i < bundleList.size(); i++) {                     #6
      ((Bundle) bundleList.get(i)).start();                           #6
    }                                                                 #6
  } catch (Exception ex) {
    System.err.println("Error starting framework: " + ex);
    ex.printStackTrace();
    System.exit(0);
  }

  return fwk;
}
...
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

As with the generic launcher, we configure the framework to clean its bundle cache on first initialization at (#1). For performance reasons, you'd likely not want to do this if you were using the framework as a plugin mechanism, since it is slower to re-populate the cache. We do it in the example to make sure we are starting from a clean slate. An important difference from the launcher, which we alluded to previously, is at (#2). Here we configure the framework to export the `org.foo.shape` package from the class path via the system bundle. This will allow bundles to import the package from the application, thus ensuring they are both using the same interface definition for shape implementations. We also need to ensure this package will be on the class path, but since we are going to package it in the application JAR file it will definitely be available.

We create the framework with the defined configuration and start it at (#3). We get the system bundle's bundle context at (#4), which we use to install the discovered bundles at (#5). Lastly, we start all installed bundles at (#6). Any errors will cause the JVM to exit. Now let's look at how we publish an external service into the framework instance.

Publishing an external service

The `publishTrapezoidService()` method is quite simple as the following code snippet illustrates:

```
...
private static void publishTrapezoidService() {
  Hashtable dict = new Hashtable();
  dict.put(SimpleShape.NAME_PROPERTY, "Trapezoid");
  dict.put(SimpleShape.ICON_PROPERTY,
    new ImageIcon(Trapezoid.class.getResource("trapezoid.png")));
  fwk.getBundleContext().registerService(                         #1
    SimpleShape.class.getName(), new Trapezoid(), dict);          #1
}
...
```

This code is basically the same as we saw back in chapter 4 for publishing services. The only difference is we use the system bundle's bundle context to register the service at (#1), since the application doesn't have its own bundle context. Of course, what makes this possible is the fact that we are using the same `org.foo.shape` package on the inside and the outside, which means our trapezoid shape will work just like the shapes provided by any of the shape bundles. Now we are ready to bind everything together to complete our functioning paint program.

Creating the paint frame

The `createPaintFrame()` method performs nearly the same functionality as the bundle activator for the original paint bundle from chapter 4. The details are shown in Listing 11.8.

Listing 11.8 Creating the paint frame and binding it to the framework instance

```
...
private static void createPaintFrame() throws Exception {
  SwingUtilities.invokeAndWait(new Runnable() {
    public void run() {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
        frame = new PaintFrame();                                          #1
        frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        frame.addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent evt) {                     #2
            try {                                                          #2
              fwk.stop();                                                  #2
              fwk.waitForStop(0);                                          #2
            } catch (Exception ex) {                                       #2
              System.err.println("Issue stopping framework: " + ex);       #2
            }                                                              #2
            System.exit(0);                                                #2
          }                                                                #2
        });
        frame.setVisible(true);                                            #3

        shapeTracker = new ShapeTracker(fwk.getBundleContext(), frame);    #4
        shapeTracker.open();                                               #4
      }
    });
  }
```

We create the paint frame itself at (#1) and then add a window listener to cleanly stop the embedded framework instance and exit the JVM process when the frame is closed at (#2). We display the paint frame at (#3), but at this point it is not actually hooked into the embedded framework instance. At (#4), we get the system bundle's bundle context and use it to create a shape tracker for the paint frame. This is what actually binds everything together. Due to our original design, we don't need to spread OSGi API usage throughout our application.

To run the stand-alone paint program, go into the `code/chapter11/paint-example/` directory and type `ant` to build it and `java -jar paint.jar` to run it. Figure 11.4 shows the result.

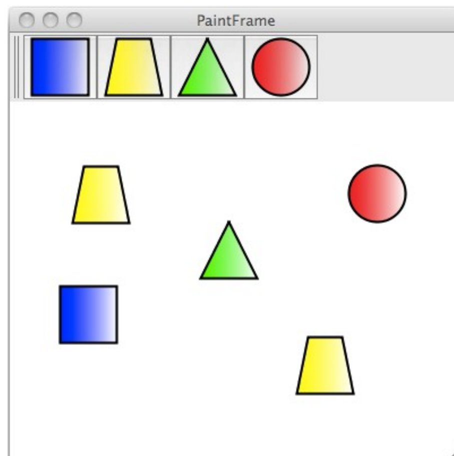


Figure 11.4 The stand-alone paint program

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

11.4 Summary

The OSGi specification do not define a standard way to configure and launch a framework. Consequently, most OSGi framework implementations provide their own approach. The standard launching and embedding API introduced in R4.2 is the next best thing to a standard launcher, since it allows us to create a single launcher that works across framework implementations. In this chapter we learned:

- The OSGi R4.2 specification introduced the `Framework` interface to represent a framework instance.
- The `Framework` interface extends the existing `Bundle` interface to extend our existing knowledge of managing bundles to framework instances.
- The Framework instances actually represents the system bundle, which provides access to the system bundle's bundle context for performing any task that a normal bundle can (e.g., installing bundles and registering services).
- The `META-INF/services` approach is used to find a `FrameworkFactory` provider, which enables framework creation without knowing a concrete framework implementation class name.
- The OSGi specification defines numerous framework configuration properties to further improve framework implementation independence.
- Although completely bundled applications is the preferred approach, the launching and embedding API also simplifies embedding framework instances into existing applications.
- When embedding a framework instance into an application, the main constraint involves dealing with the difference between being on the outside versus the inside. If direct interaction with bundles is required, then sharing common class definitions from the class path is often required.
- Other than some additional constraints, embedding a framework instance is nearly identical to launching a framework instance.

With this knowledge under out belt, we can now customize our framework usage for our own specific purposes and do so in a framework implementation neutral way. In the next chapter, we'll look into configuring our framework instances to deal with security.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

12
Security

Building your applications on top of OSGi allows you to create loosely coupled and extensible architectures. Bundles can come and go at any time and it is easy to provide third parties with the possibility to extend your application in a well-defined way. However, as with most things in live, there is a downside to this approach as you open yourself (or your users) up to security vulnerabilities in cases where the third party bundles can not trusted completely.

Luckily, the Java platform has security build in and OSGi can be used with security enabled. Unfortunately, secure sandboxes and their restrictions are difficult to get right and often hard to deal with. This is especially true in an environment as dynamic as OSGi. Fortunately, OSGi has an extensive and very powerful security model that eases this difficult task by providing a well-defined API to manage permissions where other execution models tend to leave the permission management up to implementations.

In this chapter we will make you familiar with the Java security model and how it is used by OSGi to provide the infrastructure to deploy and manage applications that must run in secure environments. You will learn how to secure your applications on the one hand and how to create bundles that are well behaved and easy to use in a security enabled OSGi framework on the other hand. That said, lets have a closer look at why you might want to care about security.

12.1 To Secure or Not Secure

Modern applications and software solutions increasingly center around loosely coupled and extensible architectures. Component or Service orientation is applied in almost all areas of

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

application development including distributed systems, ubiquitous computing, embedded systems, and client-side applications.

One of the main drawbacks of dynamically extensible applications are the potential security issues that arise due to executing untrusted code without appropriated measures in place. Just think about it. How many times did you execute code that you couldn't really trust regardless and why?

Typically, one of the reasons is that permission management is a pain and it really doesn't help that it is hard to impossible for a normal user to asses what the impact of allowing something is. Plus, the user wants to run the application (otherwise, why bother at all). So to the user, security and permissions are in the way and not really helpful.

The other problematic point about security is that it is inherently tricky to establish a meaningful identity. Somehow it must be possible to differentiate between different providers or provider types in order to permit or deny permissions. Often the location of a software artifact is used but to ensure that the artifact has not been tempered with signing must come into play which leaves us with a complicated process to create and maintain certificates and trust between certificates which ultimately is nothing a user or developer wants to deal with either.

Speaking of developers, maybe the biggest problem with security is that it adds another burden to development. More often then not, code has to be aware of security in order to be usable when security is enabled. Furthermore, fine-grained security checks make programs execute slower which often is a killer argument against it.

In summary, we can easily identify three points needed for meaningful security management and who is responsible for providing them namely:

- Identity – Defining the identity is clearly up to the provider of the code. In the simple case, we can use the location of the codebase as an identity (as in e.g., everything from the local disk or a certain domain is trusted) but if we want more then that, we have to resort to cryptographic measures by means of certificates. In any case, the provider needs to make the code available in a way that we can establish the needed credentials (be it a certain location or a certificate based signature).
- Permission Management – Once we have the identity we still need to define the permissions the code should have. This is up to whoever is responsible for the framework. This might be the gateway operator or the server admin but it is entirely possible the we are talking about an end-user. As a consequence, we need to make it as simple as possible to manage permissions.
- Permission Checks and Privilege Management – Last but not least, security has to be built into the code itself. Therefore, the developer has to think about needed security checks on the one hand and on where he want to call code outside of his control with limited permissions.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

So is it worth it? Well, it clearly depends on the given situation. In many cases it is just not inside the scope of an application to consider security. Either the speed impact would be to big or the development costs to high. Typically, this already serves as a filter for possible security enabled applications. If the costs are to high then it probably is not worth it as the benefit of having security would outweigh these arguments otherwise. Keep in mind so that if you don't design your applications and your code in a way that it is usable in security enabled environments then it is unlikely that it can function or be used in the very same. Which gets us in a kind of catch-22 in regard to security. Because it is often to painful to consider it applications and libraries are built without security in mind. As a consequence, developing for this applications or reusing the libraries in a secure context becomes next to impossible.

Ideally we would make sure that all our code has security checks in the right places and is signed by a valid certificate that is part of a chain of trust to some well known root certificate. The reminder of this chapter will show you how and what you can do and use to take advantage of the security capabilities of the OSGi framework. Lets have a look at what this looks like form a high-level perspective in the next section when we talk about Java and OSGi security.

12.2 Security – Just do it

So we do want to secure our OSGi based application. Great, but where to start? Well, let's start at the beginning and have a look at the Java security architecture and its permission model which the OSGi security model is based on.

12.2.1 Java and OSGi Security

Discussing the complete Java security architecture at this point is clearly outside the scope of this book but we want to at least mention and introduce the parts that are important to understand the remainder of this chapter. The most important aspect in this context is to understand that the OSGi related part of Java security at its core uses the codebased security features only. In other words, for the decision of whether a certain action is allowed userbased security is possible but must be added by an additional layer on top. In the Java context this is the JAAS framework and in the OSGi world we have the UserAdmin service which serves a similar purpose. In regard to codebased security it basically comes down to how to check for and assign Permissions, how to establish the (protection) domain of a bundle, and how to determine the identity (i.e., the location or signature of a bundle). If that doesn't tell you anything at the moment, don't worry. We will define and introduce the terms and concepts as we go along. But now back to the basics first.

The Java permission model is actually pretty simple. The main idea is that there is the java.security.Permission class which has a special method called implies() that accepts another Permission. With this mechanisms we are able to define our own permissions as subclasses of java.security.Permission that can be used both, to check and to assign

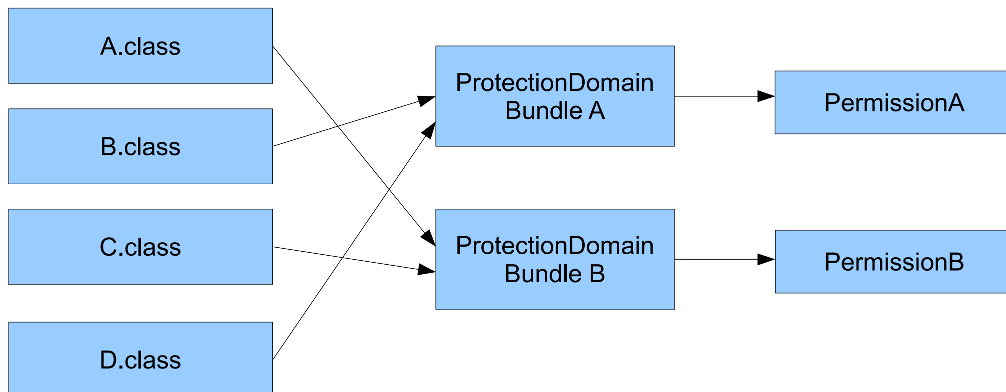
©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

permissions. How's that? Well, just think about it: all we have to do is to implement the implies() method to check whether the target (the given permission) has all the properties our permission implies. If that is the case then the check is successful otherwise not. At the same time, it follows that in order to assign our permission, all one has to do is to create an instance of it.

This leads to the question whom can we assign a permission to? For that, a special concept called a ProtectionDomain is defined as encapsulating the characteristics of a domain. Sounds complicated but in reality it is just a little abstract. For example, in OSGi such a domain is a bundle. Subsequently, all classes that are originating in the same bundle are member of the same ProtectionDomain (in this case called the BundleProtectionDomain). You probably can see already where this is going namely, a BundleProtectionDomain encloses exactly the set of classes whose instances are granted the set of permissions granted to the bundle. So there we have our answer (at least in the OSGi case). We can assign permissions to a bundle. This can happen either explicitly or indirectly as we will see later.

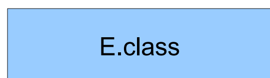
Now, the important thing is that a permission check consults all protection domains on the stack. That means that when a permission check is triggered by either invoking any of the SecurityManager.check* methods or the AccessController.checkPermission method the JVM will collect the ProtectionDomains of all classes that have been involved in reaching the check and make sure that each of the permission sets associated with them have at least one permission implying the requested permission. Have a look at figure xxx to see what the process looks like.

AccessController.checkPermission(Permission p);



Privileged Call

```
if (!(PermissionA.implies(p) &&
      PermissionB.implies(p))
{
      throw new SecurityException();
}
```



Ch2Figure 12.1: Security Check

In this case, the AccessController.checkPermission method is used to see whether at the given point all callers have a given permission p. The JVM then performs the stack walk to determine that we have instances of the classes A,B,C, and D involved. Subsequently, it determines that the A,D pair and the B,C pair originate from BundleA and BundleB, respectively. Each of which have a permission granted. Logically, it then has to check whether both assigned permission imply the given permission p. Otherwise a security exception is raised.

Wait a moment, what about the E class? Well, for that we need to introduce you to the last piece of the puzzle namely, privileged calls. We already discussed in the last section that it must be possible, somehow, to limit the ProtectionDomains on the stack to the one that makes the security sensitive call. Why's that again? Well, this gives us the ability to allow other code that calls us with less privileges then us to still have us do things that require more privileges. We can do this in this case by using the AccessController.doPrivileged() method. The way this works then, is that only the ProtectionDomains on the stack up to the last privileged call are considered. As a simple example consider the following scenario.

Let's assume we have a service that gives access to a specific resource – say a file in the filesystem which contains sensitive informations. While we are trusting the bundle containing the class implementing the service we don't want to give access to that file to anybody else directly. The idea is that other parties must ask our service to retrieve the desired information for them. How would we do that? Well, probably the easiest way is to rely on the build in security checks for files and give only our bundle the FilePermission to access the file in question. But if you payed attention than this doesn't help us much because when we access the class on behalf on another bundle (i.e., because one of our methods has been called) than the ProtectionDomain of that bundle will be on the stack as well. So we need to use a doPrivileged() around our access to the file like this,

```
AccessController.doPrivileged(new PrivilegedAction(){
  public Object run() {
    // do access with only us on the call-stack
  }
});
```

It follows, that we really need to prepare our code with security in mind otherwise we are running into problems. In this case, lets assume we did that and the service guards all its accesses to the file with a doPrivileged() then there is another typical problem that you might not want to give all other bundles access to these methods. As you will see later, its easy to prevent access to the service entirely but if a bundle has permission to access the service then the service needs to take care about the more fine-grained security checks itself. Luckily, we can just use the normal security mechanisms and define our permission and check whether we want to grant the caller the requested action or not.

```
SecurityManager sm = System.getSecurityManager();
if (sm != null) {
  sm.checkPermission(new AccessPermission());
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

In this example we just get the SecurityManager and ask it to check for a custom AccessPermission. Now all bundles (i.e., ProtectionDomains) on the stack up to the last doPriviledged() call (if any) must have AccessPermission granted. Thats pretty much all there is to it but the challenge is in thinking about and designing your code in such a way that scenarios like the just presented are taken into account.

SECURITYMANAGER V.S. ACCESSCONTROLLER

One more thing, the reason we are using the SecurityManager in the way we do is that this way we will only trigger a security check in case security is actually enabled (i.e., a SecurityManager is installed). Which can sometimes prevent a performance penalty and furthermore, OSGi encourages the use of the SecurityManager over the AccessController as some of the security performance optimization of OSGi will only be possible if the former is used.

In summary, each bundle has its own ProtectionDomain and when the AccessController or the SecurityManager is used to check whether a certain permission is granted it considers all ProtectionDomains on the call stack up to the latest privileged call. This way, we have pretty fine-grained control over what is and isn't allowed inside the framework by managing the permissions associated with each bundle. At the same time a bundle can still make security sensitive calls on behave of other bundles assuming it has the needed permissions. That, together with the fact that we can add our own custom permissions into the mix gives us a lot of possibilities to secure our applications. The downside is that managing all that is getting pretty complex real fast as you might imagine. Luckily, the OSGi specification is aiding us in this difficult task while considering security throughout all aspects. Lets learn about the OSGi specific permissions in the next section.

12.3 OSGi Permissions Revealed

In order to provide a first layer of security and make it possible to limit what bundles can do inside the framework, the OSGi specification defines a couple of permissions that bundles need in certain situations. Basically, we are talking about special permissions for framework and service related tasks that, just like with most other things, can be ordered by the three core layers as they are needed only for features in the layer they are used. We have:

- The module layer – which is where the PackagePermission and the BundlePermission is needed for bundles to be allowed to import or export packages and require bundles, respectively.
- The lifecycle layer – where most of the API requires some form of the AdminPermission to be allowed to manage the lifecycle of bundles.
- The service layer – requiring the ServicePermission for a bundle to be able to publish, find, or bind a service.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Again, you or other people's bundles can always use or define custom permissions as long as they are exported by a bundle or are on the classpath of the framework and of course the Java standard permissions still apply. However, we will briefly introduce these OSGi specific permissions in the following subsections in order to give you an idea what permissions look like and what you need to get your bundles up and running when security is enabled.

Typically, a permission accepts two parameters in its constructor. The Name, which is the target of the permission and the action. For example, a FilePermission expects the path of the file as its name and the READ, WRITE, etc. action. Together they allow to express what file the permission gives access to and what kind of access. Lets have a look at PackagePermission and how it can be used next.

Filter Based Permissions

Since the release 4.2 of the specification, OSGi supports a number of permissions that are granted when the target of the permission is related to a bundle. For example, AdminPermission can grant a bundle the permission to manage other bundles. This is expressed by using a filter expression for the name of the permission. For example, a bundle can get all services registered by bundles coming from a specific location: ServicePermission("(location=file:bundles/*)", GET);. The filter can contain the following keys:

- id – The bundle ID of the bundle.
- location – The location of the bundle. Filter wildcards for strings are supported.
- signer – A distinguished name chain as described later in this chapter.
- name – The symbolic name of a bundle. Filter wildcards for strings are supported.

12.3.1 PackagePermission

On the module layer we need a way to limit the packages a bundle can import or export. Typically, you want to make sure that a given bundle only gets access to the right providers of packages as they might contain security sensitive code. What does that mean? Well, remember when we've been talking about the possibility to run code with just your protection domain on the stack – thats one of the cases where you want to make sure that only the right kind of bundle has access to the package that contains this kind of code. At the same time, the export case is just as important. If a bundle is allowed to export a package then it might be able to have that code used by another bundle with more privileges and influence the other bundles logic in a way that it does something it shouldn't do (although in this case the protection domain of the exporting bundle would still be considered).

In summary, the PackagePermission is a bundle's authority to import or export a package. It accepts the name of the package as the Name of the permission and has two

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

actions namely, EXPORT and IMPORT. Regaring the name, for convenience it is possible to use the * wildcard to be able to target several packages with a single permission. In regard to the actions, EXPORT does imply IMPORT. In other words, if you give EXPORT permission for a package then the bundle has automatically IMPORT permissions for the package as well. The reason for the latter is that for suggestibility reasons your bundles should import what they export in most cases as we already discussed in the modularity chapters.

So what would a PackagePermission look like and how would it be used? Lets have a look at a simple conceptual example. Lets assume we have a bundle with the following imports and exports.

