Beginning Java Objects

From Concepts to Code

Third Edition

Jacquie Barker

Apress®

Beginning Java Objects

From Concepts to Code

Third Edition

Jacquie Barker

Apress®

Beginning Java Objects: From Concepts to Code

Jacquie Barker
Fairfax, VA, USA

ISBN-13 (pbk): 978-1-4842-9059-0 ISBN-13 (electronic): 978-1-4842-9060-6
https://doi.org/10.1007/978-1-4842-9060-6

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: Laura Berendson
Coordinating Editor: Gryffin Winkler

Cover image designed by Pixabay (www.pixabay.com)

Distributed to the book trade worldwide by Springer Science+Business Media LLC, 1 New York Plaza, Suite 4600, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at https://github.com/Apress/Beginning-Java-Objects-3e.

Printed on acid-free paper

To my husband, best friend, and love of my life, Steve,
who supports me each and every day in more ways
than he can possibly imagine.

Table of Contents

About the Author

Jacquie Barker is a professional software engineer, author, and former adjunct faculty member at both George Mason University in Fairfax, VA, and the George Washington University in Washington, DC. With over 30 years of experience as a software developer and project manager, Jacquie has spent the past 15 years focused on object technology and is proficient as an object modeler and Sun Microsystems Certified Java programmer.

Jacquie earned a Bachelor of Science degree in computer engineering with highest honors from the Case Institute of Technology/Case Western Reserve University in Cleveland, Ohio, and a Master of Science degree in computer science, emphasizing software systems engineering, from the University of California, Los Angeles. She has subsequently pursued postgraduate studies in information technology at George Mason University in Fairfax, VA. Jacquie's winning formula for teaching object fundamentals continues to receive praise from readers around the world, and *Beginning Java Objects: From Concepts to Code* has been adopted by many universities as a key textbook in their core IT curricula.

On a personal note, Jacquie's passions include her husband, Steve, and their three cats, Walter, Rocky, and Tanner; serving as founder and executive director of Pets Bring Joy, a 501(c)(3) nonprofit animal rescue organization (visit pbj.org); and her recent launch of a pro bono IT consulting service for start-up nonprofits (visit probonoit.org). Jacquie can be reached at jacquie.barker@gmail.com.

About the Technical Reviewer

Manuel Jordan is an autodidactic developer and researcher who enjoys learning new technologies for his own experiments about creating new integrations among them.

Manuel won the 2010 Springy Award–Community Champion and Spring Champion 2013. In his little free time, he reads the Bible and composes music on his bass and guitar.

Manuel considers that constant education and training is very valuable for any developer, the same being true about refactoring and testing. Manuel can offer these services for your company with his background on Java and Spring.

You can reach him mostly through his Twitter account @dr_pompeii.

Preface

Welcome to the third edition of *Beginning Java Objects* (BJO)! Since the first edition of BJO was published back in November 2000 and the second edition in 2005, I've been delighted by the many emails and positive reviews that I've received from readers who found my book to be a perfect "jump start" into Java and object-oriented (OO) programming. So many beginning Java books dive into a discussion of the language itself without properly grounding readers in how to "think" and structure an application from the ground up to take full advantage of object-oriented principles; I'm delighted that you've chosen BJO to get started on your Java journey.

My book is based on timeless principles that are language version independent, which means that it needn't be revised every time a new version of Java is released. What do change, sometimes seemingly in the blink of an eye, are third-party technologies used in conjunction with the core Java language, and so we've replaced material that went into detail regarding outdated technologies with a single chapter (Chapter 15) that explains *conceptually* how to move forward with building an application that achieves proper model–data layer–presentation layer separation.

We've also included mention of some of the significant enhancements to the Java language as of versions 8 through 17 (the newest version due out as of the time of writing of the third edition).

As always, I welcome reader feedback and hope to hear from you at jacquie.barker@gmail.com.

Best regards,

Jacquie

Introduction

This is a book, first and foremost, about software objects: what they are, why they are so "magical" and yet so straightforward, and how one goes about structuring a software application to use objects appropriately.

This is also a book about Java: not a hard-core, "everything there is to know about Java" book, but rather a gentle yet comprehensive introduction to the language, with special emphasis on how to transition from an object model to a working Java application—something that few, if any, other books provide.

Goals for This Book

My goals in writing this book (and, hopefully, yours for buying it) are to

- ***Make you comfortable with fundamental object-oriented (OO) terminology and concepts.***
- ***Give you hands-on, practical experience with object modeling***, that is, with developing an object-oriented "blueprint" that can be used as the basis for subsequently building an object-oriented software system.
- ***Illustrate the basics of how such an object model is translated into a working software application—a Java application, to be specific,*** although the techniques that you'll learn for object modeling apply equally well to any OO language.
- ***Help you become proficient as a Java programmer along the way.***

If you're already experienced with the Java language (but not with object fundamentals), it's critical to your successful use of the language that you learn about its object-oriented roots. On the other hand, if you're a newcomer to Java, then this book will get you properly "jump-started." ***Either way, this book is a "must-read" for anyone who wishes to become proficient with an OO programming language like Java.***

Just as importantly, this book is ***not*** meant to

- ***Turn you into an overnight pro in object modeling:*** Like all advanced skills, becoming totally comfortable with object modeling takes two things: a good theoretical foundation and a lot of practice! I give you the foundation in this book, including an introduction to the Unified Modeling Language (UML), the industry standard for rendering an object-oriented "blueprint" of a software application. (UML was first adopted as a standard for modeling objected-oriented software systems in 1997 and is still relevant today.) That being said, the only way you'll really get to be proficient with object modeling is by participating in OO modeling and development projects over time.

  My book will give you the skills, and hopefully the ***confidence***, to begin to apply object techniques in a professional setting, which is where your real learning will take place, particularly if you have an OO-experienced mentor to guide you through your first "industrial-strength" project.

- ***Teach you "everything" you'll ever need to know about Java:*** Java is a very rich language, consisting of hundreds of core classes and literally thousands of operations that can be performed with and by these classes. Also, new versions of the Java language are released by Oracle Corporation every year or so, but the good news is that key Java features needed to represent a software problem in a proper object-oriented way have not changed over the years. If Java provides a dozen alternative ways to do something in particular, I explain the one or two ways that I feel best suit the problem at hand, to give you an appreciation for how things are done. Nonetheless, you'll definitely see enough of the Java language in this book to prepare you for a role as a professional Java programmer.

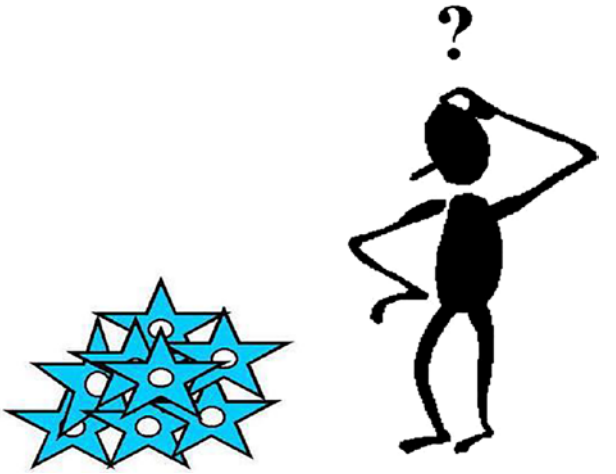
Armed with the foundation you gain from this book, you'll be poised and ready to appreciate a more thorough treatment of Java such as that offered by one of the many other Java references that are presently on the market or a deeper review of object modeling techniques from an in-depth UML reference. We'll make recommendations for such books in Chapter 15.

Why Is Understanding Objects So Critical to Being a Successful OO Programmer?

Time and again, I meet software developers—at my place of employment, at clients' offices, at professional conferences, on college campuses—who have attempted to master an object-oriented programming language (OOPL) like Java by taking a course in Java, reading a book about Java, or installing and using a Java integrated development environment (IDE) such as Eclipse, IntelliJ IDEA, NetBeans, or BlueJ. However, there is something fundamentally missing from virtually all of these approaches: a basic understanding of what objects are all about and, more importantly, ***knowledge of how to structure a software application from the ground up to make the most of objects***.

Imagine that you've been asked to build a house and that you know the basics of home construction. In fact, you're a world-renowned home builder whose services are in high demand! You've built homes of every possible architectural style, using every known type of building material: brick, lumber, stone, metal, etc. So, when your client tells you that they want you to use a brand-new type of construction material, which they'll provide, you're happy to oblige.

On the day construction is to begin, a truck pulls up at the building site and unloads a large pile of odd-looking blue star-shaped blocks with holes in the middle. You're totally baffled! You've built countless homes using more familiar materials, but you haven't got a ***clue*** about how to assemble a house using blue stars.



Scratching your head, you pull out a hammer and some nails and try to nail the blue stars together as if you were working with lumber, but the stars don't fit together very well. You then try to fill in the gaps with the same mortar that you would use to adhere bricks to one another, but the mortar doesn't stick to these blue stars very well. Because you're working under tight cost and schedule constraints for building this home for your client, however (and because you're too embarrassed to admit that you, as an ***expert*** home builder, don't know how to work with these modern materials), you press on. Eventually, you wind up with something that looks (on the surface, at least) like a house.

Your client comes to inspect the work and is terribly disappointed. One of the reasons they had selected blue stars as a construction material was that they are extremely energy efficient; but, because you have used nails and mortar to assemble the stars, they have lost a great deal of their inherent ability to insulate the home.

To compensate, your client asks you to replace all of the windows in the home with triple-pane thermal glass windows so that they will allow less heat to escape. ***You're panicking at this point!*** Swapping out the windows will require you to literally rip the walls apart, destroying the house.

When you tell your customer this, they go ***ballistic***! Another reason that they selected blue stars as a construction material was because of their modularity and hence ease of accommodating design changes; but, because of the ineffective way with which you've assembled these stars, they've lost this flexibility, as well.

This is, sad to say, the way many programmers wind up building an OO application when they don't have appropriate training in the fundamental properties of the building blocks of such an application, namely, software objects. Worse yet, the vast majority of would-be OO programmers are blissfully ignorant of the need to understand objects in order to program in an OO language. So they take off programming with a language like Java and wind up with a far from ideal result: an application that lacks flexibility when an inevitable "midcourse correction" occurs in response to changing requirements after the application has been deployed.

Whom Is This Book Written For?

Anyone who wants to get the most out of an object-oriented programming language like Java! It has been written for

- Anyone who has yet to tackle Java, but wants to get off on the right foot with the language
- Anyone who has ever purchased a book on Java and has read it faithfully, who understands the "bits and bytes" of the language, but doesn't quite know how to structure an application to best take advantage of Java's object-oriented features

- Anyone who has built a Java application, but was disappointed with how difficult it was to maintain or modify it when new requirements were presented later in the application's life cycle
- Anyone who has previously learned something about object modeling, but is "fuzzy" on how to transition from an object model to real, live code (Java or otherwise)

The bottom line is that ***anyone who really wants to master an OO language like Java must become an expert in objects FIRST!***

In order to gain the most value from this book, you should have some programming experience under your belt; virtually any language will do. You should understand simple programming concepts such as

- Simple data types (integer, floating point, etc.)
- Variables and their scope (including the notion of global data)
- Flow control ("`if` … `then` … `else`" statements, `for`/do/while loops, etc.)
- What arrays are and how to use them
- The notion of a software function/subroutine/procedure: how to pass data in and get results back out

But you needn't have had any prior exposure to Java. And you needn't have ever been exposed to objects, either—in the software sense, at least! As you'll learn in Chapter 2, human beings naturally view the entire world from the perspective of objects.