```
Import-Package: org.foo, org.bar
Export-Package: org.bar
```

When the bundle gets resolved the framework sill have to do the following security checks for the bundle.

```
System.getSecurityManager().checkPermission(new
        PackagePermission("org.foo", PackagePermission.IMPORT));
System.getSecurityManager().checkPermission(new
        PackagePermission("org.bar", PackagePermission.IMPORT));
System.getSecurityManager().checkPermission(new
        PackagePermission("org.bar", PackagePermission.EXPORT));
```

Notice, we don't need to check for the permission to import org.bar as the export permission already implies the import but that is already handled by the permission itself (in its implies method) so in order to keep the logic simple we just check for every import.

So as an example of what a PackagePermission looks like, lets have a look at the following two permissions that are one way to make the bundle work.

```
new PackagePermission("org.*", PackagePermission.IMPORT);
new PackagePermission("org.*", PackagePermission.EXPORT);
```

As you can see, we are using the wildcard capability to just give PackagePermission to all packages below org. In practice, you probably don't want to do that as it would be to broad a scope but at least it shows how to use wildcards. Furthermore, you might have noticed that we actually wouldn't need the first permission at all as the second already implies the first. Alright, lets have a look at the second module layer related permission next.

12.3.2 BundlePermission

Similar to the PackagePermission, the BundlePermission is based in the module layer and represents a bundle's authority to require/provide/attach a bundle/fragment. So while the PackagePermission is required to import or export a given package, the BundlePermission is need to require another bundle or to attach a fragment to a host. The name of the permission is the name of the bundle that is required or in the case of a fragment attached to. In regard to the actions, we have the PROVIDE and REQUIRE action which pretty much speak for themselves in this case. Furthermore, like the EXPORT action of the PackagePermission does imply the IMPORT of the package as well, the PROVIDE action implies the REQUIRE. We will not give an example of this permission at this point as it really

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

is similar to the PackagePermission. Instead, lets get on to the lifecycle layer and its associated permission namely, the AdminPermission.

12.3.3 AdminPermission

As you probably can imagine, the lifecycle layer needs to be the center of your attention when you are about to secure your framework. This is where you can decide what bundles can be installed and from whom. Furthermore, some of the methods provided are of a sensitive nature as well, like the metadata of a bundle that can contain information some other bundles should not see, etc.

For this reason the specification gives you a rather complex and powerful permission namely, the AdminPermission which represents a bundle's authority to perform specific privileged administrative operations or get sensitive informations about a bundle. The permission is centered around a name (that identifies the bundles it should match) and a set of ten actions (that the caller is allowed to perform on bundles that match the name filter).

Again, the name of the permission is a filter expression that can use any of the following parameters:

- signer – A DN chain of bundle signers
- location – The location of the bundles.
- id – The bundle ID of the bundle.
- name – The symbolic name of the bundle.

We will give examples of possible filters in a moment but first lets introduce the actions as well. As we already said, there are ten Actions namely,

- class – load a class from a bundle
- execute – start/stop bundle and set bundle startlevel
- extensionLifecycle – manage extension bundles
- lifecycle - manage bundles (upate/uninstall/etc.)
- listener – add/remove synchronous bundle listener
- metadata – get manifest and location
- resolve – refresh and resolve a bundle
- resource – get/find resources from a bundle
- startlevel – set startlevel and initial bundle startlevel
- context – get bundle context

The special action * represents all actions. As you probably can see, together they cover the security sensitive portion of the lifecycle api and allow you to limit access to lifecycle operations on a finegrained basis. We will not give examples for each possible action at this point but focus on a simple example to show you the essentials. Ready? Ok, lets assume we want to install and start a bundle programatically and have a look at the following code which does just that.

```
context.installBundle("file:bundle.jar").start();
```

Now, the corresponding security check that the framework has to do looks like the following,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
System.getSecurityManager().checkPermission(new AdminPermission(bundle,
AdminPermission.LIFECYCLE));
System.getSecurityManager().checkPermission(new AdminPermission(bundle,
AdminPermission.EXECUTE));
```

As you can see, the AdminPermission in this case provides a constructor which accepts the targeted bundle as well the action. Why's that? Well, think about it for a second. We are going to give AdminPermission to the bundles we trust to do lifecycle related operations but we might wont to be able to limit their capabilities to only manage a specific kind of bundles. Therefore, the AdminPermission must know about the target bundle. How can we specify the bundles we target for a given permission? Well, as we mentioned earlier, the AdminPermission accepts a filter string as its name like for example:

```
new AdminPermission("(&(signer=o=foo)(name=org.foo.*)(location=file://*)
  (id>=10)"), AdminPermission.LIFECYCLE+ "," + AdminPermission.EXECUTE);
```

Granted, that looks a little complicated but it really is just showing off all four possible filter parameters in a single filter. We can see that the bundle must have been signed by a certificate that has an o=foo entry (we will discuss signing and matching of certificate filters in more depth later in this chapter) and a symbolic-name that starts with org.foo (as with the other permissions before the * is a wildcard that matches anything). Furthermore, we limit the location of the targeted bundles to be on the filesystem and their bundle id to be greater or equal 10.

12.3.4 ServicePermission

Ok, where are we now? As you probably can see we have worked our way through the module and the lifecycle layer in terms of permissions. As you might have guessed, finally, a possibility is needed to give you a possibility to secure the service layer as well. This is where the ServicePermission comes into play.

Basically, what you want to be able to express is a bundle's authority to register or get a service. This is a very simplistic approach but nevertheless a very powerful one when you think about it. The benefit is that in many cases security concerns can be addressed by just denying or granting access to a service or not. Sure, in some cases you would need more finegranied control as protecting the individual methods of a service object but that is when you will have to create and define your own permissions. The OSGi specification only goes as far as the service object but in many cases that is already enough.

As with the other permissions, the ServicePermission centers around the name which represents the name of the service interface as a comma separated string. Wildcards may be used and the two possible actions are GET and REGISTER. Representing the authority to either get or register a service with the given name, respectively. How does that look like? Consider the following,

```
context.getServiceReference("org.foo.Service");
context.registerService("org.bar.Service", new Service(), null);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

As you can see, we are trying to get a reference to a service with the org.foo.Service interface in the first step and registering a service with the org.bar.Service interface in the second step. The system would in this case need to perform the following security checks,

```
System.getSecurityManager().checkPermission(
  new ServicePermission("org.foo.Service", ServicePermission.GET));
System.getSecurityManager().checkPermission(
  new ServicePermission("org.bar.Service", ServicePermission.REGISTER);
```

As you can see, the check is pretty straightforward as we only need to create and check for the ServicePermission with the given interface as a name and the GET or REGISTER action, respectively. By now, it should be easy for you to guess what the correct permissions you need to grant to the bundle look like but just to make sure, they need to look somewhat like the following,

```
new ServicePermission("org.foo.*", ServicePermission.GET);
new ServicePermission("org.bar.Service", ServicePermission.REGISTER);
```

The first permission uses a wildcard to allow access to all services that are inside the org.foo package and the second represents the authority to register services with the org.bar.Service interface.

But wait, actually, it is not as simple as we make it look as we have a another layer involved. Can you guess what we are talking about? It is a bit tricky but we already mentioned that service references returned by a call to the getServiceReference() methods are filtered by the visibility of the requested interfaces to your bundle (as you hopefully remember, otherwise have a look again at xxx). Now, maybe its obvious but it might be that your bundle doesn't have the package permission to the interface it requests a service object for. Thats tricky right? Well, in this case, the same happens as if the bundle could not see the package in question namely, the service gets filtered out despite the fact that the bundle has the needed service permission.

12.3.5 Relative FilePermissions

Finally, we have to mention at this point that there is a standard Java permission which is impacted by the way the framework will interpret it namely, java.io.FilePermission. This difference is that in normal circumstances, relative FilePermissions i.e., a file permission that has a relative path as its name, will assume that the root of the relative path is the normal root (i.e., the directory the java command has been started from). This is not the case in an OSGi environment. Rather, the root for the relative path is assumed to be the root of the data storage area of the bundle in question. Why is that? Well, there is no way to determine where the root of a bundles data storage is as it is not mandated by the specification. Nevertheless, in certain cases, you want to be able to use the wildcard mechanisms of the FilePermission in combination with its relative path feature to for example give a bundle execute permissions for all files inside its data area. Therefore, the relative FilePermissions are assumed to be relative to the bundle storage area.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

12.4 Managing Permissions

As we mentioned in the beginning of this chapter, OSGi security is based on the Java security model. However, apart form introducing and defining some additional permissions that you can use to secure your framework, it also introduces a complete new feature to the domain namely, permission management. Granted, even in the normal java security world, there is a way to define and grant permissions to protection domains by using a policy file. The Java security policy files make it possible to assign permissions mainly based on location (the so called codebase) or signer of a specific class. As you can imagine they are pretty static in nature (typically changes are only applied when you restart your JVM) also there is a possibility to re-read the policy files at runtime or replace the file based policy mechanism with your own, custom one it really doesn't sit well with the dynamic nature of OSGi frameworks. Maybe a nice analogy is the difference between plain Java and the module layer when it comes to modularity. Yes, you can get the same level of modularity without OSGi (as in fact, the module layer is implemented on top of plain Java) but the main question is do you really want to (especially with something as powerful as OSGi at your disposal already).

Alright then, what get is management API (as you might guess, we are talking services here) that allows you to assign permissions to bundles, store the information persistently, and change the configuration at runtime. One thing to point out to avoid confusion later on this that there are two kind of services around namely, the PermissionAdmin and the ConditionalPermissionAdmin. The reason is mostly historical as the PermissionAdmin has been around first and the ConditionalPermissionAdmin was introduced in later versions. We will describe both services starting with the PermissionAdmin in the following subsection followed by the ConditionalPermissionAdmin in the next section. For now, just keep in mind that there are two services around and that in case both are used then the location bound permissions of the PermissionAdmin service override any information of the CondtionalPermissionAdmin service. Otherwise, the concepts of the ConditionalPermissionAdmin service apply. The reason we mention this fact at this point is that the relationship between the two is well defined and even as it can be summarized as we just did we still will need to mention the later sometimes while describing the former in order to highlight some finegrained effects of this policy. But now lets have a look at the PermissionAdmin service.

12.4.1 PermissionAdmin

So in order to be able to use security in any meaningful way we need to be to reason about the security policy currently in effect (by which we mean to say lookup the permissions that currently are granted) and allow the management agent to assign and remove permissions at least on a per bundle basis. The PermissionAdmin service does just that namely, it provides you with information about current permissions and allows a management agent to set permissions per bundle.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

The way this works is that permissions are assigned to a bundle location. So when a bundle does attempt to do some security sensitive action only the permissions assigned to the location of the bundle are evaluated. Now, you might be thinking that this is not really enough given that a bundle's identity is not really coupled with the location anymore and you would be right. This is the main reason the ConditionalPermissionAdmin has been created which gives you more power when it comes to assigning permission. For now, we are stuck with the old way of doing it which is based on the bundle location and additionally, on a set of so called default permissions. Default permissions are the permissions that apply if no entry for the location of the given bundle exists. We will give you an example soon but for that we need to first introduce you to the format used to describe permissions. Why is there a new format and you can not just use the Permission objects themselves? Well, the answer is actually pretty straightforward if you think about it a little namely, the bundle that assigns the permissions might not have access to the classes at the time it assigns them to a location. Therefore, a PermissionInfo is defined which we have a look at next.

12.4.2 PermissionInfo

As we just said, the reason that we need a PermissionInfo abstraction is that we might be in a situation where we can not instantiate the Permission because the bundle that needs to assign the permission doesn't have access to the class. The PermissionInfo is used in this case to encapsulate the three pieces of information that we need to know about when we have to instantiate the permission later on at the time of the actual security check namely:

- type - class name of the permission
- name - name argument of the permission
- actions - actions argument of the permission

It really is as simple as it sounds. All we need to do is to create a new PermissionInfo that accepts the three needed pieces of information as arguments to its constructor. Lets have a look at a simple example,

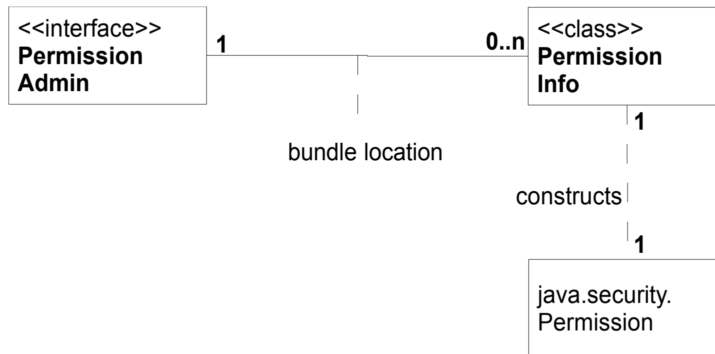
```
new PermissionInfo(
  AdminPermission.class.getName(), "(id=10)",
  AdminPermission.EXECUTE);
```

As you can see, we create a PermissionInfo for an AdminPermission that will have the filter (id=10) as its name and EXECUTE as its action. In other words, if we assign this PermissionInfo to the location of a bundle using the PermissionAdmin service then the bundle with this location will have the permission to execute (i.e., start/stop) bundles assuming that the target bundle has a bundle id that equals 10. With that, lets cut to the chase and see the PermissionAdmin in action.

12.4.3 PermissionAdmin in Action

By now, it should be pretty obvious what the PermissionAdmin service looks like from a high level perspective. It really is as simple as shown in Figure xxx.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507



Ch2Figure 12.2: Class Diagram org.osgi.service.permissionadmin

The PermissionAdmin service maintains a set of PermissionInfo objects for any number of unique bundle locations. When a security check is invoked, then for each BundleProtectionDomain on the stack the location of the individual bundle is used to lookup its PermissionInfo set from the Permission admin or the default set of PermissionInfos if no such bundle location has been assigned. Subsequently, for each PermissionInfo the described Permission is instantiated and checked against the required permission. If any permission grants the required permission the check continues until all BundleProtectionDomains are processed and the action is granted otherwise a security exception is thrown.

In case that the ConditionalPermissionAdmin is present (i.e,. provided by the framework) then the default permissions are ignored. That means that in case we don't have a specific entry for the bundle location in question we don't default to the default permissions but let the ConditionalPermissionAdmin handle the situation.

Now, how do we bootstrap this process? Surely, not every bundle can be allowed to access and modify the PermissionAdmin? True, and the answer is a little tricky but essentially boils down to the first bundle wins. To this end, the idea is that if no default permissions have been set, the default default permissions are AllPermission. This way, as long as nobody has used the PermissionAdmin, anybody is free to use it. On the other end, all methods of the PermissionAdmin require AllPermission. So as soon as you replace the defaut permissions (or rather, set them for the first time) the security of the framework is yours as long as you remember to give yourself AllPermisssion as a first step.

Again, the first thing a management agent has to do is give itself AllPermission. Then he can go on to give permissions to other bundles as he sees fit. The reason that we just switched from you to the management agent is to underline the fact that there is a race condition here again that needs to be taken care of. It is the same we have seen a couple of times when talking about the concept of a management agent namely, the need to have only one and have it be the one entity that controls the framework. Same with security. Your

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

management agent should be the one that controls the permissions of the framework and it can do so by being the first bundle that gets started. If you allow a different bundle to execute before your management agent then it is free to take over the security of the framework as all bundles have initially AllPermission.

That said, lets zoom in a little and see what it will look like to actually use the PermissionAdmin service. The interface of the service looks like:

```
package org.osgi.service.permissionadmin;
public interface PermissionAdmin{
  PermissionInfo[] getDefaultPermissions();
  String[] getLocations();
  PermissionInfo[] getPermissions(java.lang.String location);
  void setDefaultPermissions(PermissionInfo[] permissions);
  void setPermissions(String location, PermissionInfo[] permissions);
}
```

The first thing we need to do (as we just established) is to give ourself AllPermission in order to take over the security of the framework.

```
PermissionAdmin admin = getPermissionAdmin();
admin.setPermissions(
  context.getBundle().getLocation(),
  new PermissionInfo[]{
    new PermissionInfo(
      AllPermission.class.getName(), "", "")});
```

This looks a little complicated but all we do is to get the PermissionAdmin (which is just a normal service lookup) and call its setPermission() method with the location of your own bundle and a new PermissionInfo array with a single PermissionInfo that encapsulates an AllPermission. With that in place, we now have AllPermission and can proceed to set the default Permissions or rather to unset the default permissions in order to deny other bundles any permission.

```
PermissionInfo[] previous = admin.getDefaultPermissions();
admin.setDefaultPermissions(new PermissionInfo[0]);
```

This is not strictly necessary but nevertheless, you really want to define your own set of default permissions directly after you assigned yourself AllPermissions. This way, you can make sure that the default is what you actually want for bundles you don't really consider. Now, why do we capture the previous default permissions? Again, this is a best practice. As with most things a bundle does, it should make sure that it leaves the framework in a good state when it leaves and in our case, we want to be to set the previous default policy again when we are stopped like this,

```
admin.setDefaultPermissions(previous);
```

But this is only when we should be stopped and not really important for the more interesting question namely, what other permissions might have been assigned and how can we assign permissions to other bundles? Well, the former can be done using the a combination of methods provided by the service like the following,

```
String[] locations = admin.getLocations();
for (int I = 0;(locations != null)&& (i < locations.length);i++){
  Permission[] permissions = admin.getPermissions(locations[i]);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    ...
  }
```

This allows us to iterate over all previously set permissions and for example store them away as we did with the default permissions in order to restore them when we are stopped. Notice that in this case we have to be careful to remember that we already added ourself with our AllPermission entry. But more importantly, we can use a similar loop to make sure that for example all specific bundle permissions are removed (again, we would need to be careful not to delete our own AllPermission entry).