Even if you've already developed a full-fledged Java application, it's certainly not too late to read this book if you still feel fuzzy when it comes to the object aspects of structuring an application; it ultimately makes someone a better Java programmer to know the "whys" of object orientation rather than merely the mechanics of the language. You'll most likely see some familiar landmarks (in the form of Java code examples) in this book, but will hopefully gain many new insights as you learn the rationale for why we do many of the things that we do when programming in Java (or any other OO programming language for that matter).

Because this book has its roots in courses that I teach at the university level, it's ideally suited for use as a textbook for a semester-long university or advanced placement high school course in either object modeling or Java programming.

What If You're Interested in Object Modeling, but Not Necessarily in Java Programming?

Will my book still be of value to you? ***Definitely!*** Even if you don't plan on making a career of programming (as is true of many of my object modeling students), I've found that being exposed to code examples written in an OO language like Java really helps to cement object concepts. So you're encouraged to read Parts 1 and 3 even if you never intend to set your hands to the keyboard for purposes of Java programming afterward.

How This Book Is Organized

The book is structured around three major topics, as follows.

Part 1: The ABCs of Objects

Before I dive into the how-to's of object modeling and the details of OO programming in Java, it's important that we all speak the same language with respect to objects. Part 1, consisting of Chapters 1–7, starts out slowly, by defining basic concepts that underlie all software development approaches, OO or otherwise. But the chapters quickly ramp up to a discussion of advanced object concepts so that, by the end of Part 1, you will be "object savvy."

Part 2: Object Modeling 101

In Part 2—Chapters 8–12 of the book—I focus on the underlying principles of how and, more importantly, ***why*** we do the things that we do when we develop an object model of an application—principles that are common to all object modeling techniques. It's important to be conversant in the UML, and so I teach you the basics of the UML and use it for all of the concrete modeling examples in my book. Using the modeling techniques presented in these chapters, we'll develop an object model "blueprint" for a Student Registration System (SRS), the requirements specification for which is presented at the end of this introduction.

Part 3: Translating an Object "Blueprint" into Java Code

In Part 3 of the book—Chapters 13–15—I illustrate how to render the SRS object model that we've developed in Part 2 into a fully functioning Java application, to serve as the **model layer** in a three-tier application, and we'll also talk conceptually about how third-party technologies can be harnessed to provide both a user interface (known as the **presentation layer or tier**) and data persistence (known as the **data layer or tier**). The SRS source code is available for download from GitHub (`github.com/Apress/Beginning-Java-Objects-3e`), and I strongly encourage you to download and experiment with this code when you reach that chapter.

The requirements specification for the SRS is written in the narrative style with which software system requirements are often expressed. You may feel confident that you could build an application today to solve this problem, but by the end of my book, you should feel much more confident in your ability to build it as an ***object-oriented*** application. Three additional case studies—for a Prescription Tracking System (PTS), a Conference Room Reservation System, and an Airline Reservation System, respectively—are presented in the Appendix; these serve as the basis for many of the exercises presented at the end of each chapter.

Suggestions have been provided in the final chapter for how you might wish to continue your object-oriented discovery process after finishing my book. In that chapter, I furnish you with a list of recommended books that will take you to the next level of proficiency, depending on what your intention is for applying what you've learned in my book.

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

For instance:

Notes are shown in this fashion and reflect important background information.

As for styles in the text:

- When we introduce important words, we **bold** them to highlight them.
- We show file names, URLs, and code within the text like so: `objectstart.com`.
- We **`bold`** lines of code within long code passages if we want to call your attention to those lines in particular:

```
// Bolding is used to call attention to new or significant code:
Student s = new Student();
// whereas unbolded code is code that's less important in the
// present context, or has been seen before.
int x = 3;
```

- We use *`italic`* vs. `regular code font` to represent pseudocode:

```
// This is real code:
for (int i = 0; i <= 10; i++) {
    // This is pseudocode!
    compute the grade for the ith Student
}
```

Which Version of Java Is This Book Based On?

As mentioned previously, Oracle Corporation continues to release new versions of the Java language on a regular basis. The ***good news*** is that, because I focus only on core Java language syntax in my book—language features that have been stable since Java's inception—this book isn't version specific. The techniques and concepts that you'll learn by reading my book will serve you equally well when new versions of Java appear. That being said, all of the code examples in BJO are compatible with Java versions 8 through 17. (Generally speaking, new versions of the Java language are forward-compatible, meaning that code written for an older version of the language will compile properly in a newer version.)

A Final Thought Before We Get Started

A lot of the material in my book—particularly at the beginning of Part 1—may seem overly simplistic to experienced programmers. This is because much of object technology is founded on basic software engineering principles that have been in practice for many years and, in many cases, just repackaged slightly differently. There are indeed a few new tricks that make OO languages extremely powerful and that were virtually impossible to achieve with non-OO languages—**inheritance** and **polymorphism,** for example, which you'll learn more about in Chapters 5 and 7, respectively. (Such techniques can be simulated by hand in a non-OO language, just as programmers could program their own database management system (DBMS) from scratch instead of using a commercial product like Oracle or SQL Server—but who'd ***want*** to?)

The biggest challenge for ***experienced*** programmers in becoming proficient with objects is in reorienting the manner in which they think about the problem they will be automating:

- Software engineers/programmers who have developed applications using non–object-oriented methods often have to "unlearn" certain approaches used in the traditional methods of software analysis and design.
- Paradoxically, people just starting out as programmers (or as OO modelers) sometimes have an easier time when learning the OO approach to software development as their ***only*** approach.

Fortunately, the way we need to think about objects when developing software turns out to be the natural way that people think about the world in general. So learning to "think" objects—and to program them in Java—is as easy as (Part) 1, (Part) 2, (Part) 3!

Source code for this book can be found at `https://github.com/Apress/Beginning-Java-Objects-3rd-ed`.

Student Registration System (SRS) Case Study

STUDENT REGISTRATION SYSTEM (SRS) REQUIREMENTS SPECIFICATION

We have been asked to develop an automated Student Registration System (SRS). This system will enable students to register online for courses each semester, as well as tracking a student's progress toward completion of their degree.

When a student first enrolls at the university, they use the SRS to set forth a plan of study as to which courses they plan on taking to satisfy a particular degree program and choose a faculty advisor. The SRS will verify whether or not the proposed plan of study satisfies the requirements of the degree that the student is seeking. Once a plan of study has been established, then, during the registration period preceding each semester, students are able to view the schedule of classes online and choose whichever classes they wish to attend, indicating the preferred section (day of the week and time of day) if the class is offered by more than one professor. The SRS will verify whether or not the student has satisfied the necessary prerequisites for each requested course by referring to the student's online transcript of courses completed and grades received (the student may review their transcript online at any time).

Assuming that (a) the prerequisites for the requested course(s) are satisfied, (b) the course(s) meets one of the student's plan of study requirements, and (c) there is room available in each of the class(es), the student is enrolled in the class(es).

If (a) and (b) are satisfied, but (c) is not, the student is placed on a first-come, first-served waiting list. If a class/section that they were previously waitlisted for becomes available (either because some other student has dropped the class or because the seating capacity for the class has been increased), the student is automatically enrolled in the waitlisted class, and an email message to that effect is sent to the student. It is their responsibility to drop the class if it is no longer desired; otherwise, they will be billed for the course.

CHAPTER 1

Abstraction and Modeling

As human beings, we're flooded with information every day of our lives. Even if we could temporarily turn off all of the sources of "e-information" that are constantly bombarding us—emails, voicemails, podcasts, tweets, and the like—our five senses alone collect millions of bits of information per day just from our surroundings. Yet, we manage to make sense out of all of this information, typically without getting overwhelmed. Our brains naturally simplify the details of all that we observe so that these details are manageable through a process known as **abstraction**.

In this chapter, you'll learn

- How abstraction serves to simplify our view of the world
- How we organize our knowledge hierarchically to minimize the amount of information that we have to mentally juggle at any given time
- The relevance of abstraction to software development
- The inherent challenges that we face as software developers when attempting to model a real-word situation in software

Simplification Through Abstraction

Take a moment to look around the room in which you're reading this book. At first, you may think that there really aren't that many things to observe: some furniture, light fixtures, perhaps some plants, artwork, even some other people or pets. Maybe there is a window to gaze out of that opens up the outside world to observation.

Now look again. For each thing that you see, there are a myriad of details to observe: its size, its color, its intended purpose, the components from which it's assembled (the legs on a table, the lightbulbs in a lamp), etc. In addition, each one of these components

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_1

in turn has details associated with it: the type of material used to make the legs of the table (wood or metal), the wattage of the lightbulbs, etc. Now factor in your other senses: the sound of someone snoring (hopefully not while reading this book!), the smell of popcorn coming from the microwave oven down the hall, and so forth. Finally, think about all of the unseen details of these objects: who manufactured them or what their chemical, molecular, or genetic composition is.

It's clear that the amount of information to be processed by our brains is truly overwhelming. For the vast majority of people, this doesn't pose a problem, however, because we're innately skilled at **abstraction**, a process that involves recognizing and focusing on the important characteristics of a situation or object and filtering out or ignoring all of the unessential details.

One familiar example of an abstraction is a road map. As an abstraction, a road map represents those features of a given geographic area relevant to someone trying to navigate with the map, perhaps by car: major roads and places of interest, obstacles such as large bodies of water, etc. Of necessity, a road map can't include every building, tree, street sign, billboard, traffic light, fast-food restaurant, etc. that physically exists in the real world. If it did, then it would be so cluttered as to be virtually unusable; none of the important features would stand out. Compare a road map to a topographical map, a climatological map, and a population density map of the same region: each abstracts out different features of the real world—namely, those relevant to the intended user of the map in question.

As another example, consider a landscape. An artist may look at the landscape from the perspective of colors, textures, and shapes as a prospective subject for a painting. A home builder may look at the same landscape from the perspective of where the best building site may be on the property, assessing how many trees will need to be cleared to make way for a construction project. An ecologist may closely study the individual species of trees and other plant/animal life for their biodiversity, with an eye toward preserving and protecting them, whereas a child may simply be looking at all of the trees in search of the best site for a tree house. Some elements are common to all four observers' abstractions of the landscape—the types, sizes, and locations of trees, for example—while others aren't relevant to all of the abstractions.

Generalization Through Abstraction

If we eliminate enough detail from an abstraction, it becomes generic enough to apply to a wide range of specific situations or instances. Such generic abstractions can often be quite useful. For example, a diagram of a generic cell in the human body, such as the one in Figure 1-1, might include only a few features of the structures that are found in an actual cell.