```
  String[] locations = admin.getLocations();
  for (int i = 0;(locations != null) && (i < locations.length);i++){
    if (!context.getBundle().getLocation().equals(locations[i]) {
      admin.setPermissions(locations[i], null);
    }
  }
```

Assuming we did all these steps we now can be sure of two things. First, no other bundle has any permission except us and second, we have a record of the previous security policy that we can restore should we be stopped. With that, we are free to continue and start to give the bundles we are interested in specific permissions using their bundle locations and to assign meaningful defaults for bundles we don't really know using the default permissions. All of this is possible using the mechanisms and methods we already showed.

That wasn't that bad was it? By now you should have a good understanding about the basic java security model and how it is enforced at runtime. You know about the OSGi specific permissions available and what they do. Finally, you know how to use the PermissionAdmin service to define your own security policy based on the bundle location. So what else do you need to know? Well, as we already mentioned, for a long time this was what has been given to you but ever since release 4 of the specification there is the ConditionalPermissionAdmin which adds a new concept that makes your life a lot simpler plus, it reconciles the security layer with the new notion of bundle identity which isn't really based on bundle locations anymore but rather on an abstract concept of a bundle. We will learn about both in the next section.

12.5 Conditional Permission Management

In many ways, the Java security model is relatively static in nature. Unless heavily customized it assigns a set of permissions to a code base or signer. The first attempt of a new security model suitable for an OSGi environment has been the PermissionAdmin which we introduced you to in the previous section. Its main benefit is to provide a management infrastructure that allows to assign and change permissions on a per bundle basis during execution time. As we discussed already, the hard coupling to the bundle, however, as well as its hard coupling to the bundle location as a unique identifier makes it still rather limited when it comes to expressing complex security policies. The upside being that it is relatively simple.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

A different, related but somewhat more complex model, is provided by introducing the notion of a condition to the general model. The main idea is to have a set of conditional permissions where each can be applicable for any bundle if it fulfills the right conditions. This sounds more complicated then it is as in reality all there is to it is a tuple of a set of conditions and a set of permissions. Assuming that a bundle in questions satisfies all the conditions in the condition set at the time of the permission check then all the permissions in the permission set are applicable for check of this bundle. Just think about what this means for you when want to express your security policy for a second. Instead of having to define what permissions are granted to a specific bundle ahead of time you can now just define a set of conditions that have to be fulfilled in order to have a certain set of permissions (i.e., rights).

Another way of understanding the concept is to look at the normal Java policy and its possibilities to assign permissions based on codebase and on signer. Really, what it comes down to is that the codebase and the signer attribute are conditions that can be fulfilled or not. If they are then the permissions are applicable otherwise not. The condition model introduced here is more or less an extension of this idea where one the one hand the conditions are more powerful and on the other hand the set of possible conditions can be extended arbitrarily as it is the case with permissions already in the Java model. This makes it possible to create completely new kinds of policies that for example take time, location, or even remote advice (like from talking to a central server) into account when it comes to the actual security check.

Hopefully, by now you are able to see that the general idea of introducing conditions to the security model has a lot of potential and while it adds some complexity it is well worth the pain. In this section we will introduce you to the ConditionalPermissionAdmin service which is the counterpart to the PermissionAdmin service and its replacement in the long run. Regarding the complexity added to the model, just keep in mind that we will come back to this in the following section when we talk about ways to deal with it and to simplify some of the tasks need to define a reasonable security policy. But now, lets have a look at the ConditionalPermissionAdmin first.

12.5.1 ConditionalPermissionAdmin

Like its predecessor, the PermissionAdmin service, the ConditionalPermissionAdmin service (around since version 4.0 of the OSGi core specification) is the one place to go to in order to define and maintain your security policy. It introduces a new way of doing permission management by defining the concept of conditional permission management. In a nutshell, the idea is that for each bundle on the call stack permissions are granted only if the bundle satisfies the right conditions at the time of the security check. Thus, a security policy essentially is expressed by a number of condition and permission set tuples. If all conditions of the condition set of a tuple are satisfied by a given bundle then the supplied permissions in the tuple apply for this bundle during this permission check.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Due to this much more flexible and powerful model, the CondtionalPermissionAdmin is what you should use exclusively for new projects. The only reason that you turn to the PermissionAdmin would be that you need to secure a framework that is not at least at version 4 of the specification (which will rarely be the case nowadays). In the reminder of this section we will zoom in on the condition part next and then give you an example of how to use the ConditionalPermissionAdmin service in action.

12.5.2 Conditions

By now, it should be clear that the actual purpose of a condition is to act as guard which decides if a permission set is applicable or not. Subsequently, the condition must be evaluated when a security check is trigged against all bundles on the callstack. The way this done is that for each bundle protection domain that is checked at the first time a condition is found it gets instantiated with a reference to the bundle in question. From then on, it will be used for each evaluation this protection domain takes part in. The evaluation itself is pretty straight-forward as all that is done is that the isSatified() method of the condtion is called returning either true or false depending on whether the bundle in question satisfies the condition or not at this point in time. In case you are thinking ahead a little bit you might have two questions about this namely, isn't that kinda slow and what about side effects. At least, these are the two main issues addressed by the conditions other then purely deciding whether they are satisfied or not. Lets look at these two concerns in order now and explain how they are addressed.

The the first question was whether the idea of evaluating all conditions for all bundles on the callstack on every security check isn't a very expensive thing to do. The answer is that, yes, this could be a real performance bottleneck and hence, conditions provide a mechanism to mitigate these cost to some degree at least in a lot of cases. The way this is done is that they provide the possibility to query whether they are mutable or not. This allows for significant evaluation optimization as immutable conditions need to be evaluated only once per bundle protection domain (during the first security check they take part in) and subsequently, can be optimized away as they either will always remain in their state after that. For this purpose, a condition has a isMutable() method which returns true or false depending on whether the condtion is mutable or not. A good example might be a condition that mimics the way that the PermissionAdmin service works by using a given bundle location as requirement to be satisfied. As the bundle location of the given bundle for this instance of the condition is not going to change, we only have to evaluate the requirement once as subsequent evaluations will always yield the same result.

The second question regarding side-effects is a bit more complicated to explain but makes perfect sense once you get more accustomed to the condition model. Lets try to explain this by ways of an example. Imagine you want to create a condition (and in fact, it is possible to define and use custom conditions as you will learn in a little bit) that asks the user whether it should be satisfied or not. We can imagine the usual pop-up dialog that ask

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

the user whether a given bundle should have the permission (obviously, we could combine that with the mutable feature by adding a checkbox that lets the user "remember this decision"). Now, what we probably don't want is that the pop-up is happening for each bundle on the call-stack and the user needing to acknowledge each at a time (as this would be rather tedious for the user) and worse, that after he granted the permission for all bundles it still fails because the last bundle (for example) is having another condition in the set not being satisfied. Clicking on "allow" a hundred times (for a hundred bundles) and then getting a security exception message as a reward wont make many users happy. So what we need is two things namely, we need to be able to postpone conditions to underline that they should be only evaluated if all other (not postponed) conditions are satisfied and we must be able to group the evaluation for the same kind of condition into a single evaluation for the group containing all bundles. With these two features we can easily make the user experience for our little example a lot more pleasant by postponing the condition and grouping the evaluation so that the user only sees the pop-up dialog in case that all other conditions are already satisfied for all bundles and that the resulting permissions are actually implying the required one (so that in case the user "allows" the result is that the security check actually is successful). Furthermore, the grouping enables us to present the user with a single dialog listing all the bundles in question rather then an individual dialog per bundle (the hundred times clicking).

Fortunately, that is exactly what the second feature of conditions gives us. They can be postponed (as indicated by yet another method isPostponed()) which actually combines the two desired features. On the one hand, it will make sure the condition is evaluated only if all other not postponed conditions are satisfied and that it is evaluated grouped together with all other bundles on the callstack. As you can see, that is quite a powerful feature and we will show you a detailed example later in this chapter when we show you how you could use them implement such a more advanced use-cases in detail. For now lets have look a one of the conditions that are provided by the default and how it can be encoded using a ConditionInfo.

ConditionInfo and BundleLocationCondition

As conditions are the new way of defining which bundle gets what permissions obviously you well need a couple of them to be able to express anything meaningful in your security policy. Therefore, a couple of conditions is include by default and one of them allows you to mimic the behavior you already know from the PermissionAdmin namely, the BundleLocationCondition. As the name implies already, what it does is to match a given location to the location of the bundle in question and is satisfied if the given location (which my include wildcards) matches the bundle location. Since we already used a similar scenario as an example for why condition can be mutable or not it might be worth mentioning that the BundeLocationCondition is immutable and as it doesn't have any unwanted side effects nor needs grouping is not postponed. In other words, it works pretty much the same as the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

PermissionAdmin service except that in this case you must encode it inside a ConditionInfo in order to use it (i.e., give it to the ConditionalPermissionAdmin). As an example consider the following two BundleLocationConditions which we encode inside ConditionInfos,

```
new ConditionInfo(BundleLocationCondition.class.getName(),
  new String[] {context.getBundle().getLocation()});
new ConditionInfo(BundleLocationCondition.class.getName(),
  new String[] {"*://foo.baz/*"});
```

As you can see, the ConditionInfo constructor accepts two parameters. The first being the name of the condition class as a string and the second is an array of strings which are the parameters of the condition (and hence, are specific to each specific condition). In our case, the BundleLocationCondtion expects one parameter namely, the location filter string. Subsequently, the first condition will be satisfied by bundles that have the same location as our bundle (as we use the location of our bundle directly as a parameter) while the second condition accepts all bundles that are coming from the foo.baz domain. Notice, we use wildcards not only to express that we want the condition to be satisfied by all bundles from the foo.baz domain (the second wildcard) but also to express that we don't care about the protocol schema (i.e., it could be http, ftp, etc.).

Since the 4.2 release of the specification, the BundleLocationCondition accepts a second parameter as well, namely, the "!" string indicating that the result of the evaluation must be negated. So the following would match all bundles except a bundle with the location of file:bundle/foo.jar.

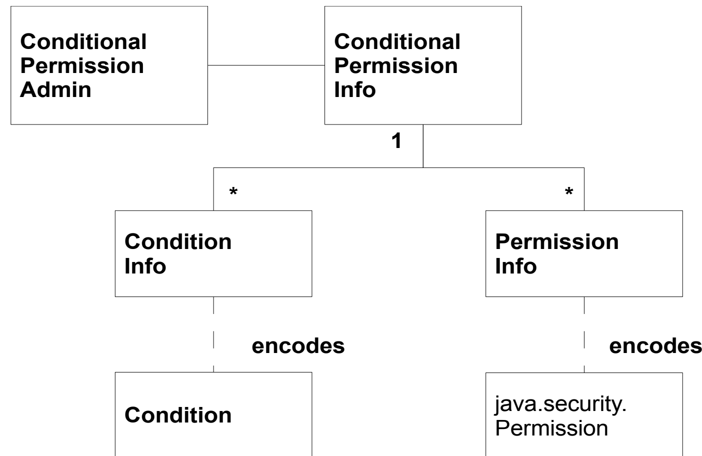
```
new ConditionInfo(BundleLocationCondition.class.getName(),
  new String[] {"file:bundle/foo.jar", "!"});
```

Ok, with that, you know what you need to know about conditions to get started with some ConditionalPermissionAdmin action. We will come back to conditions in more detail later including introducing you to the other build-in condition namely, the BundleSignerCondition and an example of how you can create and make available your own custom conditions. For now, lets go and have a look at the ConditionalPermissionAdmin service next.

12.5.3 ConditionalPermissionAdmin in Action

By now, it should be pretty clear what the ConditionalPermissionAdmin service looks like conceptually. Basically, the service maintains a set of ConditionalPermissionInfo objects that in turn contain a set of CondtionInfos and a set of PermissionInfos each. Have a look at figure xxx to see what it looks like.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507



Ch2Figure 12.3: ConditionalPermissionAdmin overview

As you can see in the figure, the high-level view of the ConditionalPermissionAdmin isn't all that different from what the PermissionAdmin service did look like. The same is true for some of the usage details as well. The interface of the service looks like:

```
package org.osgi.service.condpermadmin;

public interface ConditionalPermissionAdmin {
  ConditionalPermissionInfo  addConditionalPermissionInfo(
    ConditionInfo[] conds, PermissionInfo[] perms);
  AccessControlContext getAccessControlContext(String[] signers);
  ConditionalPermissionInfo getConditionalPermissionInfo(String name);
  Enumeration getConditionalPermissionInfos();
  ConditionalPermissionInfo setConditionalPermissionInfo(String name,
    ConditionInfo[] conds, PermissionInfo[] perms):
}
```

Again, the first thing to do when putting your security policy in place is to give your bundle Allpermission. The difference is now that you need to use conditions to identify your bundle rather then just providing the location as the identifier. Fortunately, the location condition introduced in the last section allows us to stay close to the example we did give for the PermissionAdmin which now looks like the following,

```
ConditionalPermissionAdmin condPermAdmin =
getConditionalPermissionAdmin();
condPermAdmin.addConditionalPermissionInfo(
 new ConditionInfo[] {
 new ConditionInfo(
   BundleLocationCondition.class.getName(),
   new String[]{context.getBundle().getLocation()})
  },
 new PermissionInfo[] {
  new PermissionInfo(
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
 AllPermission.class.getName(), "", "")
});
```

We first get the ConditionalPermissionAdmin and then add a single condition permission tuple using the addConditionalPermissionInfo() method. The first argument is an array of ConditionInfo objects which in this case contains only a single condition description namely, the BundleLocationCondition which gets the location of our bundle as an argument. The second argument to the method is an array of the actual Permissions that we want to grant when the conditions of the condition permission tuple are satisfied and subsequently, we add a single PermissionInfo object which describes the desired AllPermission.

With the AllPermission in place for our bundle we now have to make sure that we don't have any unexpected entries made already as we want to be the only one responsible for the security policy of this framework. So what other permissions might have been assigned and how can we assign permissions to other bundles? Well, the former can be done using the a combination of methods provided by the service like the following,

```
for(Enumeration e = condPermAdmin.getConditionalPermissionInfos();
  e.hasMoreElements();){
  ConditionalPermissionInfo info =
    condPermAdmin.getConditionalPermissionInfo((String) e.nextElement());
…
}
```

This allows us to iterate over all previously set condition and permission tuples, using the getConditionInfos() and the getPermissionInfos() methods of the ConditionalPermissionInfo object, and for example store them away as we did previously with the PermissionAdmin already in order to restore them when we are stopped. Notice that in this case we have to be careful to remember that we added ourself as well. But more importantly, we can use a similar loop to make sure that for example all specific bundle permissions are removed (again, we would need to be careful not to delete our own AllPermission entry).

```
for(Enumeration e = condPermAdmin.getConditionalPermissionInfos();
  e.hasMoreElements();){
  ConditionalPermissionInfo info =
    condPermAdmin.getConditionalPermissionInfo((String) e.nextElement());
  info.delete();
}
```

Assuming we did all these steps we now can be sure of two things. First, no other bundle has any permission except us and second, we have a record of the previous security policy that we can restore should we be stopped. With that, we are free to continue and start to implement our own security policy for other bundles. However, there is one more thing to point out which is shown in the above example as well. The point is that in the case of the PermissionAdmin we had a key to the permissions already as we could just use the location as the key to address them. In the case of the ConditionalPermissionAdmin this is not possible anymore and that's why a custom "name" is introduced. Obviously, the keys we use via the Enumeration in the example are therefore the names if the available infos.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

If the ConditionalPermissionInfo has been created using the addConditionalPermissionInfo() method then a unique name will be automatically created by the service which you can get access to using the getName() method on the info itself. In case that you want to set the name ourself then you can create or update (in case there is already one with the given name) a ConditionalPermissionInfo as follows,

```
condPermAdmin.setConditionalPermissionInfo("Management",
  new ConditionInfo[] {
  ...
  },
  new PermissionInfo[] {
  ...
  });
```

We just use the setConditionalPermissionInfo() method which expects the name as its first argument (in this example "Management") and will either create the info or update it in case there was already an info defined with the given name.

Thats it in a nutshell. You are now able to create your own condition based security policies using the CondtionalPermissionAdmin service. That wasn't that hard right? But wait, there is more. Essentially, what we did show you was the r4.1 way of using the ConditionalPermission admin. With the advent of the r4.2, we now have a couple of methods more that make the tasks of managing the condition/permission tuples even easier. The new interface still contains all the method of the one shown above plus the following methods,

```
public interface ConditionalPermissionAdmin {
  …
  public ConditionalPermissionUpdate newConditionalPermissionUpdate();
  public ConditionalPermissionInfo newConditionalPermissionInfo(
    String name, ConditionInfo[]conditions,PermissionInfo[] permissions,
    String access);
  public ConditionalPermissionInfo newConditionalPermissionInfo(
    String encodedConditionalPermissionInfo);
}
```

Specifically, the update of the condition permission table has been simplified a lot by the introduction of a ConditionalPermissionUpdate object. Have a look at the following example,

```
ConditionalPermissionAdmin admin = null;
ConditionalPermissionUpdate update =
admin.newConditionalPermissionUpdate();
```

As you can see, we can now get an update object from the service. From that, we can get a representation of the current table like this,

```
List infos = update.getConditionalPermissionInfos();
```

As you probably can see, this makes our task of backing-up and deleting all currently granted permissions a lot easier as all we need to do is the following,

```
List oldInfos = new ArrayList(infos);
infos.clear();
```

In case you are wondering "but what about my own permissions?", the reason we can first delete all entries is that we now have a `transactional` style update mechanism which allows us to modify the representation we got as we see fit and it only becomes immediate

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

when we commit the update object. For example, if we wanted to add another condition/permission entry after we cleared the table and then commit it all we need to do is,

```
infos.add(admin.newConditionalPermissionInfo(name, conditions, permissions,
  access));
update.commit();
```

The commit method will return true or false, in case that the update was successful or not, respectively. Obviously, restoring the previous permissions becomes really simple now, like this,

```
infos.clear();
infos.addAll(oldInfos);
update.commit();
```

Pretty simple huh? One thing we still need to talk about however, is the `access` parameter we passed to the `newConditionalPermissionInfo` method. What is that all about? In a nutshell, with the r4.2 version of the service we can now create not only allow but deny policies as well.

Allow v.s. Deny Policies

So what are deny policies? Well, deny policies can significantly simplify the security configuration setup because they handle the common case of an exception to the general rule. Consider, for example, a case where a bundle should be allowed to use all exported packages except a subset of packages under a common root (let's say org.foo.*). How would we implement such a policy? The problem obviously is that we can't just enumerate all possible packages other than the org.foo.* packages and at the same time we can't allow * as we want to exclude org.foo.* packages.

This is where the access type comes into play. If we set it to be DENY then, assuming the given conditions are satisfied, the bundle will not get the specified permissions. Thats what is called a deny policiy and what makes it possible to first deny our bundle access to org.foo.* packages while second, grant access to all other packages like so,

```
infos.add(admin.newConditionalPermissionInfo("deny-org.foo-packages",
  new ConditionInfo[]{ new ConditionInfo(
    BundleLocationCondition.class.getName(), new String[]{"file:foo.jar"})
  },
  new PermissionInfo[]{ new PermissionInfo(
    PackagePermission.class.getName(),
    "org.foo.*", PackagePermission.IMPORT)
  }, ConditionalPermissionInfo.DENY));
```

> **THE PERMISSION CHECK**
>
> It is important to realize that with the addition of the access type (i.e., Allow and Deny) it is now the case the the order of condition/permission tuples becomes important. Basically, the permission check will traverse the policy table of tuples in ascending index order until the first tuple is found where the conditions are satisfied and the required permission is present. If this access type is DENY, the check will fail. If an ALLOW is found, the next bundle protection domain will be checked.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

With that in place, we now made sure that the bundle at the file:foo.jar location doesn't get access to any org.foo.* package. Consequently, whats left is to allow access to all other packages,