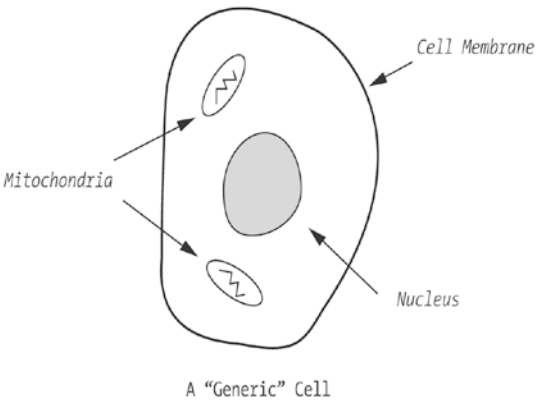


Figure 1-1. *A generic abstraction of a cell*

This overly simplified diagram doesn't look like a real nerve cell or a real muscle cell or a real blood cell; and yet, it can still be used in an educational setting to describe certain aspects of the structure and function of all of these cell types—namely, those features that the various cell types have in common.

The simpler an abstraction—that is, the fewer features it presents—the more general it is, and the more versatile it is in describing a variety of real-world situations. The more complex an abstraction, the more restrictive it is, and thus the fewer situations it is useful in describing.

Organizing Abstractions into Classification Hierarchies

Even though our brains are adept at abstracting concepts such as road maps and landscapes, that still leaves us with millions of separate abstractions to deal with over our lifetimes. To cope with this aspect of complexity, human beings systematically arrange information into categories according to established criteria; this process is known as **classification**.

For example, science categorizes all natural objects as belonging to either the animal, plant, or mineral kingdom. In order for a natural object to be classified as an animal, it must satisfy the following rules:

- It must be (or have at one time been) a living being.
- It must be capable of spontaneous movement.
- It must be capable of rapid motor response to stimulation.

The rules for what constitutes a plant, on the other hand, are different:

- It must be a living being (same as for an animal).
- It must lack an obvious nervous system.
- It must possess cellulose cell walls.

Given clear-cut rules such as these, placing an object into the appropriate category, or **class**, is rather straightforward. We can then "drill down," specifying additional rules that differentiate various types of animal, for example, until we've built up a hierarchy of increasingly more complex abstractions from top to bottom. A simple example of such an **abstraction hierarchy** is shown in Figure 1-2.

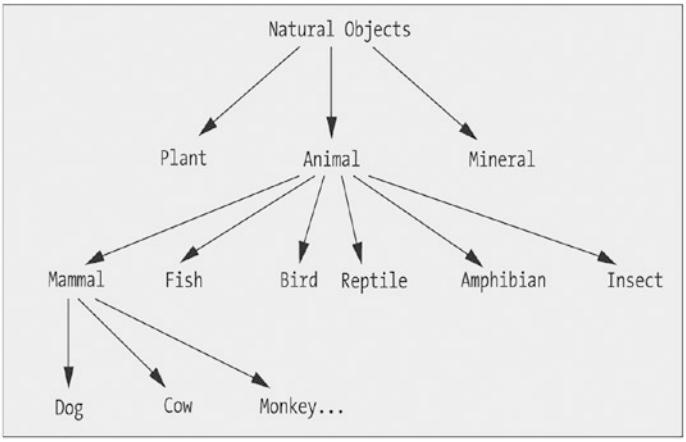


Figure 1-2. *A simple abstraction hierarchy of natural objects*

When thinking about an abstraction hierarchy such as the one shown in Figure 1-2, we mentally step up and down the hierarchy, automatically zeroing in on only the single layer or subset of the hierarchy (known as a **subtree**) that is important to us at a given point in time. For example, we may only be concerned with mammals and so can focus on the mammalian subtree, shown in Figure 1-3, temporarily ignoring the rest of the hierarchy.

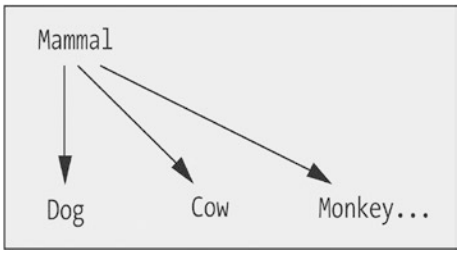


Figure 1-3. *Focusing on a small subset of the hierarchy is less overwhelming*

By doing so, we automatically reduce the number of concepts that we mentally need to juggle at any one time to a manageable subset of the overall abstraction hierarchy; in our simplistic example, we're now dealing with only four concepts rather than the original 13. No matter how complex an abstraction hierarchy grows to be, it needn't overwhelm us if it's properly organized.

Coming up with precisely which rules are necessary to properly classify an object within an abstraction hierarchy isn't always easy. Take, for example, the rules we might define for what constitutes a bird: namely, something that

- Has feathers
- Has wings
- Lays eggs
- Is capable of flying

Given these rules, neither an ostrich nor a penguin could be classified as a bird (although both should be), because neither can fly (see Figure 1-4).

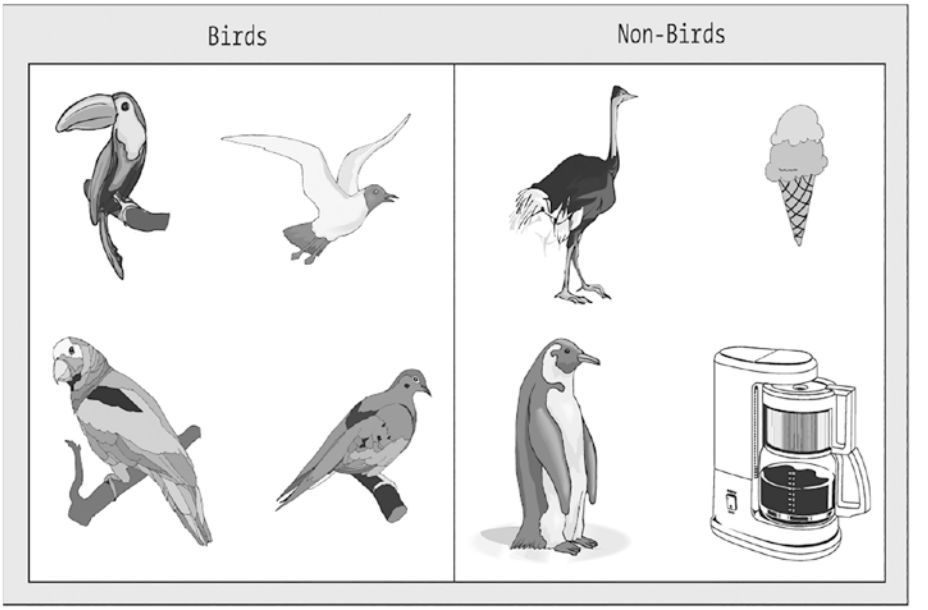


Figure 1-4. *Deriving the correct classification rules can be difficult*

If we attempt to make the rule set less restrictive by eliminating the "is capable of flying" rule, we're left with

- Has feathers
- Has wings
- Lays eggs

According to this rule set, we now may properly classify both the ostrich and the penguin as birds, as shown in Figure 1-5.

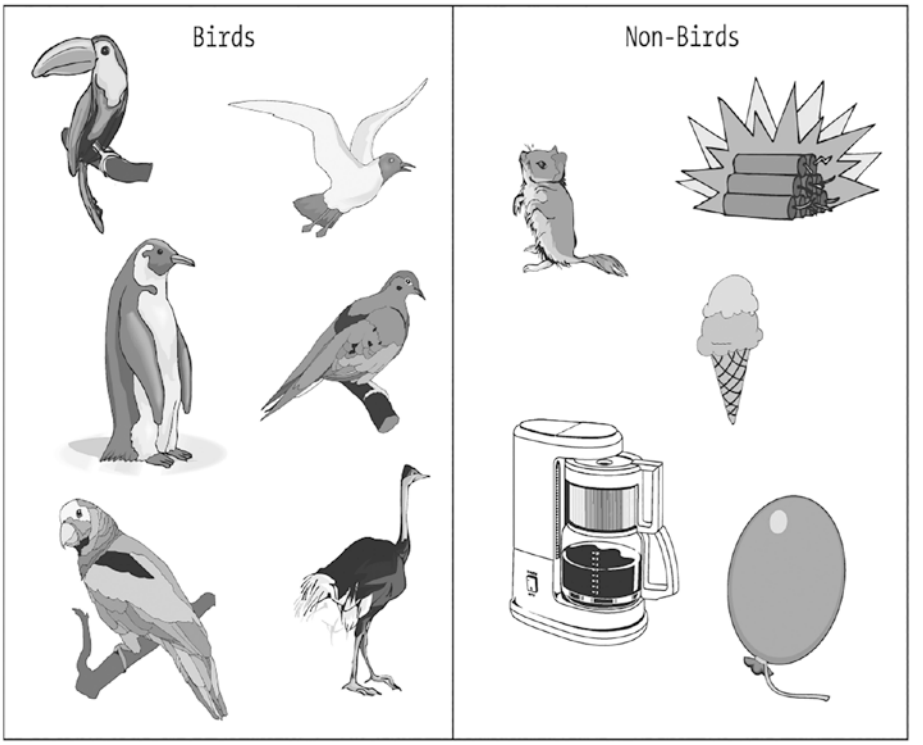


Figure 1-5. *Proper classification rules have been established*

This rule set is still unnecessarily complicated, because as it turns out, the "lays eggs" rule is redundant: whether we keep it or eliminate it, it doesn't change our decision of what constitutes a bird vs. a non-bird. Therefore, we simplify the rule set once again:

- Has feathers
- Has wings

Feeling particularly daring (!), we try to take our simplification process one step further by eliminating yet another rule, defining a bird as something that

- Has wings

As Figure 1-6 shows, we've gone too far this time: the abstraction of a bird is now so general that we'd include airplanes, insects, and all sorts of other non-birds in the mix!

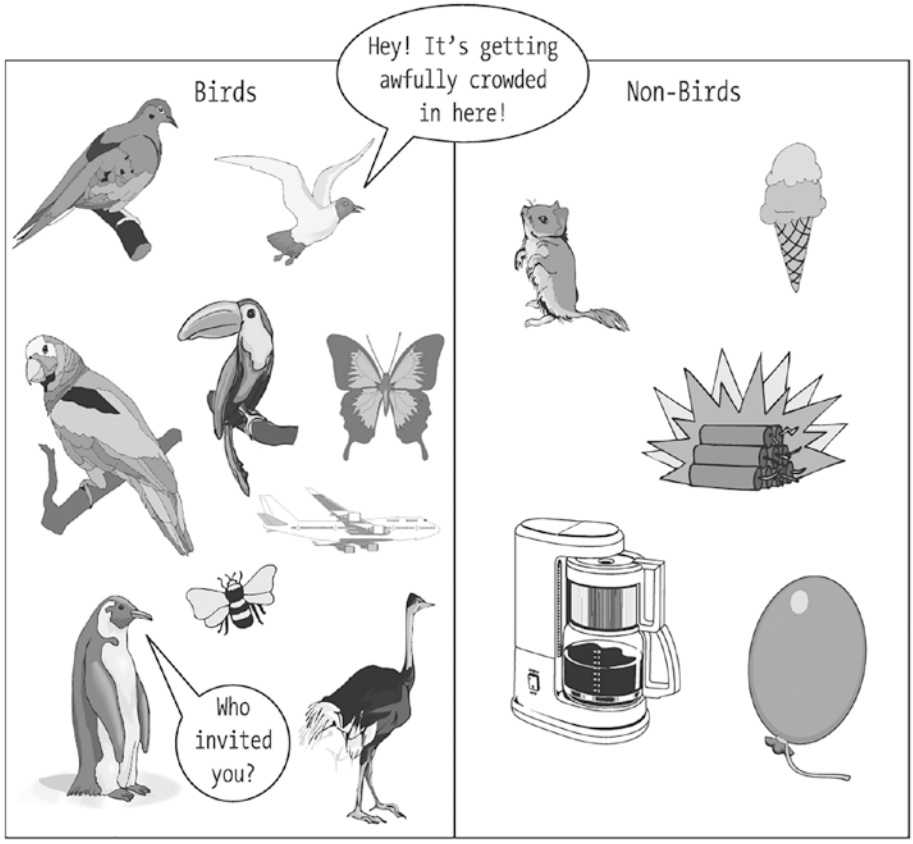


Figure 1-6. *A rule set that is too relaxed is as much of a problem as an overly restrictive rule set*

The process of rule definition for purposes of categorization involves dialing in just the right set of rules—not too general, not too restrictive, and containing no redundancies—to define the correct membership in a particular class.

Abstraction as the Basis for Software Development

When pinning down the requirements for an information systems development project, we typically start by gathering details about the real-world situation on which the system is to be based. These details are usually a combination of

- Those that are explicitly offered to us as we interview the intended users of the system
- Those that we otherwise observe

We must make a judgment call as to which of these details are relevant to the system's ultimate purpose. This is essential, as we can't automate them all. To include too much detail is to overly complicate the resultant system, making it that much more difficult to design, program, test, debug, document, maintain, and extend in the future.

As with all abstractions, all of our decisions of inclusion vs. elimination when building a software system must be made within the context of the overall purpose and **domain**, or subject matter focus, of the future system. When representing a person in a software system, for example, is their eye color important? How about their genetic profile? Salary? Hobbies? The answer is ***any*** of these features of a person may be relevant or irrelevant, depending on whether the system to be developed is a

- Payroll system
- Marketing demographics system
- Optometrist's patient database
- FBI's "most wanted" tracking system

Once we've determined the essential aspects of a situation—something that we'll explore in Part 2 of this book—we can prepare a **model** of that situation. **Modeling** is the process by which we develop a pattern for something to be made. A blueprint for a custom home, a schematic diagram of a printed circuit, and a cookie cutter are all examples of such patterns. As we'll cover in Parts 2 and 3, an **object model** of a software system is such a pattern. Modeling and abstraction go hand in hand, because a model is essentially a physical or graphical portrayal of an abstraction; before we can model something effectively, we must have determined the essential details of the subject to be modeled.

Reuse of Abstractions

When learning about something new, we automatically search our mental archive for other abstractions/models that we've previously built and mastered, to look for similarities that we can build upon. When learning to ride a two-wheeled bicycle for the first time, for example, you may have drawn upon lessons that you learned about riding a

tricycle as a child (see Figure 1-7). Both have handlebars that are used to steer; both have pedals that are used to propel the bike forward. Although the abstractions didn't match perfectly—a two-wheeled bicycle introduced the new challenge of having to balance yourself—there was enough of a similarity to allow you to draw upon the steering and pedaling expertise you already had mastered and to focus on learning the new skill of how to balance on two wheels.

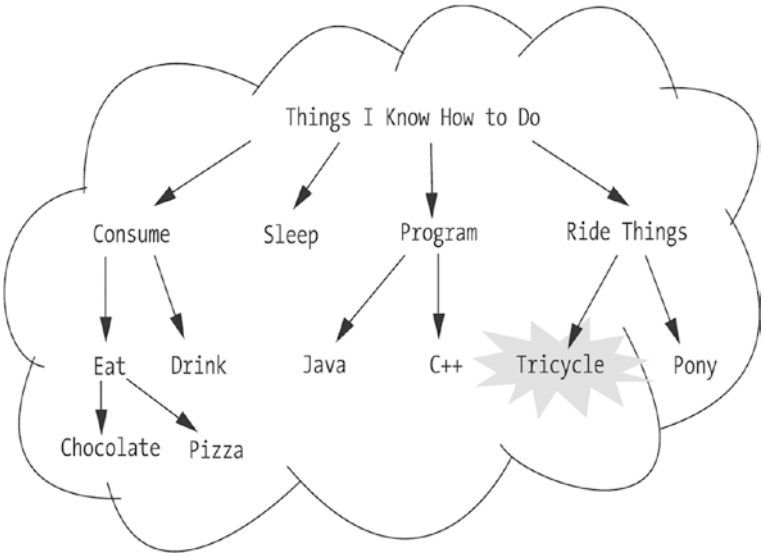


Figure 1-7. *The human brain is adept at learning by building upon already-established abstractions*

This technique of comparing features to find an abstraction that is similar enough to be reused successfully is known as **pattern matching and reuse**. As we'll cover later in the book, pattern reuse is an important technique for object-oriented software development, as well, because it spares us from having to reinvent the wheel with each new project. If we can reuse an abstraction or model from a previous project, we can focus on those aspects of the new project that differ from the old, gaining a tremendous amount of productivity in the process.

Inherent Challenges

Despite the fact that abstraction is such a natural process for human beings, developing an appropriate model for a software system is perhaps the most difficult aspect of software engineering, because

- There are an unlimited number of possibilities. Abstraction is to a certain extent in the eye of the beholder: several different observers working independently are almost guaranteed to arrive at different models. Whose is the best? ***Passionate*** arguments have ensued!
- To further complicate matters, there is virtually never only one "best" or "correct" model, only "better" or "worse" models relative to the problem to be solved. The same situation can be modeled in a variety of equally valid ways. When we get into actually doing some modeling in Part 2 of this book, we'll look at a number of valid alternative abstractions for our Student Registration System (SRS) case study that was presented at the end of the Introduction.
- Note, however, that there ***is*** such a thing as an incorrect model: namely, one that misrepresents the real-world situation (e.g., modeling a person as having two different blood types).
- There is no acid test to determine if a model has adequately captured all of a user's requirements. The ultimate evidence of whether or not an abstraction was appropriate is in how successful the resultant software system turns out to be. Because of this, it's critical that we learn ways of communicating our model concisely and unambiguously frequently throughout the Agile development life cycle to
- The intended future users of our application, so that they may provide a sanity check for our understanding of the problem to be solved before we embark upon software development
- Our fellow software engineers, so that team members can share a common vision of what we're to build collaboratively

Despite all of these challenges, it's critical to get the up-front abstraction "right" before beginning to build a system. The later in the software life cycle a modeling error is detected, the more costly it is to fix by orders of magnitude. This isn't to imply that an abstraction should be rigid—quite the contrary! The art and science of object modeling, when properly applied, yields a model that is flexible enough to withstand a wide variety of functional changes. In addition, the special properties of software objects further lend themselves to flexible software solutions, as you'll learn throughout the rest of the book.

What Does It Take to Be a Successful Object Modeler?

Coming up with an appropriate abstraction as the basis for a software system model requires

- ***Insight into the problem domain***: Ideally, we'll be able to draw upon our own real-world experiences, such as former or current experience as a student, which will come in handy when determining the requirements for the Student Registration System (SRS), the basis of our modeling and coding efforts in Parts 2 and 3 of the book.
- ***Creativity***: We must be able to think "outside the box," in case the future users whom we're interviewing have been immersed in the problem area for so long that they fail to see innovations that might be made.
- ***Good listening skills***: These will come in handy as future users of the system describe how they do their jobs currently or how they envision doing their jobs in the future, with the aid of the system that we're about to develop.
- ***Good observational skills***: Actions often speak louder than words. Just by observing users going about their daily business, we may pick up an essential detail that they have neglected to mention because they do it so routinely that it has become a habit.

But all this isn't enough. We also need

- An organized ***process*** for determining what the abstraction should be. If we follow a proven checklist of steps for producing a model, then we greatly reduce the probability that we'll omit some important feature or neglect a critical requirement.

- A way to ***communicate*** the resultant model concisely and unambiguously to our fellow software developers and to the stakeholders/intended users of our application. While it's certainly possible to describe an abstraction in narrative text, a picture is worth a thousand words, and so the language with which we communicate a model is often a **graphical notation**. Throughout this book, we'll focus on the Unified Modeling Language (UML; see Figure 1-8) notation as our model communication language (you'll learn the basics of UML in Chapters 10 and 11). Think of a graphical model as a blueprint of the software application to be built.

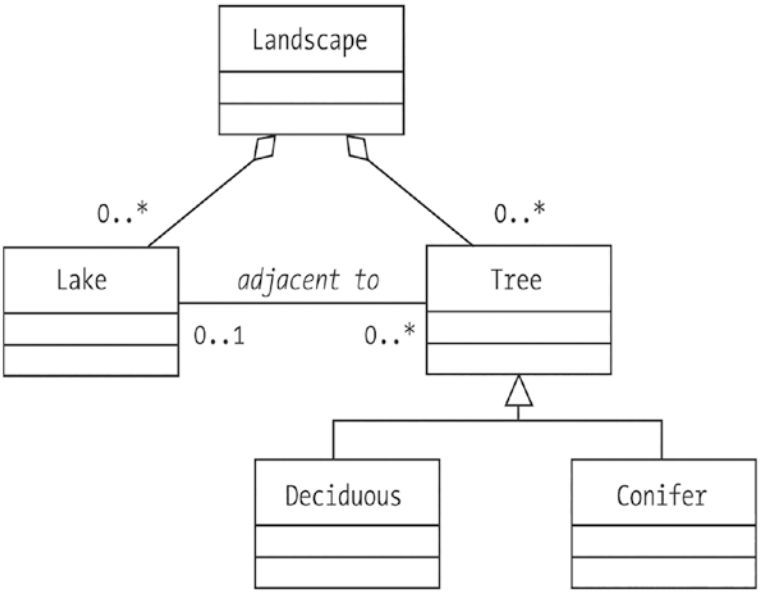


Figure 1-8. *Describing a landscape in UML notation*

- Ideally, a ***software tool*** to help us automate the process of producing such a blueprint.

Part 2 of this book covers these three aspects of modeling—process, notation, and tool—in detail.

Throughout the remainder of this book, we are going to focus on this following case study as the basis of object modeling and Java coding lessons:

STUDENT REGISTRATION SYSTEM (SRS) REQUIREMENTS SPECIFICATION

We have been asked to develop an automated Student Registration System (SRS). This system will enable students to register online for courses each semester, as well as tracking a student's progress toward completion of their degree.