```
infos.add(admin.newConditionalPermissionInfo("allow-all-packages",
  new ConditionInfo[]{ new ConditionInfo(
    BundleLocationCondition.class.getName(), new String[]{"file:foo.jar"})
  },
  new PermissionInfo[]{ new PermissionInfo(
    PackagePermission.class.getName(), "*", PackagePermission.IMPORT)
  },
  ConditionalPermissionInfo.ALLOW));
```

Thats it for the conditional permission admin. In the next section we will show you how you can use bundle signing to make your life easier when specifying your security policy while at the same time you gain even more security. Let's have a look at that right now.

12.6 The simple Life

Knowing how to secure your OSGi framework by implementing your own security policy and how to write your bundles in a way that they take security into consideration is only one piece of the puzzle. The other is to find ways and methods to make your life easy enough to actually use security. We will try to give you tools and mechanisms that can ease the pain that is normally associated with two of them in this section namely, authenticate the provider of a bundle while ensuring that the content has not been modified and grant a bundle the permissions it needs (and only the ones it needs). The former can be done using digitally signed bundles using certificated to establish the identity of the provider of the bundle and the latter by a mechanism called local permissions. We will have a closer look at signing first and then talk about the local permissions at the end of this section.

So far we already introduced you to the conditional permission idea that can help a lot in implementing your security policy. While we did that you might have noticed that we mentioned signed bundles and certificates a couple of times but never really explained how to do that. We will now come to that in the next section where we show you how you can digitally sign your bundles and in the section after that where we talk about the BundleSignerCondition which gives you the means to grant permissions based on the identity of a bundle established via certificates.

12.6.1 Signed Bundles

Digitally signing allows you to verify two things namely, the authenticity of the signer and that the content has not been modified after it was signed. Typically, the authenticated signer is called a principal. In an OSGi Framework, the principals that signed a bundle become associated with that bundle. This association is then used to grant permissions to a bundle based on the principal.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

For example, instead of determining the locations of all the bundles of a given company you want to give networking permissions too, you can grant the a company the right to use networking on your devices. The company can then use networking in every bundle they digitally sign and that you deploy. Also, a specific bundle could be granted permission to only manage the life cycle of bundles that are signed by the company in question. That gives you a very simple yet powerful way of limiting what can or can not be installed in your framework.

Signing provides a powerful delegation model. It allows an Operator to grant a restricted set of permissions to a company, after which the company can create bundles that can use those permissions, without requiring any intervention of, or communication with, the Operator for each particular bundle.

Digital signing is based on public key cryptography. Public key cryptography uses a system where there are two mathematically related keys: a public and a private key. The public key is shared with the world and can be dispersed freely, usually in the form of a certificate. The private key must be kept a secret. Messages signed with the private key can only be verified correctly with the public key. This can be used to authenticate the signer of a message (assuming the public key is trusted).

The digital signing process used with bundles is based on Java 2 JAR signing and the same tools that you can use to sign a JAR can be used to sign a bundle. We will give you an example next which shows you how you can give required permissions to the bundles signed with the correct certificate. Lets see how we can get this done step-by-step and start with creating the required certificates next.Trust and Certificate Chain Example

> Does this sound a bit abstract so far? Don't worry, we will make things more clear right now. Let's assume you have a system which features a set of core bundles (lets call that the framework domain) and an arbitrary number of 3rd party bundles that can extend it (the 3rd party domain). Furthermore, we are expecting other bundles to be around but we don't want to have them make use of our framework bundles (e.g., they shouldn't be able to import any packages exported nor use any service published by framework bundles).
>
> Now the question is how can we provide a simple yet secure model to allow us and 3rd parties to provide bundles for their respective domains without knowing what bundles that are in advance? Simple, we need to create a root certificated (what is normally called a CA) which we maintain and that get's used as the root of trust by the framework. Next, we create two subcertificates one for the framework and one for the 3rd party domain.
>
> With that in place, we can sign certificates of framework and 3rd party developers/companies with the respective subcertificate and when they in turn us their certificate to sign a bundle the framework can use the root certificate to establish a chain of trust (in other words, ensure that the certificates are known and trusted).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Certificates and Keystores

In order to create certificates and their associated public and private keys we will use the keytool command provided by the JDK. It can be used to create and manage certificates inside a so called keystore which is an encrypted file defined by Java for this purpose.

For our little example we first have to create two different certificates namely a core and third-party certificate (see the sidebar at xxx for a more complete example). In a real system the private keys would need to be kept as secret as possible. For now, we are just going to create a new keystore file which will contain our new public and private key pair for our Certificates like this,

```
keytool -genkey -keystore certificates.ks -alias core -storepass foobar  \
  -keypass barbaz -dname "CN=core, O=baz, C=DE"
keytool -genkey -keystore certificates.ks -alias third-party \
  -storepass foobar -keypass barbaz -dname "CN=third-party, O=baz, C=DE"
```

As you can see, the new keystore is called certificates.ks and we create two new key pairs with an alias of “core” and “third-party”. The store is protected by the password foobar and the keys themselves have a password of barbaz. The -dname switch allows us to specify our distinguished name in this case core and third-party from the baz organisation in germany, respectively.

DISTINGUISHED NAME (DN)

An X.509 name is a Distinguished Name (DN). A DN is a highly structured name, officially identifying a node in an hierarchical name space. The DN is used as an identifier in a local name space, as in a name space designed by an Operator. In our case (CN=root, O=baz, C=DE), we have a simple “country”/“company”/“name” name-space. Notice that the traversal of a name is reversed from the order in the DN, the first part specifies the least significant but most specific part. That is, the order of the attribute assertions is significant. Two DNs with the same attributes but different order are different DNs.

The next thing to do is to sign our key pair with itself. Might sound a little strange at first but is what you need to do in order to make it the root of the tree. It's a common thing to do as you can see by the fact that the keytool command has a support for it called “-selfcert”.

```
keytool -selfcert -keystore certificates.ks -alias core -storepass foobar \
  -keypass barbaz -dname "CN=core, O=baz, C=DE"
keytool -selfcert -keystore certificates.ks -alias third-party \
  -storepass foobar -keypass barbaz -dname "CN=third-party, O=baz, C=DE"
```

The only difference to the previous command is that we use the -selfcert instead of the -genkey switch. With that, we have our key pairs which can be used to sign other certificates in order to be part of a trusted certificate chain or bundles directly. But wait, where are our certificates now? Well, for that, we need to extract them from the certificates.ks keystore first and import it again. Why that? The reason is that we want to impor as certificate

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

entries. For now our key pairs are key entries and there is no other way to change it to be a certificate entry other than to export and import it again.

```
keytool -export -v -keystore certificates.ks -alias core -file core.cert \
  -storepass foobar -keypass barbaz
keytool -export -v -keystore certificates.ks -alias third-party \
  -file third-party.cert -storepass foobar -keypass barbaz
keytool -import -v -keystore certificates.ks -alias core-cert \
  -file core.cert -storepass foobar -keypass barbaz
keytool -import -v -keystore certificates.ks -alias third-party-cert \
  -file third-party.cert -storepass foobar -keypass barbaz
```

In a real world scenario, you would want to make that different keystores. One that contains your keys with which you can sign bundles into their domains and one that contains the certificates that you can use to verify that the certificates of signed bundles are trusted. For now, we can verify that we now have a key entry and a certificate entry using the "-list" command of the keytool. Your output should look close to:

```
keytool -list -keystore certificates.ks -storepass foobar

third-party-cert, 08.01.2010, trustedCertEntry,
fingerprint (MD5): 15:9B:EE:BE:E7:52:64:D4:9C:C1:CB:5D:69:66:BB:29
core, 08.01.2010, PrivateKeyEntry,
fingerprint (MD5): CE:37:F8:71:C9:37:12:D0:F1:C8:2B:F9:85:BE:EA:61
third-party, 08.01.2010, PrivateKeyEntry,
fingerprint (MD5): 15:9B:EE:BE:E7:52:64:D4:9C:C1:CB:5D:69:66:BB:29
core-cert, 08.01.2010, trustedCertEntry,
fingerprint (MD5): CE:37:F8:71:C9:37:12:D0:F1:C8:2B:F9:85:BE:EA:61
```

Next, we need to sign our bundles using the certificate of the domain we want them to belong to.

Signing bundles

Digitally signing is a security feature that ensures that the content of a bundle has not been modified as well as authenticating the signer. In OSGi, the principals that signed a bundle JAR become associated with that bundle. This you can use to grant permissions to a bundle based on the authenticated principal.

A given bundle JAR can be signed by multiple signers and the signing itself follows the normal java JAR signing. The only additional constraint is that for a bundle, all entries inside the bundle must be included in the signature except entires below META-INF/ (while the normal java JAR signing allows for partially signed JARs). Luckily, that enables us to just use the jarsigner tool included in the jdk (by default, it will sign all entires in the given JAR). The following will sign a bundle with our core certificate,

```
jarsigner -keystore file:certificates.ks \
  -storepass foobar -keypass barbaz core-bundle.jar core
```

To sign a second bundle with our third party certificate should be pretty obvious and looks like,

```
jarsigner -keystore file:certificates.ks \
  -storepass foobar -keypass barbaz third-party-bundle.jar third-party
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Again, in this example we don't need to worry about different keystores as we did place both, the keys and their certificates inside the same keystore. Otherwise, we would need to use the keystore containing the keys as we need the private key to sign a bundle. For verification on the other hand, all we need is the certificate. We can use the jarsigner tool for verification as well,

```
> jarsigner -verify -keystore file:certificates.ks core-bundle.jar
jar verified.
> jarsigner -verify -keystore file:certificates.ks third-party-bundle.jar
jar verified.
```

As you can see, we now have two correctly signed bundles. One is signed into the core and one into the third-party domain, respectively. With that, its easy to grant permissions based on the signer of a bundle as we will see next.

Certificate Matching

In OSGi, certificates are matched by their Subject DN. Certificate chains are represented as ";" separated lists of the subject DNs of the involved certificates. DNs can also be compared using wildcards. A wildcard (i.e., a "*") replaces all possible values but due to the structure of the DN, the comparison is more complicated than string-based wildcard matching. Basically, we need to look at three different cases.

The first is if a "*" is used standalone. In this case it matches all possible siblings of the DN tree from the point onwards. Therefore, it can only stand on the left hand side of a Subject DN. For example, a DN with a wildcard that matches all nodes descending from the o=baz node from the above example looks like:

```
*, o=baz, c=de
```

It will match both, the core and the third-party subject DNs from above. The second case is if the wildcard is used as part of the right hand argument as in,

```
cn=*, o=baz, c=*
```

In this case, we would still match our two certificates but be matching more certificates as the first filter as we allow for all countries but at the time more limiting as we require the o node to be followed by a cn node. The two kind of wildcards can be combined as well, to remove the limit and only keep the broader matching like so,

```
*, o=baz, c=*
```

Lastly, matching of a DN takes place in the context of a certificate. This certificate is part of a certificate chain. Each certificate has a Subject DN and an Issuer DN. The Issuer DN is the Subject DN used to sign the first certificate of the chain. DN matching can therefore be extended to match the signer. The semicolon (";") must be used to separate DNs in a chain.

The following example matches a certificate signed by the core certificate of our example certificates.

```
*; cn=core, o=baz, c=de
```

The wildcard matches zero or one certificates, however, sometimes it is necessary to match a longer chain. The minus sign "-") represents zero or more certificates, whereas the asterisk only represents a single certificate. For example, to match a certificate where any of our two example certificates is in the chain, use the following expression:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
-;*, o=baz, c=de
```

Obviously, matching in general only applies if the certificates in the chain are trusted, or are signed by a trusted certificate. Certain certificates are trusted because they are known by the Framework, how they are known? Well, to some degree this is implementation specific but besides the implementation specific ways there is one standard way introduces in r4.2 that allows you to specify keystores with the trusted certificates via a property. The following property can be given to the framework via its configuration properties:

```
org.osgi.framework.trust.repositories
```

This property is used to configure trust repositories for the framework. The value is a list of paths of files.The file paths are separated by the pathSeparator defined in the File class. Each file path should point to a JKS key store. The framework will use the key stores as trust repositories to authenticate certificates of trusted signers. The key stores must only be used as read-only trust repositories to access public keys. The keystore must not have a password.

Ok, so how do use this matching then together with the conditional permission admin? This is what the BundleSignerConditions is for.

12.6.2 BundleSignerCondition

A Bundle Signer Condition is satisfied when the related bundle is signed with a certificate that matches its argument. That is, this condition can be used to assign permissions to bundles that are signed by certain principals.

The first string argument is a matching Distinguished Name as defined in Certificate Matching. The second argument is optional, if used, it must be an exclamation mark ("!"). The exclamation mark indicates that the result for this condition must be reversed. For example:

```
new ConditionInfo(BundleSignerCondition.class.getName(),
  new String[]{"cn=core,o=bar,c=de"});
```

This condition would match if the bundle has been signed by the core certificate of our example certificates while the following will not match if it has been signed by the core certificate but by any other certificate.

```
new ConditionInfo(BundleSignerCondition.class.getName(),
  new String[]{"cn=core,o=bar,c=de", “!”});
```

That pretty much is it on the topic of signing bundles. The only topic we might need to cover is what happens if a bundle is signed by multiple signers.

Multiple Signers

A bundle can be signed by multiple signers, in that case the signer will match against any of the signers’ DN. Using multiple signers is both a feature as well as it is a possible threat. From a management perspective it is beneficial to be able to use signatures to handle the grouping. However, it could also be used to maliciously manage a trusted bundle. For example a trusted bundle could later have a signature added by an untrusted party. This will grant the bundle the permissions of both, which ordinarily is a desirable feature. However,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

there might be unexpected effects like for example becoming eligible to manage bundles that shouldn't be manged by the bundle in question. This should be carefully considered when multiple signers are used. The deny policies in Conditional Permission Admin can be used to prevent this case from causing harm.

All right, you should now be able to use certificates to sign your bundles and grant permissions based on the signer of a bundle. With that out of the way, let's have a look at the next topic namely, how you can use local permissions to know what permissions a bundle needs.

12.6.3 Local Permissions

Now that we know how we can use certificates and the BundleSignerCondition to establish different levels of trust inside the framework lets get back to one of the biggest problems with security we mentioned in the beginning. Namely, how do I know what permissions I need to grant to a given bundle in order for it to function? Well, in standard java this is an unsolved problem and basically requires you to rely on third party information. Obviously, that doesn't work that well if we want to be able to use bundles that we don't know a lot about. Fortunately, in OSGi, we have a concept called local permissions. This concepts embodies a good working principle of security namely, to minimize permissions as much as possible while at the same time allowing the developer to define the needed permissions of a bundle instead of the deployer.

Local permissions are defined by a Bundle Permission Resource that is contained in the bundle; this resource defines a set of permissions. These permissions must be enforced by the Framework for the given bundle. That is, a bundle can get less permissions than the local permissions but it can never get more permissions. At first sight, it can seem odd that a bundle carries its own permissions. However, the local permissions define the maximum permissions that the bundle needs, providing more permissions to the bundle is irrelevant because the Framework must not allow the bundle to use them. The purpose of the local permissions is therefore auditing by the deployer.

Analyzing a bundle’s byte codes for its security requirements is cumbersome, if not impossible. Auditing a bundle’s permission resource is (relatively) straight-forward. For example, if the local permissions request permission to access the Internet, it is clear that the bundle has the potential to access the network. By inspecting the local permissions, the Operator can quickly see the security impact of the bundle. It can trust this audit because it must be enforced by the Framework when the bundle is executed. The Framework guarantees that a bundle is never granted a permission that is not implied by its local permissions. A simple audit of the application’s local permissions will reveal any potential threats.

So what does this look like in practice? We are really just talking about a file inside the bundle in a directory called OSGI-INF. The file itself must be called permissions.perm and contains a listing of all the permissions the bundle needs.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

As a simple example lets assume we provide a foo bundle that only exports a single package called org.foo. The local permissions of that bundle would be described in a file that needs to contain an entry for the required PackagePermission and be placed as OSGi-INF/permissions.perm inside the bundle. It would look like,

```
# Tuesday, Dec 28 2009
# Foo Bundle
( ..PackagePermission "org.foo" "IMPORT,EXPORT" )
```

As you can see, lines that start with a # are considered comments and are ignored. All other none empty lines describe a required permission. The format is that of the encoded PermissionInfo for that permission. Simple but very effective when it comes to auditing the security impact of a given bundle.

Well, so much for the simple life. After you made it that far, you should now have a good understanding about how you can implement security policies in OSGi. We did talk about the PermissionAdmin und the ConditionalPermissionAdmin services which allow you to grant permissions based on the location of a bundle or more general conditions, respectively. We showed how you can use the keytool and jarsigner tools to create certificates and sign bundles. Subsequently, you can use the BundleSignerCondition to grant permissions to bundles based on their signers. Finally, we introduced you to local permissions allowing to at the one hand, limit the permissions a bundle gets to the minimum while at the other hand enabling to easily audit the security requirements of a given bundle. What else would there be to know about? Well, while the location and signers of a bundle make for a solid basis of a security policy, we already told you that the condition model of the ConditionalPermissionAdmin is pretty powerful and allows you to provide your own conditions. Let's have a look at how to do that and why you might want to next.

12.7 Advanced Permission Management

The actual purpose of a condition is to act as guard which decides if a permission set is applicable or not. Subsequently, the condition must be evaluated when a security check is trigged against all bundles on the callstack. The way this done is that for each bundle protection domain that is checked at the first time a condition is found it gets instantiated with a reference to the bundle in question. From then on, it will be used for each evaluation this protection domain takes part in.

To this end, we already introduced you to the concept of mutable and immutable conditions allowing to optimize the condition evaluation. So far, however, we didn't actually told you how you can create your own conditions but only introduced two standard condtions namely, the BundleLocationCondition and the BundleSignerCondition. While the two are often all that is needed to implement a specific security policy the concept becomes much more powerful if we are able to provide our own conditions. This is actually not that hard and we will have a look at that now.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

12.7.1 Custom conditions

Obviously, providing custom conditions is a very security sensitive. We for sure don't want a malicious bundle to shadow our actual condition with a faulty one. For that reason, providing conditions isn't possible via normal bundles. In a nutshell, custom conditions are only valid if they are made available from the classpath of the framework (i.e., are provided by the system bundle). Otherwise, implementing a custom condition is pretty easy to do.

Basically, what is happening when a CondtionInfo is used to constructed a new condition instance is that the framework loads the specified condition class form the classpath and tries to call a static method on that class,

```
public static Condition getCondition(Bundle bundle, ConditionInfo info);
```

If such a method is not available it will fallback to trying to find a constructor to invoke with the following signature:

```
(Bundle bundle, ConditionInfo info)
```

Assuming one of the two methods work then the instance is used as part of the already described permission check. Obviously, for that to work the custom condition itself is required to implement the Condition interface which looks like,

```
public interface org.osgi.service.condpermadmin.Condition{
  public static Condition TRUE;
  public static Condition FALSE;
  public boolean isPostponed();
  public boolean isSatisfied();
  public boolean isMutable();
  public boolean isSatisfied(Condition[] conditions, Dictionary context);
}
```

Besides the two static condition objects which are the default objects to use for a condition that is allways TRUE or FALSE, respectively, the interface is only contains four methods. We will have a look at two of them in the next section when we talk about postponed conditions namely, the `isPostponed()` and the isSatisfied(Condition[] conditions, Dictionary context) methods. For now, we are going to look at the two other methods which are what you need for the simple case.

Probably, the easiest way to show you what you need to do is by way of an example. Lets assume we want to restrict certain permission sets to be available before a certain point in time. For example, we could imagine that we want to implement a simple license policy via our security policy where we allow the usage of a bundle until a certain date. Have a look at listing xxx where we provide a condition that we can use to make this possible.

Listing xxx: BeforeDateCondition Example

```
class BeforeDateCondition implements Condition {
  private final m_date;
  public static Condition getCondition(Bundle bundle, ConditionInfo info){
    return new BeforeDateCondition(Bundle bundle, info);#2
  )
  public BeforeDateCondtion(Bundle bundle, ConditionInfo info){
    m_date = Long.parseLong(info.getArgs()[0]);#3
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
    }
    public boolean isMutable(){
      return m_date > System.currentTimeMillis();#4
    }
    public boolean isPostponed(){
      return false;#5
    }
    public boolean isSatisfied(){
      return System.currentTimeMillis() < m_date;#6
    }
    public boolean isSatisfied(Condition[] conditions, Dictionary context){
      return false;#7
    }
  }
```

As you can see, the implementation is simple. When we get a call to the static getCondition() method we create a new instance of our BeforeDateCondition. In the constructor, we parse the date as a long from the first (and only) argument in the condition info and assign it to the m_date member. When the framework evaluates the condition, it will first check whether the condition is postponed by calling the isPostponed() method. We return false as we are not postponed. As a consequence, the framework will then call our isSatisfied() method. There, we check whether the current time in milliseconds is still lower then the given end-date in the m_date member and return the result.

Simple really. Thats all we need to do. The only thing left to explain is what happens in the call to isMutable(). The idea of mutable and immutable conditions really is to optimize condition evaluation. Assuming that a condition is immutable then the framework only needs o evaluate (i.e., call it's isSatisfied() method) one time and can cache the result. Otherwise, it needs to evaluate the condition on every check. Now, in this example, we have an interesting case as the condition can not be immutable as long as the end-date hasn't been reached (the result will change in the future). As soon as it is however, we can be immutable. Luckily, the framework will recheck whether the condition has become immutable after each evaluation. We make use of this in the isMutable() method by checking whether the end-date has been reached and if so returning true.

Ok, now that we have your custom condition what is left is two things. We need to use it as part of our security policy and therefore, make it available to the framework. The former is just as you would expect nothing different from using the standard conditions. We just change the condition info to be,

```
new ConditionInfo("org.foo.BeforeDateCondition", new String[]{<<Date>>});
```

Obviously, we need to replace the <<Date>> placeholder with desired date. The latter, we can do by either putting the condition class on the classpath of the framework or by making use of a so-called extension bundle. Let's have a look at that next.

Deliver Custom Conditions as Extension Bundles

Again, our custom condition needs to be available from the classpath. However, given the dynamic nature of OSGi, isn't there a way to still be able to deliver it at runtime? Luckily

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

there is. We can use an extension bundle that can deliver optional parts of the framework implementation and is able to contribute to the framework classpath.

Extension Bundles

Extension bundles can deliver optional parts of the Framework implementation or provide functionality that must reside on the boot class path. These packages cannot be provided by the normal import/export mechanisms.

Framework extensions are necessary to provide implementation aspects of the Framework. For example, a Framework vendor could supply the optional services like Permission Admin service and Start Level service with Framework extension bundles. An extension bundle should use the bundle symbolic name of the implementation system bundle, or it can use the alias of the system bundle, which is system.bundle.

The following example uses the Fragment-Host manifest header to specify an extension bundle for a specific Framework implementation

Fragment-Host: org.apache.felix.framework; extension:=framework

The following example uses the Fragment-Host manifest header to specify a extension bundle in general.

Fragment-Host: system.bundle; extension:=framework

An extension bundle must throw a BundleException if it is installed or updated and it specifies any of the following headers. Import-Package, Require-Bundle, Bundle-NativeCode, DynamicImport-Package, or Bundle-Activator.

How would that look like? All we need is to create a bundle that contains our custom condition with the following manifest,

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.beforedatecondition
Bundle-Name: Before Date Condition Extension Bundle
Bundle-Version: 1.0.0
Fragment-Host: system.bundle; extension:=framework
Export-Package: org.foo
```

If the extension bundle is installed, all we need to do to be able to use the condition is to actually use it in our security policy. That wasn't that hard, but what about this postponed condition thing? Well, lets look at that next.

12.7.2 Postponed conditions

Certain Condition objects could optimize their evaluations if they are activated multiple times in the same permission check. For example, a user prompt could appear several times in a permission check but the prompt should only be given once to the user. These conditions are called postponed conditions, conditions that can be verified immediately are called immediate

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

conditions. The isPostponed() method can inform if the condition is immediate or postponed. A Condition must always return the same value for the isPostponed method so that the Conditional Permission Admin can cache this value. If this method returns false, the isSatisfied() method must be quick and can be called during the permission check, otherwise the decision must be postponed until the end of the permission check because it is potentially expensive to evaluate. Postponed conditions must always be postponed the first time they are evaluated.

For example, a condition could verify that a mobile phone is roaming. This information is readily available in memory and therefore the isPostponed() method could always return false. Alternatively, a Condition object that gets an authorization over the network should only be evaluated at most once during a permission check to minimize the delay caused by the network latency. Such a Condition object should return true for the isPostponed method so all the Condition objects are evaluated together at the end of the permission check.

The Conditional Permission Admin provides a type specific Dictionary object to all evaluations of the same postponed Condition implementation class during a single permission check. It is the responsibility of the Condition implementer to use this Dictionary to maintain states between invocations. The condition is evaluated with a method that takes an array and a Dictionary object: isSatisfied(Condition[],Dictionary). The array always contains a single element that is the receiver. This is actually new in r4.2 as earlier versions of the specification could verify multiple conditions simultaneously.

THE PERMISSION CHECK IN THE PRESENTS OF POSTPONED CONDITIONS

In addition to the access check for immediate conditions, the presents of postponed conditions will change the evaluation internally but shouldn't change it as far as the outcome is concerned. In other words, from the point of view of the policy creator it doesn't matter whether a postponed or an immediate condition is used. The thing to note however, is that the postponed conditions will only be evaluated if there is no immediate entry that gives the required permissions.

As an example, lets implement a AskTheUser condition which does ask the user whether he wants to allow access or not. We will split this into two parts namely, an AskTheUser object which will present the user with a swing dialog asking to confirm a configurable question and the actual AskTheUserCondition. The latter is a postponed condition that uses the AskTheUser object to decide whether it is satisfied or not.

This is a good example for a postponed condition because you probably don't want to on the one hand bother the user about stuff he in reality doesn't need to answer and on the other, you want to postpone the (very slow) user interaction until it is really necessary. Have a look at Listing xxx which shows the AskTheUser implementation.

Listing xxx: AskTheUser

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
public class AskTheUser implements Runnable {
  private final String m_question;
  private volatile boolean m_result;

  public AskTheUser(String question) {
    m_question = question;
  }
  public void run() {
    m_result = (JOptionPane.OK_OPTION ==
      JOptionPane.showConfirmDialog(null, m_question));
  }
  public boolean ask() throws Exception {
    SwingUtilities.invokeAndWait(this);
    return m_result;
  }
}
```

As you can see, we basically just provide a constructor that accepts the question to ask as a string and when we call the ask() method a JoptioinPane confirm dialog with that question will be presented to the user. If the user confirms, the ask() method will return true, otherwise false.

With this in place, we can now let the user decide whether he wants the condition be satisfied or not as shown in the AskTheUserCondition implementation in listing xxx.

Listing xxx: AskTheUserCondition

```
public class AskUserCondition implements Condition {
  private final Bundle m_bundle;
  private final String m_question;
  private final boolean m_not;
  public AskUserCondition(Bundle bundle, ConditionInfo info) {
    m_bundle = bundle;
    m_question = info.getArgs()[0].replace(
      "${symbolic-name}", bundle.getSymbolicName());
    m_not = !(info.getArgs().length == 2 && "!".equals(info.getArgs()[1]));
  }
  public static Condition getCondition(Bundle bundle, ConditionInfo info) {
    return new AskUserCondition(bundle);
  }
  public boolean isMutable() {
    return true;
  }
  public boolean isPostponed() {
    return true;
  }
  public boolean isSatisfied() {
    return false;
  }
  public boolean isSatisfied(Condition[] conditions, Dictionary context) {
    if (context.get("result") != null) {
      if (m_not) {
         return !((Boolean) context.get("result")).booleanValue();
      }
      else {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
          return ((Boolean) context.get("result")).booleanValue();
        }
      }
      Boolean result = ((Boolean) AccessController.doPrivileged(
        new PrivilegedAction() {
          public Object run() {
            AskTheUser question = new AskTheUser("m_question");
            try {
              return question.ask() ? Boolean.TRUE : Boolean.FALSE;
            } catch (Exception e) {
              return Boolean.FALSE;
            }
          }
        }));
      context.put("result", result);
      if (m_not) {
        return !result.booleanValue();
      }
      else {
        return result.booleanValue();
      }
  }
```

As you can see, the implementation is simple for the most part. In the constructor we get the question we need to ask the user from the first argument of the ConditionInfo and replace ${symbolic-name} with the symbolic name of our bundle. Additionally, if there is a second parameter which equals "!" we set the m_not member to true. We make it a mutable postponed condition by having isMutable() and isPostponed() return true. In the isSatisfied(Condition[],Dictionary) call we then first check whether we already asked the user by looking up a "result" in the context dictionary. If it is present we just return the cached answer (optionally, inverting it if m_not is true). Otherwise, we create a new AskTheUser object with our question and call its ask() method. We will cache the result in the context dictionary and return it (again, inverting it if m_not is true).

The only slightly complicated bit is that we need to make the call to the ask() method inside an `AccessController.doPrivileged()`. The reason is that swing will need to make a lot of security calls and we want to limit the involved protection domains to the minimum (i.e., the protection domain of the condition itself). As the condition must be on the classpath it will have the protection domain of the framework which needs to have AllPermission so the call will go through but if we get an exception (for example in case that we are running headless and can't use swing) we will return false to indicate that we are not satisfied as we couldn't actually ask the user.

This concludes the section on postponed conditions. We are now able to let the user take access decisions by simply including a condition info like the following,

```
new ConditionInfo(
  AskTheUserCondition.class.getName(),
  new String[]{"Do you want to grant ${symbolic-name} AllPermission?"});
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

We can now use this info as part of a condtion/permission tuple where the permission part actually contains AllPermission and the tuple itself gets an access type of ALLOW. In case we wanted to use the condition in a tuple with an access type of DENY we could just create it with a not as a second parameter,

```
new ConditionInfo(
  AskTheUserCondition.class.getName(),
  new String[]{"Do you want to grant ${symbolic-name} AllPermission?",
  "!"});
```

With that, we could conclude the security chapter. We sure covered a lot of ground and hope that we didn't go to fast. We imagine that it probably isn't that easy to see how all of this adds up to a working security policy. Therefore, we will not finish the chapter at this point but, give a complete example for a working security policy next. It will be based on the Paint program from the previous chapters and in the next (and last) section we will present you with a working security policy for it. Hopefully, this will help you to get a better understanding of how all these things can be used together.

12.8 Bringing it all back home

Congratulations. If you have made it until this point, you should now know what there is to know about OSGi security. Granted, we covered a lot of new things on top of the normal Java security and especially, if you haven't been too familiar with that one in the first place, we acknowledge the fact that this might have been a somewhat overwhelming experience but we hope it was worth it. In order to help you see how all of this connects and is used together we want to now present you with one example that uses as much as possible of the introduced concepts and technologies.

Waiting for input

12.9 Summary

In this chapter we made you familiar with the Java security model and how it is used by OSGi to provide the infrastructure to deploy and manage applications that must run in secure environments. We showed you how to secure your applications and how to create bundles that are well behaved and easy to use in a security enabled OSGi framework. In this chapter we learned:

- That it is important to have security in mind when we write our bundles because otherwise, they will likely not be good citizens in a security enabled environment.
- Java security gives us the possibility to create secure sandboxes and that we can use the OSGi security to manage our security policies.
- OSGi provides us with the necessary permissions to be able to express security policies for bundles.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

- The Permission Admin gives an easy and reasonable simple model to assign permissions based on the location of a bundle.
- While the Conditional Permission admin introduces a completely new way of managing security by means of conditions that must be satisfied in order for certain permissions to be applicable.
- We can make our lives a lot simpler by signing our bundles with certificates and have seem tell us what permissions they actually need by providing local permissions.
- Its easy to implement and provide additional conditions which we can then use to express our security policies.
- We can use postponed conditions if the evaluation of a condition would be expensive which can help in certain scenarios to avouid performance penalties as well as given a possiblitiy to batch evaluations so that for example a user would only be asked on time whether he wants to grant a permission instead of multiple times. .

Using the paint program from the earlier chapters we learned how we can apply all of this to a real scenario by securing the paint program. With this knowledge under out belt, we can now secure our framework according to our own specific security policies and develop bundles that can work in security enabled frameworks. In the next chapter, we'll look into how we can use and provide web services using OSGi as well as how we can build web applications on top of OSGi.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

13
Web applications and services

So this is it, the last chapter, hopefully throughout the course of this book we've been able to convince you that OSGi is a technology that is both simple to use and extremely powerful. We as authors believe OSGi is the inevitable future of Java development and we hope to have convinced you of this fact too. Let's quickly review what we've covered during the course of this book: we started the book by introducing you to the core concepts of OSGi development, namely the module, service and life-cycle layers; in the middle section of the book we then moved onto practical considerations of developing OSGi including migration, managing, testing, debugging and working with legacy code; finally in this last section of the book we have so far covered a number of advanced topics including component development, embedded use cases, and how to manage security.

This final chapter will bring us right up to date with our recent technological past, namely how to build and deploy web applications using OSGi, and show you the benefits OSGi can bring to traditional web development frameworks. We will reuse a lot of knowledge from earlier on in the book to build a dynamic, distributed OSGi application, but don't worry if you skipped through, we'll provide you with relevant pointers back if you need an introduction or a refresher on concepts for which you might be lacking in context.

Almost all organizations and many individuals in the world today have some form of web presence, be this: via social networking sites, static html pages; simple one tier web applications; medium sized n tiered architectures; or massive globe spanning behemoths. Developers building these systems are familiar with a number of key technologies, namely web services for back-end communication between business tiers and web applications for user interaction via a browser.

During the course of this chapter we are going to look at a number of the key technologies in this area and show you how to integrate them with OSGi. By way of an

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

example we are going to look at a number of simple examples before moving onto extend an existing stock watcher web application from the GWT (Google Web Toolkit) tutorial to use OSGi. Firstly we will look at how to use the module and lifecycle layers of OSGi install, update and remove this application in and OSGi framework; then we will show you how to to make use of OSGi services to allow for dynamic installation and update of business logic; finally we will show you how to use web services with OSGi to make remote calls to other OSGi framework's to offload processing from the web tier of our application.

For the purposes of brevity we will focus on the OSGi aspects of these technologies and as such may skip over (or even plain ignore) some of the more complex aspects of web development and distributed computing in general. Hopefully the genuine web developers among you will be able to forgive us these transgressions. Our goal is to show you how OSGi can work in a web context not to show you how to build and manage all aspects of web applications or services. Figure 13.1 provides a simple diagram of the components of that we will be building during this chapter. Let's get started.

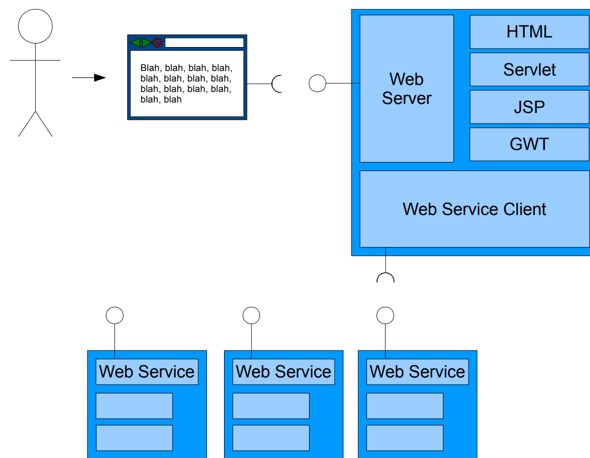


Figure 13.1 During this chapter we will build a simple web application hosted on a single OSGi framework that calls out to a number of back end OSGi frameworks via web services protocols.

13.1 Web applications

Unless you have been living on the moon for the last decade you must at some point have had some exposure to web applications, whether this is as a user or a provider. Web applications are a class of application that present their user interface via a standard web browser such as Internet Explorer, Firefox or Safari. They are in fact ubiquitous: from consumer shopping carts to online banking; from travel booking to social networking; from games to employment to government, the list is pretty much endless.

If you're reasonably familiar with Java you will know that there are a plethora of tools and technologies available to help you build such applications. In fact there are so many that it

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

would be impossible for us to cover all of the possibilities in a single chapter, instead we will pick a few of the more popular Java toolkits and show you how OSGi can improve upon their design and usage. From here you should be able to extend the general principles we cover in this section to integrate OSGi with any other toolkit of your choice. In this section we will cover the use of OSGi with the following technologies:

- Static content
- Java Servlets
- JavaServer Pages
- Google Web Toolkit

So what are the benefits that OSGi can bring to web application development that we should bother to break with the current status quo? Well if you've been following the themes of this book you will know that the major benefits of OSGi are (repeat after me):

- Modularity
- Lifecycle
- Services

By modularizing our web applications we can improve the physical and logical structure of our application's so they are easier to maintain and are easier to deploy. By using the OSGi lifecycle layer we can control when certain pieces of functionality are installed or enabled so making our applications lighter and more agile. Finally by using services we can decouple our applications making it easy to swap in different implementations and (as you will see later in this chapter) even move those implementations to other machines to improve performance. All without changing a single line of client code.

There are two main routes into the OSGi framework for web applications, either via the OSGi HttpService or as a Web Application Bundle. The HttpService is a service supplied by the OSGi compendium specification, that allows programatic registration of Servlet and static resources. A Web Application Bundle (WAB) is a Web ARchive (WAR) file that supplies the relevant OSGi meta data and relies on the OSGi frameworks lifecycle layer to control when resources are made available. Let's look first at the HttpService.

13.1.1 The OSGi HTTP service

If you are starting from scratch with your web application the simplest way of providing a web application in OSGi is to use the OSGi `org.osgi.service.http.HttpService`. We find the HttpService just like any other service in OSGi, via the BundleContext:

```
String name = HttpService.class.getName();
ServiceReference ref = ctx.getServiceReference( name );
if ( ref != null ) {
  HttpService svc = (HttpService) ctx.getService( ref );
  if ( svc != null ) {
    // do something
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
  }
}
```

If this appears to make no sense to you we suggest you review chapter [ref] on the use of OSGi services before continuing. Having found the HttpService what can we do with it? The HttpService provides a simple API to register and unregister static resources (for example images or html pages) and Java Servlets. It also provides a simple authentication scheme that backs onto the `org.osgi.service.useradmin.UserAdmin` service (covered in chapter [ref]). The API for the HttpService interface is shown in listing 13.1:

Listing 13.1 The HttpService API

```
package org.osgi.service.http;

import java.util.Dictionary;
import javax.servlet.Servlet;

public interface HttpService {
  HttpContext createDefaultHttpContext();

  void registerResources(String alias, String name, HttpContext context);

  void registerServlet(String alias, Servlet servlet, Dictionary
initparams, HttpContext context);

  void unregister(String alias)
}
```

Registering resources

Let's dive in by starting our web application with a bundle that registers a set of static resources. We'll reuse our knowledge of components from chapter [ref] to build a simple iPojo binding component that registers the resources in our bundle with an HttpService. Listing 13.2 shows the complete source of this component. For those who have skipped through this book you may wonder why we are using iPojo instead of a simple BundleActivator. The reason is due to the complex start ordering problems associated with using multiple services – you could do this without a component framework but trust us you really don't want to go there, refer to chapter [ref] for more information.