When a student first enrolls at the university, they use the SRS to set forth a plan of study as to which courses they plan on taking to satisfy a particular degree program and choose a faculty advisor. The SRS will verify whether or not the proposed plan of study satisfies the requirements of the degree that the student is seeking. Once a plan of study has been established, then, during the registration period preceding each semester, students are able to view the schedule of classes online and choose whichever classes they wish to attend, indicating the preferred section (day of the week and time of day) if the class is offered by more than one professor. The SRS will verify whether or not the student has satisfied the necessary prerequisites for each requested course by referring to the student's online transcript of courses completed and grades received (the student may review their transcript online at any time).

Assuming that (a) the prerequisites for the requested course(s) are satisfied, (b) the course(s) meets one of the student's plan of study requirements, and (c) there is room available in each of the class(es), the student is enrolled in the class(es).

If (a) and (b) are satisfied, but (c) is not, the student is placed on a first-come, first-served waiting list. If a class/section that they were previously waitlisted for becomes available (either because some other student has dropped the class or because the seating capacity for the class has been increased), the student is automatically enrolled in the waitlisted class, and an email message to that effect is sent to the student. It is their responsibility to drop the class if it is no longer desired; otherwise, they will be billed for the course.

Summary

In this chapter, you've learned that

- Abstraction is a fundamental technique that people use to perceive the world.
- Developing an abstraction of the problem to be automated is a necessary first step of all software development.
- We naturally organize information into classification hierarchies based upon rules that we carefully structure, so that they are neither too general nor too restrictive.
- We often reuse abstractions when attempting to model a new concept.
- Producing an abstraction of a system to be built, known as a model, is in some senses second nature to us and yet paradoxically is one of the hardest things that software developers have to do in the life cycle of an information systems project. It's also one of the most important.

EXERCISES

1. Sketch a class hierarchy that relates to all of the following classes in a reasonable manner:

   Apple

   Banana

   Beef

   Beverage

   Cheese

   Consumable

   Dairy Product

   Food

Fruit

Green Bean

Meat

Milk

Pork

Spinach

Vegetable

Justify your answer, noting in particular any challenges that you faced in doing so.

2. What aspects of a television set would be important to abstract from the perspective of
   - A consumer wishing to buy one?
   - An engineer responsible for designing one?
   - A retailer who sells them?
   - The manufacturer?
3. Select a problem area that you would like to model from an object-oriented perspective. Ideally, this will be a problem that you're actually going to be working on at your place of employment or that you otherwise have a keen interest in. Assume that you're going to write a program to automate some aspect of this problem area, and write a one-page overview of the requirements for this program, patterned after the SRS case study.

   Make certain that your first paragraph summarizes the intent of the system, as the first paragraph in the SRS case study does. Also, emphasize the ***functional requirements***—that is, those that a nontechnical end user might state as to how the system should behave—and avoid stating ***technical requirements***, for example, "This system must run on a Unix platform and must use the TCP/IP protocol to…"

4. Read the case study for a Prescription Tracking System (PTS) in the Appendix. In your opinion, how effective is this case study as an abstraction? Are there details that you think could have been omitted or missing details that you think would have been important to include? If you had an opportunity to interview the intended users of the PTS, what additional questions might you ask them to better refine this abstraction?

CHAPTER 2

Some Java Basics

What you'll learn conceptually about objects in Part 1 of this book, and about object modeling in Part 2, is language neutral and thus could apply equally well to Java, Python, Ruby, or an as-yet-to-be-invented object-oriented (OO) language. Before diving into the basics of objects in Chapter 3, however, I'd like to spend some time getting you comfortable with the basics of Java, as this is the programming language that I'll use to illustrate object concepts as they are introduced throughout the rest of this book.

In this chapter, you'll learn about

- The architecture-neutral nature of Java
- The anatomy of a simple Java program
- The mechanics of compiling and running such programs
- Primitive Java types, operators on those types, and expressions formed with those types
- Java's **block-structured** nature
- Various types of Java **expressions**
- Loops and other flow control structures
- Printing messages to the command window from which a program was launched, which is especially useful for testing code as it evolves
- Elements of Java programming style

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_2

Java Is Architecture Neutral

To execute a program written in a conventionally compiled language like C or C++, the source code must first be compiled into an executable form known as **binary code** or **machine code**. Binary code, in essence, is a pattern of 1s and 0s understandable by the underlying **hardware architecture** of the computer on which the program is intended to run.

Even if original C or C++ source code is written to be **platform independent**—that is, the program does not take advantage of any platform-specific language extensions such as a specific type of file access or graphical user interface (GUI) manipulation—the resultant executable version will nonetheless still be tied to a particular platform's architecture and can therefore be run on only that architecture. That is, a version of the program compiled for a Linux workstation will not run on a Windows PC, a version compiled for a Windows PC will not run on a macOS machine, and so forth. This concept is depicted in Figure 2-1.

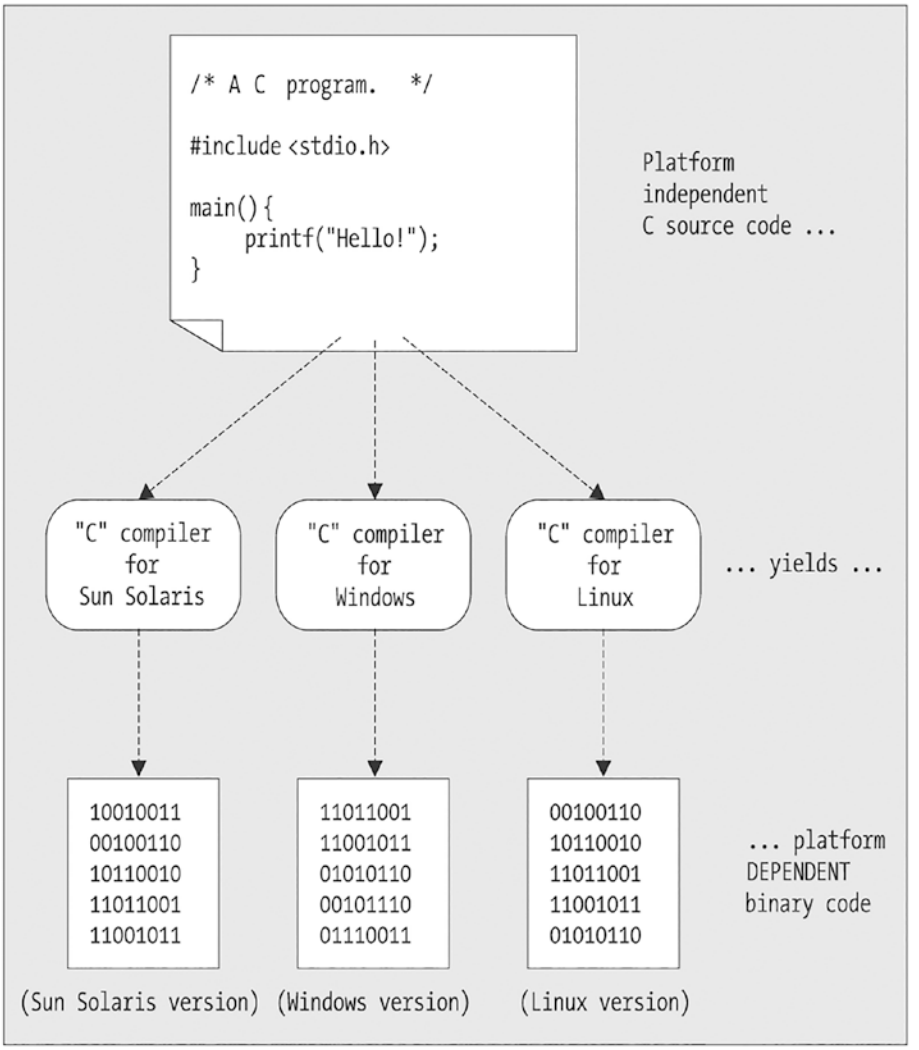


Figure 2-1. *Conventionally compiled languages yield platform-dependent executable programs*

In contrast, Java source code is not compiled for a particular platform, but rather into a special intermediate format known as **bytecode**, which is said to be both platform independent and **architecture neutral**. That is, no matter whether a Java program is compiled under Windows, Linux, macOS, or any other operating system for which a Java compiler is available, the resultant bytecode turns out to be the same and hence can be run on any computer for which a (platform-***specific***) Java Virtual Machine (JVM) is available. This is illustrated in Figure 2-2.

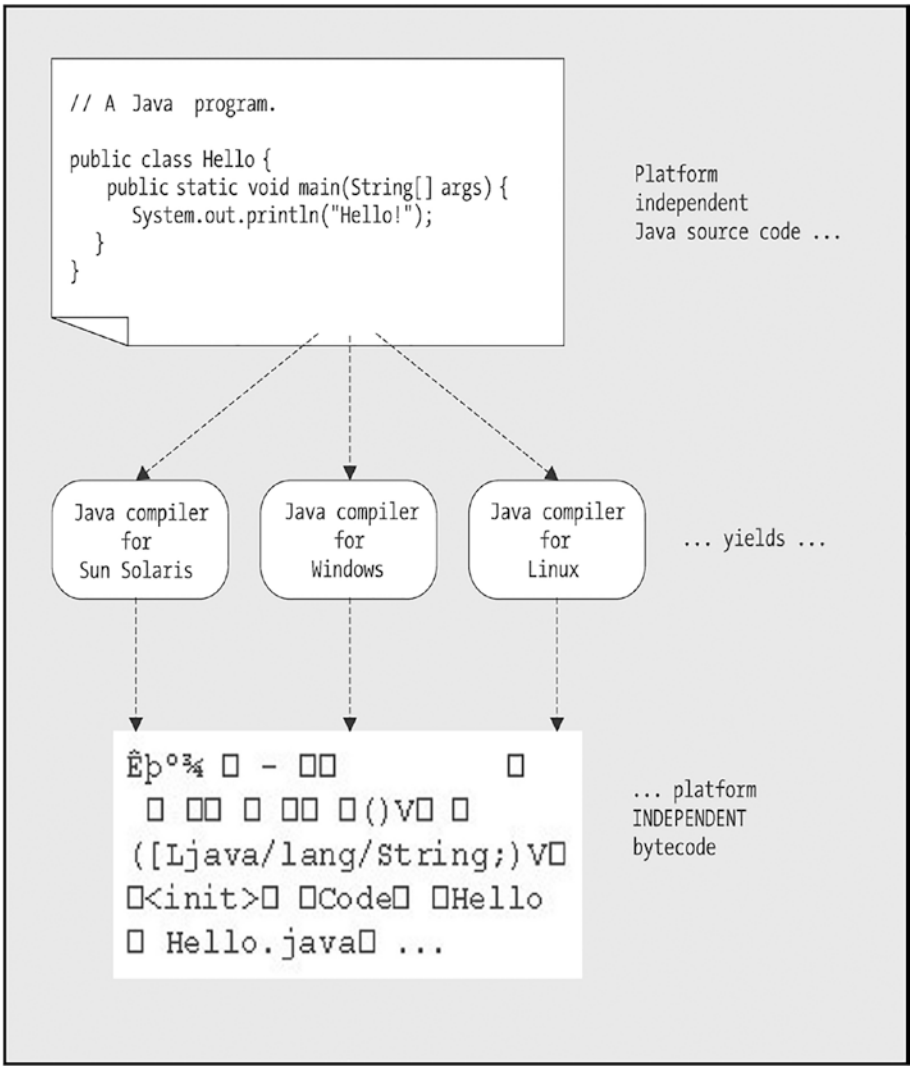


Figure 2-2. *The Java compiler yields platform-independent bytecode*

The JVM is a special piece of software that knows how to **interpret** and execute Java bytecode. That is, instead of a Java program running directly under the control of the operating system the way traditionally compiled programs do, the JVM itself runs under direct control of the operating system, and our Java program in turn runs under control of the JVM, as illustrated in Figure 2-3. The JVM in essence serves as a translator, translating the universal Java bytecode "language" into the machine code (binary code) "language" that a particular computer can understand, in the same way that a human interpreter facilitates a discussion between someone speaking German and someone speaking Japanese by translating their statements as they converse.