Listing 13.2 ResourceBinder class

```
package org.foo.httpservice.resourceapp;

import org.osgi.service.http.HttpService;
import org.osgi.service.http.NamespaceException;
import org.osgi.service.log.LogService;

public class ResourceBinder {
  private LogService s_log;                                          [1]

  protected void addHttpService(HttpService service) {              [2]
    try {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
      service.registerResources( "/", "/html", null );                   [3]
    } catch (NamespaceException e) {
      s_log.log(LogService.LOG_WARNING, "Failed to register static
content", e);
    }
  }

  protected void removeHttpService(HttpService service) {                   [4]
    service.unregister("/");                                                [5]
  }
}
```

At [1] the OSGi log service is bound to the object via iPojo to the ResourceBinder s_log member field. At [2] and [4] the HttpService is bound or unbound (respectively) to the ResourceBinder. The real work with respect to this example is done in [3] and [5]. At [3] we register content from the /html directory within our bundle to the root context of our HttpService. In other words the file /html/index.html from within our bundle will be served as /index.html from the OSGi HttpService. At [5] we unregister it when the service is removed.

Having defined our ResourceBinder class we now need to define the iPojo meta file which will be used by iPojo to inject byte code into the ResourceBinder class. iPojo then handles the various concurrency issues inherent in this scenario, listing 13.3 shows the meta data file that will achieve this.

Listing 13.3 ResourceBinder iPojo component

```
<?xml version="1.0" encoding="UTF-8"?>
<iPOJO>
  <component className="org.foo.webapp.servletapp.ResourceBinder">      [1]
    <requires field="s_log"/>                                            [2]
    <requires>                                                           [3]
      <callback type="bind" method="addHttpService"/>
      <callback type="unbind" method="removeHttpService"/>
    </requires>
  </component>

  <instance component="org.foo.webapp.servletapp.ResourceBinder"/>      [4]
</iPOJO>
```

At [1] we define the implementation of the resource binder component and at [2] we add a one-to-one dependency on the OSGi LogService, to be injected via the field reference. At [3] we add a one-to-one dependency on the OSGi HttpService, in this case binding the service via the methods addHttpService and removeHttpService. At [4] we construct an instance of this component which tells iPojo to instantiate a ResourceBinder object and wire together it's dependencies when the bundle in which this component is packaged is started.

So what does the end result of this look like? Figure 13.2 provides a diagram of the service and bundle level dependencies of our resource binder component.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

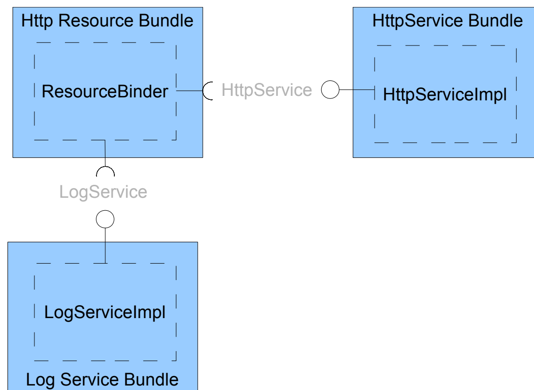


Figure 13.2 The ResourceBinder has a mandatory dependency on both the HttpService and the LogService for providing content and logging errors respectively.

In order to see this example working start up the resource binder by running:

```
$ cd chapter14/httpservice
$ ant
$ java -Dorg.osgi.service.http.port=8080 -jar launcher.jar bundles/
```

Configuring the osgi http service

The HttpService is registered by an implementation bundle and as such the client code has no control of the port or url that the service running on, that is the job of the administrator starting the osgi framework. The HttpService is defined by the OSGi Compendium specification and defines a number of configuration parameters to configure the ports that it runs on:

org.osgi.service.http.port – This property specifies the port used for servlets and resources accessible via HTTP. The default value for this property is 80.

org.osgi.service.http.port.secure – This property specifies the port used for servlets and resources accessible via HTTPS. The default value for this property is 443.

In your web browser navigate to http://localhost:8080/index.html you should see the following web page:

XXX TODO XXX

HttpContext

So if you were observant of the API specification for HttpService you may have been wondering what the HttpContext parameter the registerResources method is for – given in our previous example we passed in null. Well the answer is that it provides a way to inject policy mechanisms for resource lookup and access into the HttpService. Ok but what does

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

that mean, I hear you say? Let's first look at the API followed by an example to show you what this allows you to do. Listing 13.4 shows the API for the HttpContext interface:

Listing 13.4 The HttpContext interface

```
package org.osgi.service.http;

import java.io.IOException;
import java.net.URL;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public interface HttpContext {
  boolean handleSecurity(HttpServletRequest req, HttpServletResponse resp)
throws IOException;                                                    [1]

  URL getResource(String name);                                        [2]

  String getMimeType(String path);                                     [3]
}
```

[1] provides a callback method to allow the http service to verify if a request should be allowed for a given resource. [2] provides a mechanism to map where a particular resource is mapped to – note that the response is a URL (so it is possible to host contents from any scheme accessible via URL, if you so wish). Finally [3] provides a mechanism to control the mime type headers that are returned with the stream for a particular resource.

In our previous example we passed in a null HttpContext. In this case the HttpService uses a default implementation which can also be accessed via the helper method `HttpService.createDefaultHttpContext()` this is defined by the OSGi specification to have the following behaviour as shown in table 13.1:

Table 13.1 Default behavior of HttpContext implementations

Method	Behavior
`handleSecurity`	Implementation specific, though all open source implementations so far reviewed by the authors simply return true
`getResource`	Returns a resource from the bundle which has registered the resource or servlet. Note if security is enabled this implies that the bundle that provides the http service must be given resource AdminPermission to retrieve contents of this bundle.
`getMimeType`	Always returns null

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Let's look at an example that uses the HttpContext interface combined with the BundleTracker service we met in chapter [ref] to register resources from other bundles with the HttpService. Listing 13.5 shows the body of the addBundle method of our BundleTracker:

Listing 13.5 Tracking http resources in HttpResourceTracker

```
@Override
public Object addingBundle(Bundle bundle, BundleEvent event) {
  ArrayList<String> aliases = new ArrayList<String>();

  String[] resources = findResources(bundle);                          #1

  if ( resources != null ) {
    HttpContext ctx = new ProxyHttpContext(bundle);                    #2

    for ( String p : resources ) {
      String[] split = p.split("\\s*=\\s*");
      String alias = split[0];
      String file = split.length == 1 ? split[0] : split[1];
      try {
        http.registerResources(alias, file, ctx);                      #3
        aliases.add( alias );
      } catch (NamespaceException e) {
        e.printStackTrace();
      }
    }
  }

  return aliases.isEmpty() ? null : aliases.toArray(new
String[aliases.size()]);
}
```

At #1 we find any resources that the bundle advertises via the manifest header “HTTP_Resources”. The format of this header is a comma separated list of directories which may optionally be aliased (we'll see this working in just a second). If any resources are found then we create an ProxyHttpContext (shown below in listing 13.6) at #2 and finally register the resources with the HttpService at #3.

Listing 13.6 Proxy HttpContext that reads resources from a Bundle

```
public class ProxyHttpContext implements HttpContext {

  private final Bundle bundle;

  public ProxyHttpContext(Bundle bundle) {
    this.bundle = bundle;
  }

  public URL getResource(String name) {
    return bundle.getEntry(name);                                      #1
  }
...
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

We need to create a ProxyHttpContext as the default implementation of the HttpContext will attempt to find the resources in our tracker bundle, but in fact we want to find the resources in the bundle that is being tracked. The key line of code in this class is shown at #1 which passes the getResource call through to the registered bundle. In order to use our resource tracker we define a trivial bundle that contains no code and in fact just packages up some resources with our header as shown in listing XXX:

```
module: org.foo.http.resource
Include-Resource: html=html,images=images
HTTP_Resources:/resource=html,/resource/images=images
```

If we deploy this bundle in our osgi runtime along side the HttpService and our ResourceTracker then the resources are bound and we can browse them on the url `http://localhost:8080/resource/index.html`. In fact this is just one very trivial usage of the HttpContext object, other possible scenarios might include:

- Manage authenticated access to web content served by the HttpService
- Mapping local file system resources into the HttpService
- Others?

Now that we're familiar with registering static resources with the HttpService let's move on and look at how the HttpService can be used to hook in Java Servlets into an OSGi environment.

OSGi and servlets

Java servlets are the building block upon which a vast number of web applications have been built. The interface is relatively simple to understand, and there are a huge number of tools and framework's available to help you develop web applications based on this specification. As with static content the HttpService provides us with a mechanism to dynamically register servlets with the running service, in this case the method in question is:

```
void registerServlet(String alias, Servlet servlet, Dictionary initparams,
HttpContext context);
```

With static content we had to look up the HttpService and then call registerResource method once we'd found the service. We showed you how to use the bundle tracker and the configuration admin service to dynamically register content. Can we do the same for servlets? Well yes as it happens but this time instead of tracking bundles we need to track services. Listing 13.7 provides a trivial example of how to track Servlets in the osgi registry that are published with a "`web-contextpath`" attribute:

Listing 13.7 Tracking servlets in the OSGi service registry

```
static class ServletTracker extends ServiceTracker {
  private final HttpService m_http;

  public ServletTracker(BundleContext ctx, HttpService http) {
    super(ctx, buildServletFilter(ctx), null);                          #1
    m_http = http;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
  }

  @Override
  public Object addingService(ServiceReference reference) {
    Servlet servlet = (Servlet) super.addingService(reference);          #2
    String servletContext = (String) reference.getProperty("web-
contextpath");                                                           #3
    try {
      m_http.registerServlet(servletContext, servlet, null, null);       #4
    } catch (ServletException e) {
      e.printStackTrace();
    } catch (NamespaceException e) {
      e.printStackTrace();
    }
    return servlet;
  }
```

Here we show the tracker constructor at #1 which uses the filtered form of the ServiceTracker super constructor which uses the following buildServletFilter method to track Servlet services with a specified service attribute.

```
private static Filter buildServletFilter(BundleContext ctx) {
  String ldap = "(&(" + Constants.OBJECTCLASS + "=" +
Servlet.class.getName() + ")(web-contextpath=*))";
  try {
    return ctx.createFilter(ldap);
  } catch (InvalidSyntaxException e) {
    throw new IllegalStateException(e);
  }
}
```

At #2 we read the service from the OSGi registry and read the context path with which it is intended to be registered from it's service attributes at #3. Finally we register the service with the HttpService at #4.

SERVLETCONTEXTS AND HTTPCONTEXTS

One important note that you should be aware of is that the HttpService specification specifies that only Servlet objects that are registered with the same HttpContext object are considered to be a part of the same ServletContext. Or to say this another way the HttpService implementation creates a ServletContext for each unique HttpContext object that is registered. If null is passed in this results in the HttpService calling createDefaultHttpContext so each servlet registered with a null HttpContext is considered to be in a separate ServletContext.

Paxweb support

Before leaving this section we should also note the support provided by the Pax Web project hosted at www.ops4j.org. This provides a number of bundles at the core of which is the WebContainer interface that extends the HttpService. This new interface provides a number of extra methods to register other servlet related services, including Java Server Pages, Filters and Event listeners.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

We'll not go into depth on this as it is not part of the core OSGi specification. But we will work our way through a simple example which allows us to run one of the examples from another Manning publication “Web Development with JavaServer Pages” within an OSGi context. Listing 13.8 shows a declarative service Java component that registers our JSP's when the WebContainer service is published to the OSGi registry.

Listing 13.8 Binder to register jsp pages in the PaxWeb WebContainer

```
package org.foo.webapp.jspapp;

import org.ops4j.pax.web.service.WebContainer;
import org.osgi.service.http.HttpContext;

public class Binder {
  private volatile HttpContext http;

  protected void bindWebContainer(WebContainer c) {
    http = c.createDefaultHttpContext();
    c.registerJsps(null, http);                                       #1
  }

  protected void unbindWebContainer(WebContainer c) {
    c.unregisterJsps(http);                                           #2
    http = null;
  }
}
```

This component registers all JSP pages within the bundle under the web container context at #1 and unregisters the JSP's at #2. Listing 13.9 shows the declarative services component specification for this component.

Listing 13.9 Declarative Service component definition for JSP binder

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="sample.component" immediate="true">
  <implementation class="org.foo.webapp.jspapp.Binder" />             #1

  <reference name="webcontainer"
        interface="org.ops4j.pax.web.service.WebContainer"            #2
        cardinality="1..1"
        policy="static"
        bind="bindWebContainer"
        unbind="unbindWebContainer"
    />
</component>
```

At #1 we create an instance of our jsp binder and at #2 we lookup the Pax WebContainer service and inject it into our binder via the named bind and unbind methods. We'll leave it as an exercise to the reader but you could also trivially extend this to use the BundleTracker pattern from listing 13.5 to track JSP bundles from a common point vs duplicating binding logic in different bundles. To run the jsp example, in a console launch the OSGi framework via the following command:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
cd $osgi-in-action/chapter14/pax-web
ant dist
java -jar launcher.jar bundles
```

To see the shopping cart application in action browse to the following url: `http://localhost:8080/jsps/catalog.jsp` where you should see a simple shopping cart page Add a couple of items to the cart to verify it's working and you should see something like the page depicted in figure 13.3.

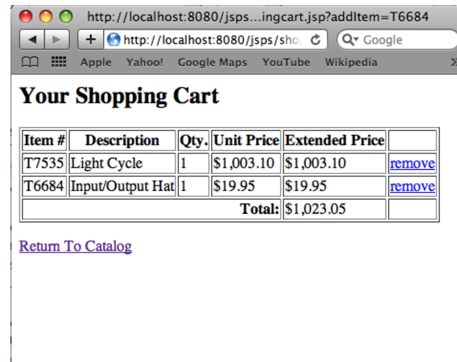


Figure 13.3 JSP shopping cart application running in an OSGi environment.

So by this point we've shown you how you can deploy a range of web application technologies from static resources to servlets to JSP's via the HttpService or it's extensions. However this may leave you wondering "Hey what happened to my WAR files?" A good question, in the next section we will look at how to package and deploy web applications that conform to the Web ARchive format in OSGi.

13.1.2 Introducing web application bundles

Over the past 10 years since the Servlet 2.2 specification came out in August 1999 we have been packaging and deploying Servlets, JSP's and other web technologies in WAR files. These provide a standard way to map a range of web service tools to a servlet container context. You could say it has taken the OSGi alliance a while to notice this, as it is only in there latest 4.2 specification that they provide the mechanism to load a WAR file in an OSGi context in a standard way.

However this would also be being unfair as the OSGi specification has been driven primarily by need, thus avoiding the curse of early specification. Up until this time there has been very little need for the complex mapping that WAR files specify as OSGi has been focussed on smaller lightweight environments for which the HttpService has been sufficient. But as OSGi has moved up the stack a need for specification has arisen to allow vendors to interoperate. Hence the addition of RFC66 (OSGi and Web Applications) which provides a standard way for Servlet and JSP application components to interoperate with OSGi services.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

So what is a web application bundle? As you might imagine a web application bundle (WAB) is pretty much just a standard WAR file that has been converted into a bundle. More specifically it is a WAR file that adheres to the Servlet 2.5 and JSP 2.1 specification that additionally declares it's dependencies via the standard OSGi classpath semantics.

In order to demonstrate the process of creating a Web Application Bundle we will take the stock watcher application from the Google Web Toolkit tutorial and convert this to run in an OSGi context.

```
<target name="osgi">
  <path id="bnd.class.path">
    <fileset dir="${root.dir}/lib" includes="osgi.*.jar"/>
    <fileset dir="build" includes="*.war"/>
  </path>
  <mkdir dir="../bundles" />
  <pathconvert pathsep=":" property="bnd.cp" refid="bnd.class.path"/>
  <bnd files="build.properties" classpath="${bnd.cp}" exceptions="true"/>
</target>
```

This calls BND to wrap the war file generated by the GWT build into a WAB file. BND in turn takes it's configuration parameters from the file build.properties in the same directory which we list here for your convenience.

```
Bundle-SymbolicName: com.google.gwt.sample.stockwatcher                  #1
Bundle-ClassPath: WEB-INF/lib/gwt-servlet.jar,WEB-INF/classes            #2
Include-Resource: war                                                    #3
Import-Package: \                                                        #4
  com.google.gwt.benchmarks;resolution:=optional,\
  junit.framework;resolution:=optional,\
  *
Web-ContextPath:  /stockwatcher/stockPrices                              #5
```

Most of these headers look pretty similar to those we've seen before (though if you need a recap you should read chapter 2 section [ref] also if you are not familiar with BND syntax you can refer to appendix [ref]. Briefly, at #1 we specify the bundle symbolic name for our WAB. At #2 we set up the bundle classpath to include the gwt-servlet.jar provided by Google and which is embedded within this bundle and the WEB-INF/classes directory which contains the classes of our application. At #3 we embed the various resources that are used by this application including JavaScript files and images. At #4 we specify that the bundle contains two optional package imports which are only used in testing scenarios – for a review of optional package imports refer to chapter [ref].

The only new header here is `Web-ContextPath` at #5. What does this do? Well the Web-ContextPath header is used to identify the bundle as a web application. The header is used by a specific class of bundle known as a *web container extender*. As the name suggests this container bundle uses the extender pattern, which have we discussed already in this book in section [ref] chapter [ref], to find bundles with the Web-ContextPath header and register the servlet resources specified in these WAB's as a running web application. The value of this header specifies the context root that the web container will use to register the web

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

application. All web accessible resources inside the bundle are then served up relative to this path.

Before we delve any further into the inner workings of WAB files, lets launch our GWT application to show it in action. To do this first compile the source:

For unix derivatives:

```
$ cd $OSGi_In_Action/chapter14/gwtapp
$ ant
```

For windows:

```
% cd %OSGi_In_Action%\chapter14\gwtapp
% ant
```

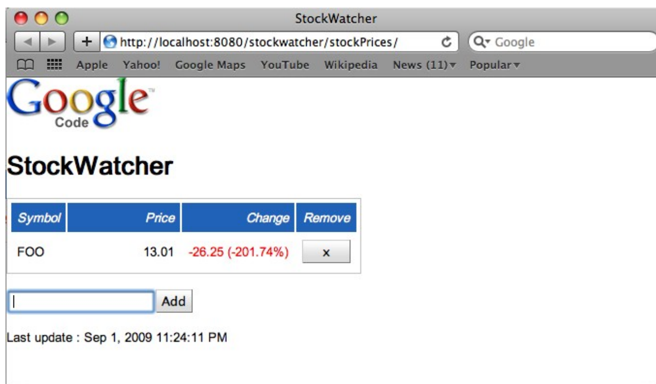


Figure 13.4 The google stock watcher application running in an OSGi context.

You can launch the GWT application using the launcher – `java -jar launcher.jar bundles/` then browse to http://localhost:8080/stockwatcher/stockPrices/ which should look something like the contents of figure 13.4.

At this point we're taking advantage of the modularity layer of OSGi to allow us to: share classes installed elsewhere in the OSGi framework; ensure that we have the correct dependencies installed; and check that we're not sharing private implementation classes with other parts of the JVM. We're also using the lifecycle layer to allow dynamic installation, update and removal of our web application.

Modularity improves memory consumption

In this trivial example the point on being able to share installed classes is of relatively minor importance as the StockWatcher has very few external dependencies. But consider the benefits of being able to share classes in a larger web application environment. In standard WAR development each application must embed it's own set of dependencies inside the WAR file under WEB-INF/lib. For utility classes such as collections tools, xml parsers, logging frameworks ,etc this can mean that a lot of classes get defined many

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

times over for each WAR file installed in your application server. This starts to chew up memory resources as the JVM must allocate a unique area of PermGen space for each applications private classpath. In an OSGi environment you can break out your dependencies to separate bundles which are then shared between your installed applications, so reducing the overall memory foot print of your application.

So modularity – check, lifecyle – check. The question that probably springs to mind is can we use the services layer? Well we have good news for you, the example you are running already *is* using services! Listing 13.10 shows how this is achieved.

Listing 13.10 Accessing the BundleContext from within a Servlet

```
public class StockPriceServiceImpl extends RemoteServiceServlet implements
StockPriceService {                                                           #1

  private BundleContext ctx;

  @Override
  public void init() throws ServletException {                                #2
    ctx = (BundleContext)
getServletContext().getAttribute(OSGiConstants.OSGI_BUNDLE_CONTEXT_ATTRIBUT
E);
  }

  @Override
  public void destroy() {
    ctx = null;                                                               #3
  }
```

Here we have extended `com.google.gwt.user.server.rpc.RemoteServiceServlet` with our own implementation class, though the details of GWT are not important for this example. The important point to note is at #2 and #3 where we override the `init` and `destroy` methods of `javax.servlet.GenericServlet`. Here we grab a reference to the osgi `BundleContext` via an attribute on the `javax.servlet.ServletContext`. You will note we have defined a Constants class for the actual attribute name. This is because at the time of writing the RFC specification is very new and we have to use the 1.2 implementation of the spring web extender, this conforms to all of the required behavior except for the trivial case of the servlet context attribute, as you can see below we have included the real value "`osgi-bundlecontext`" as per the specification in this class so that when a new version becomes available we can swap this out.

```
package com.google.gwt.sample.stockwatcher.server;

public class OSGiConstants {
  public static final String OSGI_BUNDLE_CONTEXT_ATTRIBUTE =
"org.springframework.osgi.web.org.osgi.framework.BundleContext";
  //public static final String OSGI_BUNDLE_CONTEXT_ATTRIBUTE = "osgi-
bundlecontext";
}
```

Having cached a reference to the BundleContext we can then use this to lookup other services within the framework. In this case we've added a trivial StockProvider service with the following interface:

```
package org.foo.stockprovider;
import java.util.Map;
public interface StockProvider {
  Map<String, Double> getStocks(String[] symbols);
}
```

This returns a map of stock prices for the given symbols. In the StockPriceServiceImpl class we then lookup a service in the implementation of the getPrices method as shown in listing 13.11:

Listing 13.11 Reading stock prices from the StockProvider service

```
public StockPrice[] getPrices(String[] symbols) throws DelistedException,
ServiceUnavailableException {
  StockPrice[] prices = null;

  String clazz = StockProvider.class.getName();

  ServiceReference ref = ctx.getServiceReference( clazz );              #1

  if ( ref != null ) {
    StockProvider provider = (StockProvider) ctx.getService(ref);       #2
    if ( provider != null ) {
      try {
        prices = readPrices(provider, symbols);                         #3
      }
      finally {
        ctx.ungetService(ref);                                          #4
      }
    }
  }

  if ( prices == null ) {                                               #5
    throw new ServiceUnavailableException();
  }

  return prices;
}
```

This uses the standard service lookup mechanism discussed in chapter [ref]: at #1 we see if a service is registered with the specified interface; at #2 we try to get that service; at #3 we perform some application specific calls to the StockProvider service; at #4 we tell the framework we are no longer using the service; and finally if any of these stages fail we throw an application specific `ServiceUnavailableException` at #5 to let the front end display a suitable error message to the user.

It is worth noting here that we are back to using plain OSGi service lookups in this environment. As such it would be advisable, when you are building your own applications, to consider designing in an aggregation component - if you have complex service dependencies

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

- using any of the component technologies referenced in chapter [ref]. This will allow you to focus your presentation tier on the job at hand instead of placing in a large amount of OSGi boiler plate code to handle the lifecycle issues around multiple services.

We've now looked at a range of web application technologies and shown you how they can be integrated with OSGi, the benefits of this approach include:

- enforcement of logical API boundaries, which helps avoid common pitfalls in code evolution
- improved memory consumption due to use of shared classes
- simplified collaboration between functional units due to the use of the services pattern

In the next section we will turn our attention to how we can make OSGi services available across network boundaries – i.e. how do we build distributed OSGi applications?

13.2 Web services

Up until this point in the book all of our applications have been single JVM architectures. But this is very rarely the case in web environments – in fact the entire ethos of internet development is to allow distributed processes to communicate over network protocols. We've seen how you can do this at the low level in OSGi in chapter [ref] where we built our own telnet implementation. But this is the early twenty first century and as such the zeitgeist in the room has to be Web Services. In this section we will investigate how to communicate between JVM's using OSGi technology and Web Service protocols.

Necessarily we will only touch on this area as distributed computing is very a large topic and there are many complex and subtle areas of interest which we could not hope to do justice in a single section of a single chapter. However we will show you over the next few of pages how to communicate between distributed OSGi frameworks using the new Remote Services section of the OSGi compendium specification.

Let's look at the stock watcher application we built at the end of the last section. At the moment this has a three tier architecture comprised of a web browser connected to a backend servlet engine which talks to an in-process StockProvider service. A logical step for us in this section of the book is to split the StockProvider service out onto a separate JVM and communicate with this service via an over the wire protocol such as SOAP, this new architecture is shown in figure 13.5.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

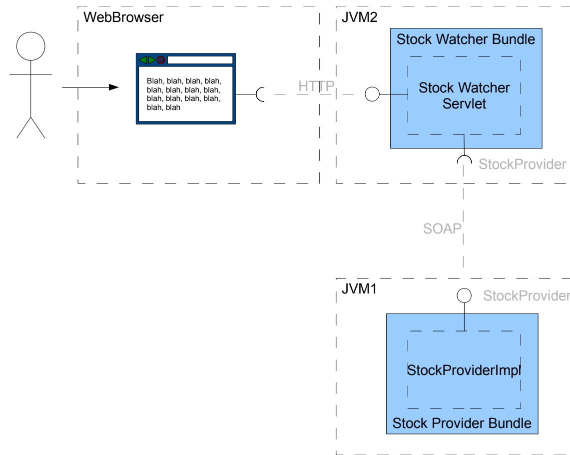


Figure 13.5 The google stock watcher application running in an OSGi context.

13.2.1 Providing a web service

The first step in creating our distributed OSGi application is to create the remote implementation of the StockProvider service. To do this we create a BundleActivator as shown in listing 13.12.

Listing 13.12 Reading stock prices from the StockProvider service

```
package org.foo.stockprovider.server;

...

public class Activator implements BundleActivator {

  public void start(BundleContext ctx) throws Exception {
    Dictionary props = new Hashtable();

    props.put("service.exported.interfaces","*");                    #1
    props.put("service.exported.intents","SOAP");                    #2
    props.put("service.exported.configs","org.apache.cxf.ws");       #3
    props.put("org.apache.cxf.ws.address","http://localhost:9090/stockprovi
der");                                                               #4

    ctx.registerService(StockProvider.class.getName(), new
StockProviderImpl(), props);                                         #5
  }

  public void stop(BundleContext ctx) throws Exception {
  }
}
```

As you can see this is a very typical OSGi BundleActivator, we register a service into the OSGi registry via the BundleContext using an interface and a set of properties. You may be asking yourself where are the remote communications in this example? Well in fact the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

distributed OSGi specification only defines a set of attributes that should be added to a service to indicate that it should be made remote, the actual remote communications are handled by another bundle or set of bundles entirely, these types of bundles are classified as distribution provider bundles.

The key attribute in listing 13.12 is in fact "`service.exported.interfaces=*`" at #1, this tells any watching distribution providers that the developer intends this service to be made available as a remote service. The value of "*" indicates that all interfaces specified in the OSGi registration should be exported remotely. This can also be changed to a String[] specifying the specific interfaces that should be exported.

OPT-IN FOR REMOTE SERVICES

This opt in mechanism is sensible, as not all services in an OSGi context make sense to be used in a remote context. Consider the white board pattern for Servlets we provided earlier in section [ref] for example, it makes very little sense to register a Java servlet interface remotely, as the java interface is entirely an in memory API.

The rest of the attributes at #2, #3 and #4 specify either intents or configuration to be used by the distribution provider when it is deciding how to publish the remote service. We'll look at intents and configuration in detail a little later, but for now you would probably guess that we're saying that the service should be exposed using a SOAP interface that will be available from the specified URL.

Let's test this theory by launching the Stock provider service in an OSGi framework along side the Apache CXF OSGi distribution provider:

```
cd $osgi-in-action/chapter14/webservice
ant clean dist
java -jar launcher.jar bundles/
```

We can then test our guess by browsing to the following url in a web browser:

```
http://localhost:9090/stockprovider?wsdl
```

You should see something like the following (truncated):

```
<wsdl:definitions name="StockProvider"
targetNamespace="http://stockprovider.foo.org/">
  <wsdl:types>
    <xsd:schema attributeFormDefault="qualified"
elementFormDefault="qualified"
targetNamespace="http://stockprovider.foo.org/">
      <xsd:complexType name="string2doubleMap">
        <xsd:sequence>
          <xsd:element maxOccurs="unbounded" minOccurs="0" name="entry">
```

So our guess was correct! By deploying our StockProvider bundle into an OSGi runtime along side a distribution provider and registering our service with some special attributes, our StockProvider service is now available to be called remotely! Pretty neat. Before we move on to the client side of our example let's look a little more at the attributes we skipped over, namely intents and configuration.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Intents and Configuration

To understand intents and configuration it is useful to consider the actual mechanics of publishing a service remotely in OSGi. In fact this is a classic white board pattern in that a service is registered using an agreed upon attribute "`service.exported.interfaces`" and it is the distribution provider's task to make that service available remotely. Given no other information a distribution provider could pick any number of ways of making this service

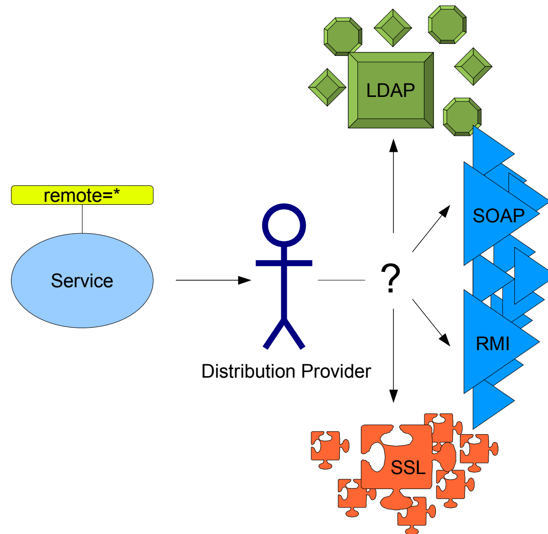


Figure 13.6 When making remote services available the number of options is bewildering, protocols, transports, authentication schemes, encryption algorithms all play their part.

available remotely, including any number of protocols, SOAP, REST, RMI, etc, a range of different security patterns encrypted or authenticated communications and a range of different transports including HTTP, JMS, etc. There is no best option. In fact the key to building well performing distributed applications is that one size most definitely does not fit all - different schemes are appropriate in different scenarios.

Having said this it makes no sense for business level services to specify the minutiae details of the mechanism via which they should be made available remotely. In fact coming full circle back to our theme from chapter [ref] this is another area where separation of concerns is applicable. Intents and configurations provide a layer of indirection between the service provider and the distribution provider. They allow the service provider to specify just enough information to ensure the service behaves as expected and allowing the distribution provider scope to optimize the communications for the environment they are deployed within. So now we know what intents and configuration are for in the abstract. let's now look at them in the concrete.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Intents

Intents are a pattern borrowed from the SCA specification, an intent is actually just a string value which two parties have agreed on the semantics of what this means at the distribution level. Huh? I hear you say, let's look at an example:

```
props.put("service.exported.intents", new String[]
{"propagatesTransaction", "authentication"});
```

In this case we're telling the distribution provider that when the service is published it must be done in such a way that transactional boundaries are transmitted to the service and that a client application will have to authenticate prior to using the service.

> **Qualified intents**
>
> Intents have a hierarchical nature which is expressed by delimiting the intent value with the '.' character. For example "authentication.transport" indicates that the service should use transport level authentication. The practical upshot of this is that a service that specifies simply "authentication" as an intent may be implemented by a provider that provides "authentication.transport". But a service that specifies "authentication.transport" may not be implemented by a provider that only provides "authentication".

Having agreed on the meaning of these intents a distribution provider can make it's best guess at the underlying implementation for these intents. This aids in decoupling distributed applications as we can specify the qualities of the remote communications without tying ourselves explicitly to a particular implementation. If you move your application to a different environment a different distribution provider may make equally valid but different guesses as to the best mechanism to use to distribute your services. We'll not go into depth on possible intent values in this book because the SCA specification defines many and just providing a list here is pretty boring.

> **Extra Intents**
>
> The osgi specification also provides the ability to configure intents via the service attribute "`service.exported.intents.extra`". This attribute is reserved for services that wish to allow external entities to manage the intents the service is published with at runtime. The recommendation is that this attribute should be configurable via the ConfigurationAdmin service. This separation allows a developer to specify a static set of intents they know about when developing the service and a systems administrator to apply separate intents at deployment time without requiring the bundle to be recompiled.

So intents allow a distribution provider to make a best guess at how to distribute an application. This is useful in that it decouples the service provider and distribution provider so allowing for a degree of flexibility when migrating software between environments. But a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

best guess may not in fact be appropriate, what if we need to give specific instructions to the distribution provider. Well this leads us onto the next level of attributes, configuration.

Configurations

Configurations provide a mechanism for the service provider to communicate explicit settings to the distribution provider. Given the range of possible configurations schemes the OSGi specification simply defines a mechanism for advertising how the configuration is encoded. From our for example earlier we saw the following attribute added to our service:

```
props.put("service.exported.configs","org.apache.cxf.ws");
```

This advertises the fact that the service should be configured using the cxf web services configuration scheme. The OSGi specification then suggests that a naming convention is used to map underlying configuration attributes <config name>.<key>. In our example earlier we specified a single configuration:

```
props.put("org.apache.cxf.ws.address","http://localhost:9090/stockprovider”
);
```

In general if you export a number of configuration schemes, you should expect to namespace the configuration elements for those schemes as follows:

```
props.put("service.exported.configs", new String[]{"foo",“bar”});
props.put("foo.key1","value1” );
props.put("foo.key2","value2” );
props.put("bar.key1","value3” );
props.put("bar.key2","value4” );
```

This concludes our overview of intents and configurations, they provide an extensible mechanism for communicating the service providers knowledge about how a service should behave in a remote environment to the distribution provider. In general intents and configurations should be kept to a minimum to allow the distribution providers flexibility in wiring together services. Let's now turn our attention to the other side of the equation, client side distributed applications.

13.2.2 Consuming a web service

Let's turn our attention back to our GWT stock watcher application. Currently this looks up a StockProvider service via the OSGi registry and uses this to retrieve stock prices. What do we need to change in order to make this work against our distributed version of the stock provider service?

Well perhaps surprisingly, nothing at all, well at least nothing in terms of our stock watcher application. The only extra step we need to do is provide our distribution provider with configuration that tells it how to discover the distributed StockProvider service, the distribution provider will then automatically create a proxy of the distributed service and inject this into the local service registry for our stock watcher application to use.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

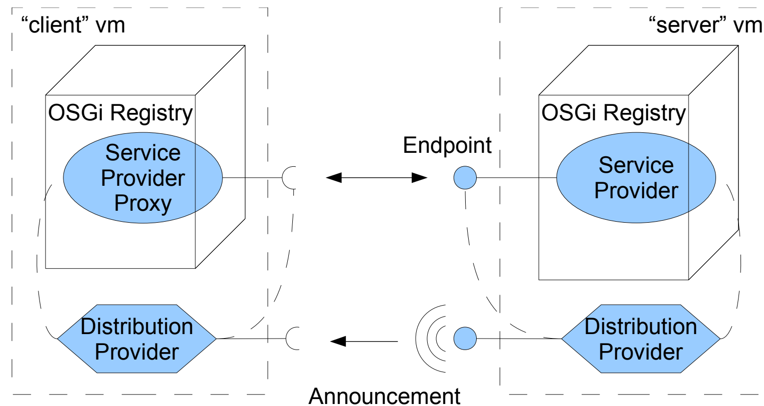


Figure 13.7 The distribution provider bundle creates a remote endpoint for the service provider and it may also announce the location and type of this endpoint for other distribution provider bundles to find. The client side distribution provider discovers remote endpoints and creates proxies to these services which it injects into the local OSGi registry.

Those who have used technologies such as Zeroconf, SSDP, UDDI, or Jini networking patterns will be familiar with the concept of discovery. Even if you are not familiar with these technologies you might well guess that discovery is a pattern used in network computing to allow a service provider to announce the presence of a service. This is often achieved via a central registry or peer to peer patterns such as TCP-multicast. These services are then "discovered" as needed by client applications and proxy services created to allow communication with these services. The distributed OSGi specification provides an extensible pattern for implementing service discovery which we'll cover in more depth in the next section.

Local discovery

The distributed OSGi specification does not define how discovery is implemented, only how it should behave. The Apache CXF dosgi implementation provides a version of discovery based on the Hadoop Zookeeper project, but the usage of this is beyond the scope of this book. However the Apache CXF project also provides a trivial version of discovery based on a local or "static" discovery file. Local discovery is provided as a mechanism to bind services in OSGi to statically defined external services – such as a web service hosted by Google or Facebook for example. The "discovery" process is managed via the existence of xml files contained within bundles that are started within the OSGi framework. Listing 13.13 shows the xml that describes our stock provider service, which is a mirror image of the service we published earlier.

Listing 13.13 Local discovery xml file for the StockProvider service

```
<service-descriptions xmlns="http://www.osgi.org/xmlns/sd/v1.0.0">
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
  <service-description>
    <provide interface="org.foo.stockprovider.StockProvider" />          #1
    <property name="service.exported.interfaces">*</property>
    <property name="service.exported.configs">                           #2
      org.apache.cxf.ws
    </property>
    <property name="org.apache.cxf.ws.address">                          #3
      http://localhost:9090/stockprovider
    </property>
  </service-description>
</service-descriptions>
```

This file is packaged in the directory `OSGI-INF/remote-service/remote-services.xml` in a new bundle `stockprovider-client-1.0.jar`. At #1 we define the interface that this service will provide at #2 and #3 we provide the configuration entries needed by the discovery provider to bind to the remote service. In fact if we look at this bundle this is the only file that is contained in this bundle.