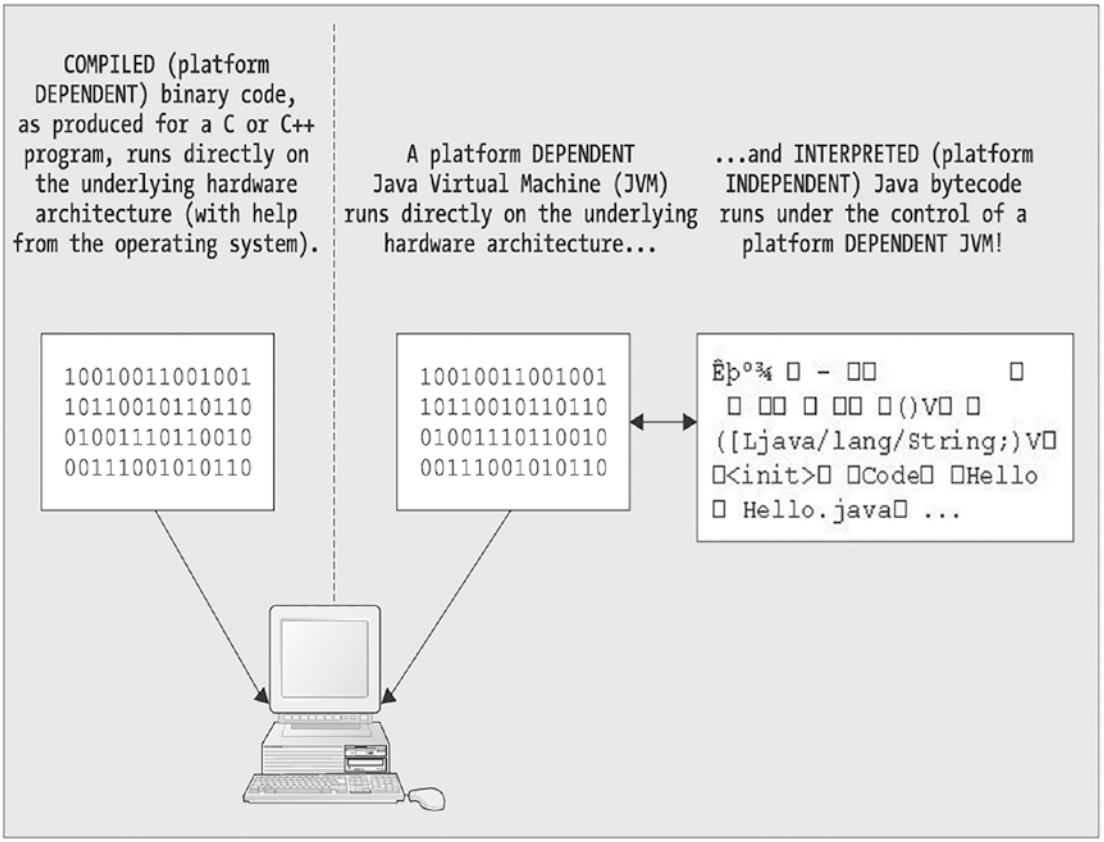


Figure 2-3. *A platform-dependent JVM is used to execute platform-independent bytecode*

The interpreted nature of the Java language tends to make it execute a tiny bit slower, in general, than compiled languages because there is an extra processing layer involved when an application executes, as illustrated in Figure 2-3. For traditional information systems applications that involve a human user in the loop, however, the difference in speed is imperceptible; other factors, such as the speed of the network (in the case of distributed applications), the speed of a DBMS server (if a database is used), and especially human "think time" while responding to an application's user interface, can cause any JVM response time delays to pale by comparison.

As long as you have the appropriate JVM installed on a given target platform, you can transfer Java bytecode from one platform to another without recompiling the original Java source code, and it will still be able to run. That is, bytecode is transferable across platforms, as illustrated in Figure 2-4.

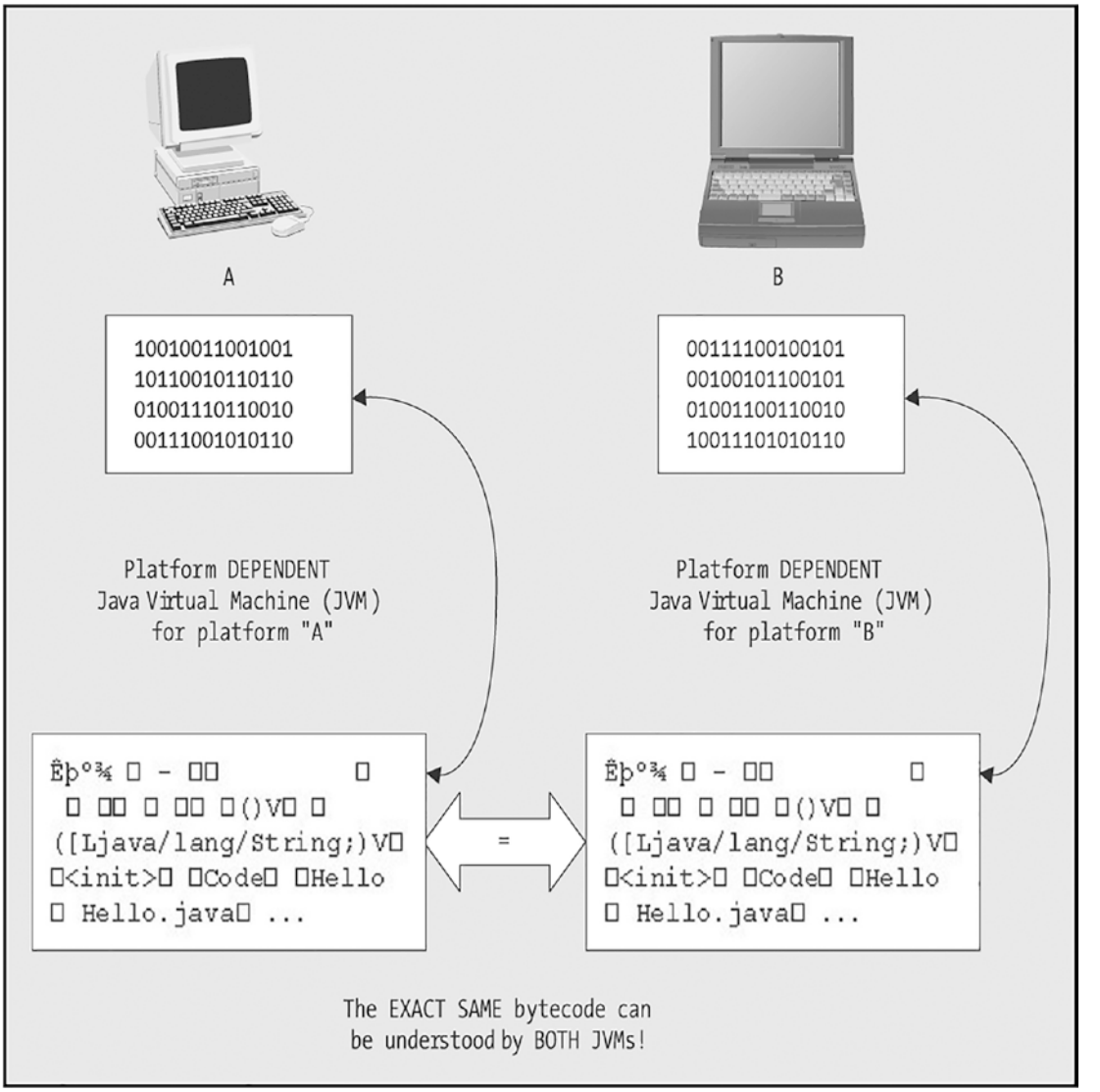


Figure 2-4. *The exact same bytecode is understood by JVMs on two different platforms*

PSEUDOCODE VS. REAL JAVA CODE

I occasionally use little bits of pseudocode in the code examples throughout Parts 1 and 2 of this book to hide irrelevant logic details. To make it clear as to when I'm using pseudocode rather than real code, I use *italic* rather than `regular SansMono condensed font`.

This is real Java syntax:

```
for (int i = 0; i <= 10; i++) {
```

This is pseudocode:

```
    compute the grade for the ith Student
}
```

I'll remind you of this fact a few more times, so that you don't forget and accidentally try to type in and compile pseudocode somewhere along the way.

Anatomy of a Simple Java Program

Figure 2-5 shows one of the simplest of all Java programs.

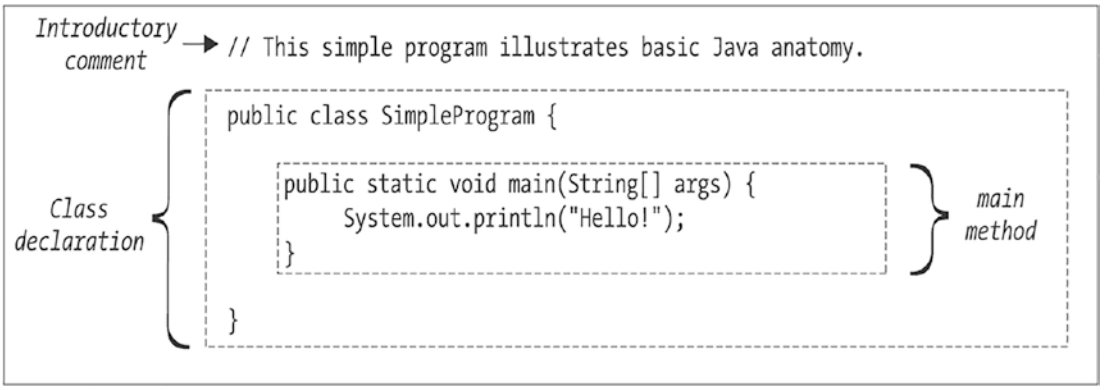


Figure 2-5. *Anatomy of a simple Java program*

Let's go over the key elements of our simple program.

Comments

The first thing we see in our simple Java program is an introductory comment:

```
// This simple program illustrates some basic Java syntax.
```

Java supports three different comment styles: traditional, end-of-line, and Java documentation comments.

Traditional Comments

Java **traditional comments** derive from the C language and begin with a forward slash followed by an asterisk (/*) and end with an asterisk followed by a forward slash (*/). Everything enclosed between these delimiters is treated as a comment and is therefore ignored by the Java compiler, no matter how many lines the comment spans:

```
/* This is a traditional (C-style) comment. */

/* This is a multiline traditional comment. This is a handy way to
   temporarily comment out entire sections of code without having
   to delete them. From the time that the compiler encounters the
   first "slash asterisk" above, it doesn't care what we type here;
   even legitimate lines of code, as shown below, are treated as
   comment lines and thus ignored by the compiler until the first
   "asterisk slash" combination is encountered.
x = y + z;
a = b / c;
j = s + c + f;
*/

/* We often use leading asterisks on the second through last line of a
   traditional
 * comment simply for cosmetic reasons, so that the comment is more
   visually
 * distinct; but, these extra asterisks are strictly cosmetic - only the
 * initial "slash asterisk" and the final "asterisk slash" are noted by the
 * compiler as having any significance.
 */
```

Note that we can't ***nest*** block comments—that is, the following will ***not*** compile:

```
/* This starts a comment ...
x = 3;
/* Whoops!  We are mistakenly trying to nest a SECOND comment
before terminating the FIRST!
This is going to cause us compilation problems, because the
compiler is going to IGNORE the start of this second/inner comment --
we're IN
a comment already, after all! -- and so as soon as we try to terminate
this SECOND/inner comment, the compiler will think that we've
terminated the
FIRST/outer comment instead ... */
z = 2;
// The compiler will "complain" on the next line.
*/
```

When the compiler reaches what we intended to be the terminating */ of the "outer" comment on the last line of the preceding code example, the following two compilation errors will be reported:

```
  illegal start of expression
    */
    ^
```

and

```
 ';' expected
    */
      ^
```

End-of-Line Comments

The second type of Java comment derives from C++ and is known as an **end-of-line comment**. We use a double slash (//) to note the beginning of a comment that automatically ends when the end of the line is reached, as shown here:

```
x = y + z;   // text of comment continues through to the end of the line ==>
a = b / c;

// Here's a BLOCK of sequential end-of-line comments.
```

```
// This serves as an alternative to using traditional comments
// (/* ... */) and is preferred by many Java programmers.

m = n * p;
```

Java Documentation Comments

The third and final type of Java comment, **Java documentation comments** (a.k.a. **"javadoc comments"**), can be parsed from source code files by a special `javadoc` command-line utility program (which comes standard with the Java Development Kit [JDK]) and used to automatically generate HTML documentation for an application.

We'll defer an in-depth look at javadoc comments until Chapter 13.

The Class Declaration

Next comes a **class wrapper**—more properly termed a **class declaration**—of the form

```
public class ClassName {
    ...
}
```

For example:

```
public class SimpleProgram {
    ...
}
```

where braces { `...` } enclose the **class body** that includes the main logic of the program along with other optional building blocks of a class.

In subsequent chapters, you'll learn all about the significance of classes in an OO programming language. For now, simply note that the symbols `public` and `class` are two of Java's **keywords**—that is, symbols reserved for specific uses within the Java language—whereas `SimpleProgram` is a name/symbol that I've invented.

The main Method

Within the `SimpleProgram` class declaration, we find the starting point for the program, called the **main method** in Java. (In an object-oriented language, functions are referred to as **methods**.) The `main` method serves as the entry point for a Java application. When we execute a Java program by interpreting its bytecode with an instance of the JVM, the JVM calls the `main` method to jump-start our application.

With trivial applications such as the `SimpleProgram` example, all of the program's logic can be contained within this single `main` method. For more complex applications, on the other hand, the `main` method can't possibly contain all of the logic for the entire system. You'll learn how to construct an application that transcends the boundaries of the `main` method, involving multiple Java source code files/classes, a bit later in the book.

The first line of the method, shown here

```
public static void main(String[] args) {
```

defines what is known as the `main` method's **method header** and should appear exactly as shown (for now–we'll revisit this topic again in Chapter 13).

Our `main` method's **method body**, enclosed in braces `{ ... }`, consists of a single statement:

```
System.out.println("Hello!");
```

which prints the message

```
Hello!
```

to the command window from which our program is launched. We'll examine this statement's syntax further in a bit, but for now, note the use of a semicolon (`;`) at the end of the statement. Semicolons are placed at the end of all individual Java statements. Braces `{ ... }`, in turn, delimit **blocks** of code, the significance of which I'll discuss in more detail a bit later in this chapter.

Other things that we'd typically do inside of the `main` method of a more elaborate program include declaring variables, initializing data, displaying a user interface, creating objects, and calling other methods.

Now that we've looked at the anatomy of a simple Java program, let's get a very simple Java development environment set up for you.

Setting Up a Simple Java Development Environment

For folks just getting started with Java programming, I advocate using a simple text editor at first so that you aren't distracted by the bells and whistles of a specific IDE and so that the IDE doesn't do so much work for you that you don't really learn what's going on at the most basic levels. That being said, an editor that understands Java syntax is definitely helpful, and so I personally recommend a very inexpensive tool called TextPad, available at `www.textpad.com/home`.

Once you have installed TextPad or selected another simple text editor of your choice, it's time to install the Java development environment. Because vendor instructions change frequently, it isn't practical for us to provide step-by-step instructions; as of the time of writing of this book, the latest instructions can be found on Oracle's website at `https://docs.oracle.com/en/java/javase/18/install`. If this link is out of date, please search the doc.oracle.com website for the latest version of the Java Platform, Standard Edition, JDK Installation Guide.

Once you've gotten your Java development environment up and running, let's take a look at how Java code is compiled and executed.

The Mechanics of Java

The simplest way to compile and execute a Java program regardless of platform is via command-line commands.

Compiling Java Source Code into Bytecode

To compile Java source code from the command line, we use the `cd` command, as necessary, to navigate into the working directory where our source code resides. We then type the following command

```
javac sourcecode_filename
```

for example

```
javac SimpleProgram.java
```

to compile it.

If there were more than one `.java` source code file in the same directory, we could either list the names of the files to be compiled, separated by spaces

```
javac Foo.java Bar.java Another.java
```

or use the wildcard character (*), for example

```
javac *.java
```

to compile multiple files at the same time.

If all goes well—namely, if no compiler errors arise—then a bytecode file by the name of `SimpleProgram.class` will appear in the same directory where the `SimpleProgram.java` source code file resides. If compiler errors do arise, on the other hand, we of course must correct our source code and attempt to recompile it.

Executing Bytecode

Once a program has been successfully compiled, we execute the bytecode version via the command

```
java bytecode_filename     (note that we OMIT the .class suffix)
```

For example:

```
java SimpleProgram
```

Note that it's important to ***omit*** the `.class` suffix of the bytecode file name (which is `SimpleProgram.class` in this case). Typing the suffix will result in the error shown in the following:

```
Exception in thread "main" java.lang.NoClassDefFoundError: SimpleProgram/java
```

By default, the JVM will look in your default working directory for such bytecode files. If the JVM finds the specified bytecode file, it executes its `main` method, and your program is off and running!

If for some reason the bytecode you are trying to execute is not in this default location, you must inform the JVM of additional directories in which to search, known as **specifying a classpath**. You can do so by specifying a list of directories (separated by semicolons [;] under Windows or by colons [:] under Linux and macOS) after the `-cp` flag on the `java` command as follows:

```
java –cp list_of_directory_names_to_be_searched bytecode_filename
```

For example, on Windows:

```
java –cp C:\home\javastuff;D:\reference\workingdir;S:\foo\bar\files
SimpleProgram
```

At a minimum, we typically want the JVM to search our ***current working directory*** for bytecode files. This happens by default if no **–cp** value is provided

```
java SimpleProgram
```

but it's generally a best practice to specify the current working directory as a single period (`.`), which is the Windows/Linux/macOS shorthand for "the current working directory," as a classpath entry, for example

```
java –cp  .  SimpleProgram
```

or, when multiple entries are needed in the classpath, to specify " . " as one of the entries, for example, on Windows:

```
java –cp C:\home\javastuff;D:\reference\workingdir;. SimpleProgram
```

Now that we've looked at the mechanics of compiling and running Java programs, let's explore some of the basic syntax features of Java in more detail.

Primitive Types

Java is said to be a **strongly typed** programming language in that when a variable is declared, its type must also be declared. Among other things, declaring a variable's type tells the compiler how much memory to allocate for the variable at run time and also constrains the contexts in which that variable may subsequently be used in our program.

The Java language defines eight **primitive types** (all eight of these type names are Java keywords), as follows.

Four types of ***integer*** numeric data:

- `byte`: 8-bit unsigned integer
- `short`: 16-bit signed integer
- `int`: 32-bit signed integer
- `long`: 64-bit signed integer

Two ***floating-point*** numeric types:

- `float`: 32-bit single-precision floating point
- `double`: 64-bit double-precision floating point

Plus two additional primitive types:

- `char`: A single character, stored using 16-bit Unicode encoding (vs. 8-bit ASCII encoding), enabling Java to handle a wide range of international character sets.
- `boolean`: A variable that may only assume one of two values: `true` or `false` (both of these values are reserved words in Java). Boolean variables are often used as flags to signal whether or not some code should be conditionally performed, as in the following code snippet:

```
boolean error = false;  // Initialize the flag.
// ...

// Later in the program (pseudocode):
if (some error situation arises) {
     // Set the flag to true to signal that an error has occurred.
     error = true;
}
// ...

// Still later in the program:
// Test the flag's value.
if (error == true) {
    // Pseudocode.
    take corrective action ...
}
```

We'll talk specifically about the syntax of the `if` statement, one of several different kinds of Java flow control statements, a bit later in this chapter.

An important reminder If you wish to attempt to compile any of the Java code snippets that you come across throughout the book, remember that (a) pseudocode (*italicized*) won't compile and (b) all code must, at a minimum, be enclosed within a `main` method, which in turn must be enclosed within a `class` declaration, as was illustrated in Figure 2-5.