```
cd $osgi-in-action/chapter14/webservice-client
ant clean dist
jar -tf bundles/stockprovider-client-1.0.jar
META-INF/MANIFEST.MF
OSGI-INF/
OSGI-INF/remote-service/
OSGI-INF/remote-service/remote-services.xml
```

We now know how to export and import services from remote processes into an OSGi context let's now see our updated stock watcher application in action.

Using our web service

Our stock watcher application is unmodified and we only needed to add a trivial bundle with some discovery information to enable the client side of our stock provider application!

Let's look at our updated stock watcher application in action. To launch the stock provider web service type the following into your console:

```
cd $osgi-in-action/chapter14/webservice
ant clean dist
java -Dorg.osgi.service.http.port=8081 -jar launcher.jar bundles/
```

Here we've moved the server http service onto a separate port to avoid clashes with our client application. Then launch the new stock watcher web application in another console:

```
cd $osgi-in-action/chapter14/webservice-client
ant clean dist
java -jar launcher.jar bundles/
```

Now browse to http://localhost:8080/stockwatcher/stockPrices/ and add a stock name "foo". You should see results appear in the browser and in our first console which is hosting the web application you should see the following messages:

```
Retrieved {FOO=4.736842484258008}
Retrieved {FOO=48.88924250369791}
Retrieved {FOO=22.847587831790904}
```

This indicates that the method invocation is being sent across the wire from our stock watcher jvm via our SOAP interface to the stock provider jvm.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

DEALING WITH FAILURE

One thing that should be obvious to the experienced distributed software developers (note distributed software not distributed developers) is that remote services are unreliable. In RMI we deal with this via the `java.rmi.RemoteException` which is a checked exception telling us that something went wrong when trying to communicate with the remote service. In OSGi the equivalent exception is `osgi.framework.ServiceException` which extends `RuntimeException` and hence an unchecked exception.

By now you should have a fair idea of the basic mechanics involved in using distributed services in an OSGi context. The final area we will look at in this section is the role of the distribution provider in defining the characteristics of a remote service and how a client can choose the characteristics of remote services it uses.

Exported intents and imported intents may differ

As we saw in the previous section, a service provider advertises their service in the OSGi registry with a set of intents and properties. Is this the end of the story? Well no, having handed this service off to a distribution provider the distribution provider must of course honor the requirements of the service provider but it is then free to add any other additional behaviors that they feel are appropriate, this could include default communications protocols, authentication schemes, buffering strategies, etc, etc. as shown in figure 13.8. We shall see in the next section when we look at the flip side of the equation that it is also possible for a client to specify the intents that a service must provide which will include the sum of the service providers intents and the distribution providers default intents.

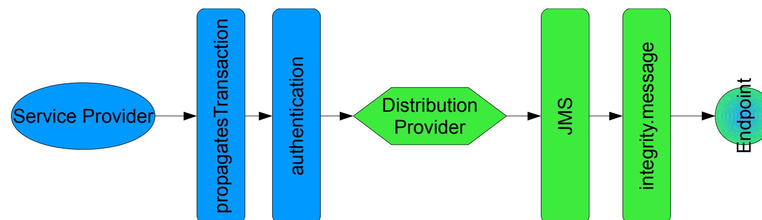


Figure 13.8 Service providers and distribution providers can each define intents that are applied to a service endpoint.

We have now seen a fully fledged web service application working in conjunction with a GWT web application to provide a distributed computing platform. Admittedly this is not a very complex environment as there is only one service and one client. What if we're working in a larger environment with multiple services and multiple clients, what facilities does the distributed OSGi specification provide to enable this more realistic architecture? Let's look at the complex issue of matching client applications to server applications.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Match-making services

We briefly covered intents and configuration in section [ref] these provide the service provider with control over how their services are exposed in a remote environment. However this problem is symmetric in that clients often need to use services with specific characteristics, for example: a medical insurance web application may require that communications with the back end servers are encrypted to ensure patient confidentiality is maintained; or a financial trading application may require a certain protocol be used to communicate between services either for performance or regulatory reasons.

Again we will only touch on this area as the number of options and hence the potential rich “feature sets” that this implies are much too large for us to cover here. But as you may have guessed clients can select the sort of services they'd like to use via filters that match the intents and configurations specified on our services at export time.

Let's first consider the simplest case – how do we differentiate between local OSGi services and remote OSGi services? Well in fact the distribution provider automatically adds a new attribute “`service.imported`” to the imported service, so if we explicitly want to bind to a remote service we can set up a filter of the following form:

```
ServiceReference ref =
context.getServiceReferences(MyService.class.getName(),"(service.imported=*
)")
```

Alternatively if we explicitly want to bind to a local service we can set the following filter:

```
ServiceReference ref =
context.getServiceReferences(MyService.class.getName(),"(!
(service.imported=*))")
```

Now let's consider the more complex case of matching remote service qualities. The service provider intents we specified earlier in section [ref] specified that the service required transactions to be propagated and for clients to be authenticated via the attribute “`service.exported.intents`”. However we also saw that a distribution provider is able to add default intents to a service when it is exported. How then do we refer to the merged set of intents with which the service is published? Well as before the distribution provider automatically adds a new attribute “`service.intents`” to the imported service. This attribute contains the union of the required intents from the service provider and the default intents supplied by the distribution provider. Therefore when the client is searching for a service it can specify the intents it is looking for via a service filter like the following:

```
ServiceReference ref =
context.getServiceReferences(MyService.class.getName(),"(&(service.intents=
propagatesTransaction)(service.intents=confidentiality))")
```

This specifies that the communications with the service must be encrypted and that transactions must be propagated.

Matching qualified intents & configurations

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

One slightly thorny area surrounds the matching of qualified intents where, for example the client requires `(service.intents=confidentiality)` but a service provides `service.intents=confidentiality.message`. In fact these two intents should match as the client doesn't care how the confidentiality is achieved, however if we think of a pure LDAP match this would necessarily fail. To work around this the OSGi specification mandates that distribution providers should expand out all implied qualified intents on services such that LDAP queries function correctly for clients. As such `service.intents=confidentiality.message` becomes `service.intents=[confidentiality,confidentiality.message]`

Another complex issue surrounds the matching of configuration values, given the range of different configuration schemes it is impossible to be prescriptive but the OSGi specification does provide a suggestion that it should be possible to perform an additional level of comparison based on the configuration meta data.

So far we have looked at how to interact with web services from the OSGi side of of the looking glass, i.e. publishing and subscribing to distributed services via the OSGi service registry. But what if you are coming at this from the other side of the looking glass? What if you are a distributed software developer and you want to publish or subscribe to remote services using your protocol of choice for injection into the OSGi registry?

13.2.3 Distributing services

In this section we will lead you through a short example that shows you how you can start to implement a trivial distribution provider framework. The goal of this is not to replace any existing frameworks but instead to show the underlying mechanics at play. We will create a simple Registry interface that abstracts away the task of publishing and subscribing to remote services. We'll see how we can publish services from the OSGi registry marked with the `service.exported.interfaces` attribute into our registry abstraction. Then we'll see how we can provide services from our registry and add them to the OSGi BundleContext. Figure 13.9 provides a view of the classes involved in this example.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

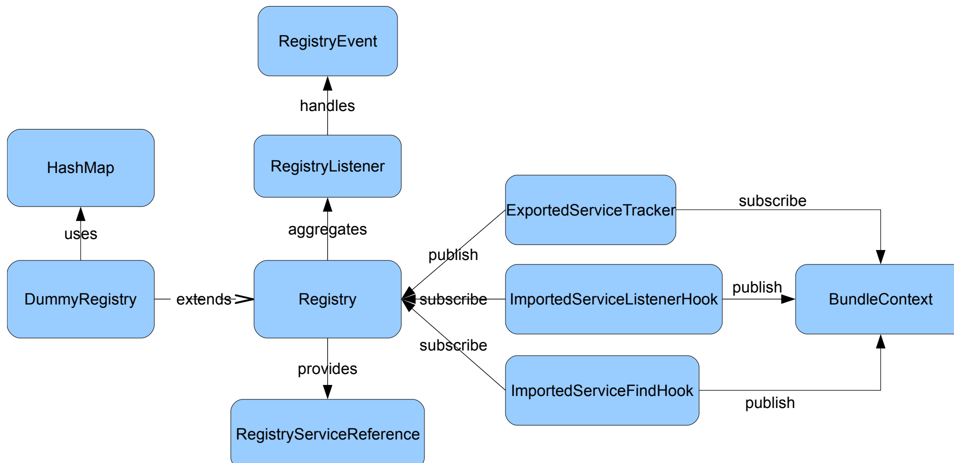


Figure 13.9 Simple registry scheme that abstracts mechanism

During this example we are going to focus our attention on the following classes:

- `ExportedServiceTracker`
- `ImportedServiceListenerHook`
- `ImportedServiceFindHook`

But before we get there let's look briefly at the other classes in this diagram, firstly the Registry API provides a simple lookup and listener scheme that mirrors the semantics of the OSGi registry but importantly is separate from it. We will implement a DummyRegistry that actually does no remote communication at all instead it just passes the services via a separate java.util.HashMap.

HASHMAP?

You might think this is a bit of a cheat to use a HashMap in this example and in fact it is. However this HashMap based approach demonstrates all the key functionality of remote services from an OSGi perspective as services are published and subscribed to via an externally managed registry. By necessity we are going to ignore the complex issues in the area of distributed computing such as:

- Network discovery protocols
- Remote procedure calls
- Object marshaling in OSGi environments

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Though these are all important topics there are a million and one ways to implement them - all of which are beyond the scope of this book. We leave you as architects or developers with the task of choosing your favorite protocols du jour should you wish to implement a real world version of this registry.

Having described the overall architecture let's start by exploring the ExportedServiceTracker which handles the task of publishing services with the `service.exported.interfaces` attribute to our registry service.

ExportedServiceTracker

The `org.foo.dosgi.hooks.ExportedServiceTracker`, class extends the `ServiceTracker` class we met in chapter [ref] as the name implies it tracks any services which have been marked as being exported, listing 13.14 shows how this is done.

Listing 13.14 Constructing an exported service tracker

```
public ExportedServiceTracker(BundleContext ctx, Registry registry,
String[] intents, String[] configs) {
  super(ctx, createFilter(ctx), null);                                  #1
  this.ctx = ctx;
  this.registry = registry;
  this.intents = intents == null ? new String[0] : intents;            #2
  this.configs = configs == null ? new String[0] : configs;            #3
}

private static Filter createFilter(BundleContext ctx) {
  try {
    return ctx.createFilter("(service.exported.interfaces=*)");        #4
  } catch (InvalidSyntaxException e) {
    throw new IllegalStateException(e);
  }
}
```

At #1 we call the ServiceTracker super constructor passing in an OSGi filter which we create at #4 to specify all services with a `service.exported.interfaces` attribute of any value. Note here the * is interpretted by the LDAP syntax to be a wild card pattern match so matches the values: *, foo, bar, [foo,bar], etc. At #2 and #3 we store the intents and configurations that this registry supports, we will see a little later how these intent and configuration values are derived, but for now let's look at the addingService method in listing 13.15:

Listing 13.15 Dealing with new exported services

```
@Override
public Object addingService(ServiceReference ref) {
  Object svc = super.addingService(ref);                               #1

  if ( isValidService(ref) ) {                                         #2
    String[] ifaces = findExportedInterfaces( ref );                   #3
    for ( String iface : ifaces ) {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
        registry.registerService(ref, iface, svc);                          #4
      }
    }

    return svc;                                                             #5
  }
```

This method is called when an exported service is published to the OSGi bundle context. The first stage is to get a reference to the service itself which we achieve at #1 by calling the addingService method of the parent ServiceTracker class, this service is used at #4 and returned at #5. We then check that the intents service matches the supported intents and configurations that were parsed in in our constructor. If the service can be published with the required intents we push the service to our Registry at #4.

NOTE

In this example we've pushed the service multiple times for each registered interface at #4, in a real world scenario it might be more appropriate to register the service once with multiple interfaces – this approach is used for simplicity only.

Listing 13.16 shows how we can find the interfaces that a service reference exports (if any). The method findExportedInteraces returns a String[] containing the interface names or null if the service is not exported.

Listing 13.16 Checking the exported interfaces of a service

```
  private String[] findExportedInterfaces(ServiceReference ref) {
    Object ifaces = ref.getProperty( "service.exported.interfaces" );       #1
    if ( ifaces == null ) {
      return null;
    }
    else {
      String[] strs = PropertyUtil.toStringArray( ifaces );                 #2
      if ( strs.length == 1 && "*".equals( strs[0] ) ) {                    #3
        ifaces = ref.getProperty( Constants.OBJECTCLASS );                  #4
        strs = PropertyUtil.toStringArray(ifaces);
      }
      return strs;
    }
  }
```

At #1 we check if the service is exported, if it is we use a utility class at #2 to return the interfaces as the value may be a String or a String[]. At #3 we check to see if the * name of the interface is "*" if it is we read the interfaces from the OSGi header `objectClass`. Finally in listing 13.17 we check to see if this exported service is supported by the registry by comparing the exported intents and configs with the intents we supplied to our ExportedServiceTracker in listing 13.14.

Listing 13.17 Checking if a service matches supported intents and configurations

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

```
private boolean isValidService(ServiceReference ref) {
  List<String> list = readIntents(ref);                                 #1
  list.removeAll( Arrays.asList(intents));                              #2
  if ( list.isEmpty() ) {                                               #2
    list = readConfigs(ref);                                            #3
    list.removeAll( Arrays.asList(configs) );                           #4
    return list.isEmpty();                                              #4
  }
  else {
    return false;
  }
}
```

We read the intent and configuration values from the service reference attributes at #1 and #3. We then use the equals method of the String class and the removeAll method of List at #2 and #4 to check that the service does not export any intents that the registry does not support.

NOTE

The isValidService reference provides a naïve implementation of the check to see if a given service matches our registry services supplied intents. It is naïve because it does not take into account the qualified naming convention we mentioned in section 13.2.1 a proper implementation would need to do this. However the logic to achieve this is too long to list here and doesn't really add much to our discussion so we'll neatly skip over it and leave it as an exercise for the reader.

The modified and remove methods of our tracker are broadly similar, so let's turn our attention to the client side of the equation. How does a client find out about services in OSGi? As we've seen we can either do a direct lookup via the BundleContext.getServiceReference call or we can register a ServiceListener to be notified when services appear. In the latest OSGi specification two new service interfaces have been added to the OSGi specification which help us in this regard:

- org.osgi.framework.hooks.service.FindHook
- org.osgi.framework.hooks.service.ListenerHook

Services published with these interfaces have special powers within the OSGi framework, they are called by the framework itself when other bundles: perform find calls via getServiceReference ;or register or unregister listeners via addServiceListener and removeServiceListener respectively. In order to save ourselves repetition of some boiler plate code in the following examples we define a RegistryWatcher helper class to handle the lookup of services from our Registry and injection into the OSGi context. Listing [ref] shows the implementation of the addWatch method which we provide to give context to the following examples:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

Listing 13.18 Registry watcher helper addWatch method

```
public void addWatch(String clazz, String filter) {
  synchronized (watches) {
    if (watches.add(new Watch(clazz, filter))) {                       #1
      Collection<RegistryServiceReference> services = registry
          .findServices(clazz, filter);                                #2
      for (RegistryServiceReference ref : services) {
        if (!regs.containsKey(ref)) {                                  #3
          Future<ServiceRegistration> future = exec
              .submit(new Registration(ref));                          #4
          regs.put(ref, future);
        }
      }
    }
  }
}
```

Using our helper class and these new interfaces we can now find out when a remote service is needed and inject it into the local OSGi registry on demand. Let's see how this is done.

ImportedServiceListenerHook

Let's look at our imported service listener in listing 13.19. This class tracks listener registrations and adds a watch in our Registry interface for services with the specified interface:

Listing 13.19 Listeners added to osgi context can be tracked via the ListenerHook

```
public void added(Collection listeners) {
  for (final ListenerInfo info : (Collection<ListenerInfo>) listeners) {
    if (!info.isRemoved()) {                                           #1
      LDAPExpr expr = LDAPParser.parseExpression(info.getFilter());    #2
      expr.visit(new ExprVisitor() {
        public void visitExpr(LDAPExpr expr) {
          if (expr instanceof SimpleTerm) {                            #3
            SimpleTerm term = (SimpleTerm) expr;
            if (term.getName().equals(Constants.OBJECTCLASS)) {
              watcher.addWatch(term.getRval(), info.getFilter());      #4
            }
          }
        }
      });
    }
  }
}
```

At #1 we check if the listener is removed, this may seem a little odd given it happens in the added method but this protects our listener against a race condition that can occur due to asynchronous event delivery. We then inspect the body of the ldap expression using a utility class to walk our way through the filter expression at #2 and #3. Finally at #4 we add a watch for the service interface specified by the OSGi Constants.OBJECTCLASS attribute to the Registry. The result is that when a bundle registers a listener for a class the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

ImportedServiceListenerHook will find any existing services in the Registry and add them to the local OSGi context it will also watch for new services and register them when they appear. This covers the asynchronous lookup scenario, how about direct lookups? Let's look at this next.

ImportedServiceFindHook

When a service calls BundleContext.getServiceReference() we'd like to be able to intercept this and inject a remote service into the OSGi registry. This is exactly what our ImportedServiceFindHook achieves:

```
public class ImportedServiceFindHook implements FindHook {
…
  public void find(BundleContext ctx, java.lang.String name,
      java.lang.String filter, boolean allServices, Collection references)
  {
    watcher.findServices(name, filter);
  }
}
```

In fact our implementation is trivial as it simply requests our registry watcher to find any existing services in the Registry, which then adds the services to the local OSGi context.

Registry Services

We're almost there now the last area we need to look at is the relationship between a distribution provider and a bundle that uses a specific set of intents or configurations. In fact a bundle that uses intents and configurations is defining an implicit dependency on a provider that can satisfy those intents and configurations. In order to make this relationship explicit the OSGi specification states that our Registry service implementation should add the following attributes to their services:

- remote.intents.supported – (String+) The vocabulary of the given distribution provider.
- remote.configs.supported – (String+) The configuration types that are implemented by the distribution provider.

Thus a bundle that depends on the availability of specific intents or configuration types can create a service dependency on an anonymous service with the given properties. The following filter is an example of depending on a hypothetical org.json configuration type:

(remote.configs.supported=org.json)

Putting it all together

We'll skip over the implementation of the DummyRegistry here as it is indeed trivial but the curious can take a look in XXX. We can now complete the picture by creating a test bundle that exports a trivial Foo service using the service.exported.interfaces=* attribute as shown below:

```
Hashtable props = new Hashtable();
props.put("service.exported.interfaces","*");

context.registerService(Foo.class.getName(), new FooImpl(), props);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

And then add a service tracker that finds our imported service in listing 13.20, here we explicitly look for the service.imported=* header to ensure that we find the "remote" version of our service:

Listing 13.20 Tracking our "remote" service

```
Filter filter = context.createFilter("(&(" + Constants.OBJECTCLASS + "=" +
Foo.class.getName() + ")(service.imported=*))");
ServiceTracker tracker = new ServiceTracker(context, filter, null) {
  @Override
  public Object addingService(ServiceReference reference) {
    System.out.println( "Found " + reference + " !!!!!!!" );
    return super.addingService(reference);
  }

  @Override
   public void removedService(ServiceReference reference, Object service) {
    System.out.println( "Lost " + reference + " !!!!!!!" );
    super.removedService(reference, service);
  }
};
tracker.open();
```

To test this all fits together we can then boot the framework using the following commands:

cd $osgi-in-action/chapter14/webservice-impl

ant clean dist

java -jar launcher.jar bundles

Found [org.foo.dosgi.test.Foo] !!!!!!!

This registry is intended as an example only – as we stated at the start of this section there are many more issues that need to be taken into account when building real world distributed OSGi infrastructure. However it demonstrates that the underlying mechanics are easily implemented in an OSGi setting and can work seamlessly with existing OSGi applications.

13.3 Summary

So that's it, we've reached the crest of the hill, during this chapter we have shown you how to build web applications that take advantage of the OSGi framework. This included using all aspects of the OSGi framework to handle:

- Static resources
- Servlet and JSP based applications
- A simple stock watcher application using Google Web Toolkit

Having looked at the client tier web technologies we then turned our attention to the next level in the stack namely web services and how to use them in an OSGi context. Here we:

- Showed you how to migrate a local OSGi service via an OSGi distribution provider onto

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507

a remote JVM

- Highlighted the extreme flexibility of service based programming as our client application was unchanged during this process

Finally to give you full context we lead you though a brief tour of the behind the scenes mechanics of an OSGi distribution provider. This introduced you to some advanced features of the OSGi framework and demonstrated the infinite extensibility capabilities of the OSGi framework.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=507