Variables

Before a variable can be used in a Java program, the type and name of the variable must be ***declared*** to the Java compiler, for example:

```
int count;
```

Assigning a value to a variable is accomplished by using the Java **assignment operator**, an equal sign (=). An assignment statement consists of a (previously declared) variable name to the left of the = and an expression that evaluates to the appropriate type to the right of the = (we'll cover some other types of Java expressions later in the chapter). For example:

```
int count = 1;

total = total + 4.0;  // Here, we assume that total was declared to be
                         a double
                      // variable earlier in the program.

price = cost + (a + b)/length;  // We once again assume that all
                                   variables were
                                // properly declared earlier in the
                                   program.
```

An initial value can be supplied/computed on the same line that declares the variable:

```
int count = 3;
```

Or a variable can be declared in one statement and then assigned a value in a separate statement later on in the program:

```
double total;
// intervening code ... details omitted
total = total + 4.0;
```

A value can be assigned to a `boolean` variable using the `true` or `false` literal:

```
boolean finished;
// ...
finished = true;
```

A literal value may be assigned to a variable of type `char` by enclosing the value (a single Unicode character) in ***single*** quotes:

```
char c = 'A';
```

The use of ***double*** quotes ("...") is reserved for assigning literal values to `String` variables, a distinct type discussed later in this chapter. The following would not compile in Java:

```
char c = "A"; // We must use single quotes when assigning
values to char variables.
```

Variable Naming Conventions

When discussing Java variable names, there are two aspects to consider:

- First, is a particular name deemed ***valid*** by the Java compiler?
- Second, does a particular ***valid*** name adhere to the naming ***convention*** that has been adopted by the OO programming community across all languages?

Valid variable names in Java must start with either an alphabetic character or a dollar sign (whose use is discouraged, since it is used by the compiler when generating code) and may contain any of these characters plus numeric digits. No other characters are allowed in variable names.

The following are all valid variable names in Java:

```
int simple;               // starts with alphabetic character
int more$money_is_2much;  // may contain dollar signs, and/or
                             underscores, and/or
                                          // digits, and/or alphabetic
                                             characters
```

These are invalid:

```
int 1bad;                 // inappropriate starting character
int number#sign;          // contains invalid character
int foo-bar;              // ditto
int plus+sign;            // ditto
int x@y;                  // ditto
int dot.notation;         // ditto
```

That being said, the ***convention*** that is observed throughout the OO programming community is to form variable names using primarily alphabetic characters, avoiding the use of underscores, and furthermore to adhere to a style known as **camel casing**. With camel casing, the first letter of a variable name is in ***lowercase***, the ***first*** letter of each subsequent concatenated word in the variable name is in ***uppercase***, and the rest of the characters are in ***lowercase***. All of the following variable names are both valid and conventional:

```
int grade;
double averageGrade;
String myPetRat;
boolean weAreFinished;
```

Recall that, as mentioned earlier, Java keywords can't be used as variable names. The following won't compile, because `public` is a Java keyword:

```
int public;
```

In fact, the compiler would generate the following ***two*** error messages:

```
not a statement
int public;
^
```

```
';' expected
int public;
    ^
```

Variable Initialization

In Java, variables aren't necessarily assigned an initial value when they're declared, but all variables ***must*** be explicitly assigned a value before the variable's value is ***used*** in an assignment statement. For example, in the following code snippet, two `int`(eger) variables are declared; an initial value is explicitly assigned to variable `foo`, but not to variable `bar`. A subsequent attempt to add the two variables' values together results in a compiler error:

```
int foo;
int bar;
// We're explicitly initializing foo, but not bar.
foo = 3;
foo = foo + bar;   // This line won't compile.
```

The following compiler error would arise on the last line of code:

```
variable bar might not have been initialized
foo = foo + bar;
            ^
```

To correct this error, we would need to assign an explicit value to `bar`, as well as `foo`, before using them in the addition expression:

```
int foo;
int bar;
foo = 3;
// We're now initializing BOTH variables explicitly.
bar = 7;
foo = foo + bar;  // This line will now compile properly.
```

In Chapter 13 you'll learn that the rules of automatic initialization are somewhat different when dealing with the "inner workings" of objects.

The String Type

I'll discuss one more important Java type in this chapter: the `String` type, which is not considered to be a primitive type (we'll discuss the special nature of Strings as objects in Chapter 13). A `String` represents a sequence of zero or more Unicode characters.

The symbol `String` starts with a capital "S," whereas the names of primitive types are expressed in all lowercase: `int`, `float`, `boolean`, etc. This capitalization difference is deliberate and mandatory—`string` (lowercase) won't work as a type:

```
string s = "foo";  // This won't compile.
```

Error message:

```
cannot find symbol
symbol:  string
```

There are several ways to create and initialize a `String` variable. The easiest and most commonly used way is to declare a variable of type `String` and to assign the variable a value using a **string literal**. A string literal is any text enclosed in ***double*** quotes, even if it consists of only a ***single character***:

```
String name = "Steve";
String shortString = "A";
```

Two commonly used approaches for initializing a `String` variable with a temporary placeholder value are as follows:

- Assigning an empty string, represented by two consecutive double quote marks:

  ```
  String s = "";
  ```

- Assigning the value `null`, which is a Java reserved word that is used to signal that a `String` has not yet been assigned a "real" value:

  ```
  String s = null;
  ```

The plus sign (+) operator is used for arithmetic addition with numeric data types, but when used in conjunction with `String`s, it represents **string concatenation**. Any number of `String` values can be concatenated with the + operator, as the following code snippet illustrates:

```
String x = "foo";
String y = "bar";
String z = x + y + "!";  // z assumes the value "foobar!" (x and y's
                            values are
                         // unaffected)
```

You'll learn about some of the many other operations that can be performed with or on `String`s, along with insights into their OO nature, in Chapter 13.

Case Sensitivity

Java is a **case-sensitive** language. That is, the use of uppercase vs. lowercase in Java is deliberate and mandatory, for example:

- Variable names that are spelled the same way but that differ in their use of case represent ***different*** variables:

  ```
  // These are two DIFFERENT variables as far as the Java compiler
  // is concerned.
  int x;  // lowercase
  int X;  // uppercase
  ```

- All keywords are rendered in lowercase: `public`, `class`, `int`, `boolean`, and so forth. ***Don't get "creative" about capitalizing these***, as the compiler will violently object—often with unintelligible compilation error messages, as in the following example, where the reserved word `for` is improperly capitalized:

  ```
  // The reserved word 'for' should be lowercase.
  For (int i = 0; i < 3; i++) {
  ```

This in turn produces the following seemingly bizarre compiler error:

```
'.class' expected
For (int i = 0; i < 3; i++) {
         ^
```

- The name of the `main` method is lowercase.
- As mentioned earlier, the `String` type starts with an uppercase "S."

Java Expressions

Java is an **expression-oriented language**. A **simple expression** in Java is either

- ***A constant***: `7`, `false`
- ***A** char**acter) literal enclosed in single quotes***: `'A'`, `'3'`
- ***A** String **literal enclosed in double quotes***: `"foo"`, `"Java"`
- ***The name of any properly declared variables***: `myString`, `x`
- ***Any*** two ***of the preceding types of expression that are combined with one of the Java*** binary operators ***(discussed in detail later in this chapter)***: `x + 2`
- ***Any*** one ***of the preceding types of expression that is modified by one of the Java*** unary operators ***(discussed in detail later in this chapter)***: `i++`
- ***Any of the preceding types of expression enclosed in parentheses***: `(x + 2)`

plus a few more types of expression having to do with objects that you'll learn about later in the book.

Expressions of arbitrary complexity can be assembled from the various different simple expression types by nesting parentheses, for example: `((((4/x) + y) * 7) + z)`.

Arithmetic Operators

The Java language provides a number of basic arithmetic operators, as shown in Table 2-1.

Table 2-1. *Java Arithmetic Operators*

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder (the remainder when the operand to the left of the % operator is divided by the operand to the right, for example, `10 % 3 = 1`, because 3 goes into 10 three times, leaving a remainder of 1)

The + and - operators can also be used as unary operators to indicate positive or negative numbers: `-3.7`, `+42`.

In addition to the simple assignment operator, =, there are a number of specialized **compound assignment operators**, which combine variable assignment with an arithmetic operation, as shown in Table 2-2.

Table 2-2. *Java Compound Assignment Operators*

Operator	Description
+=	a += b is equivalent to a = a + b.
-=	a -= b is equivalent to a = a - b.
*=	a *= b is equivalent to a = a * b.
/=	a /= b is equivalent to a = a / b.
%=	a %= b is equivalent to a = a % b.

The final two arithmetic operators that we'll look at are the **unary increment** (++) and **decrement** (--) operators, which are used to increase or decrease the value of an `int` variable by 1 or of a floating-point (`float`, `double`) value by 1.0. They're known as unary operators because they're applied to a single variable, whereas binary operators combine the values of two expressions as discussed previously.

The unary increment and decrement operators can also be applied to `char` variables to step forward or backward one character position in the Unicode sorting sequence. For example, in the following code snippet, the value of variable `c` will be incremented from 'e' to 'f':

```
char c = 'e';
c++; // c will be incremented from 'e' to 'f'.
```

The increment and decrement operators can be used in either a **prefix** or **postfix** manner. If the operator is placed ***before*** the variable it's operating on (***prefix*** mode), the increment or decrement of that variable is performed ***before*** the variable's ***updated*** value is used in any assignments made via that statement. For example, consider the following code snippet, which uses the prefix increment (++) operator. Assume that `a` and `b` have previously been declared as `int` variables in our program:

```
a = 1;
b = ++a;
```

After the preceding lines of code have executed, the value of variable `a` will be 2, as will the value of variable `b`. This is because, on the second line of code, the increment of variable `a` (from 1 to 2) occurs ***before*** the value of `a` is assigned to variable `b`. Thus, the single line of code

```
b = ++a;
```

is logically equivalent to the following ***two*** lines of code:

```
a = a + 1;  // Increment a's value first ...
b = a;      // ... THEN use its value.
```

On the other hand, if the increment/decrement operator is placed ***after*** the variable it's operating on (***postfix*** mode), the increment or decrement occurs ***after*** the variable's ***original*** value is used in any assignments made via that statement. Let's look at the same code snippet with the increment operator written in a ***postfix*** manner:

```
a = 1;
b = a++;
```

After the preceding lines of code have executed, the value of variable b will be 1, whereas the value of variable a will be 2. This is because, on the second line of code, the increment of variable a (from 1 to 2) occurs ***after*** the value of a is assigned to variable b. Thus, the single line of code

```
b = a++;
```

is logically equivalent to the following ***two*** lines of code:

```
b = a;      // Use a's value first `...
a = a + 1;  // ... THEN increment its value.
```

Here's a slightly more complex example; please read the accompanying comment to make sure that you can see how x will end up being assigned the value 10:

```
int y = 2;
int z = 4;
int x = y++ * ++z;  // x will be assigned the value 10, because z will be
                    // incremented from 4 to 5 BEFORE its value is
                       used in the
                    // multiplication expression, whereas y will remain
                       at 2 until
                    // AFTER its value is used in the multiplication
                       expression.
```

As you'll see in a bit, the increment and decrement operators are commonly used in conjunction with loops.

Relational and Logical Operators

A **logical expression** compares two (simple or complex) expressions *exp1* and *exp2* in a specified way, resolving to a `boolean` value of `true` or `false`.

To create logical expressions, Java provides the **relational operators** shown in Table 2-3.

Table 2-3. *Java Relational Operators*

Operator	Description
exp1 == *exp2*	`true` if *exp1* equals *exp2* (note the use of a ***double*** equal sign for testing equality)
exp1 > *exp2*	`true` if *exp1* is greater than *exp2*
exp1 >= *exp2*	`true` if *exp1* is greater or equal to *exp2*
exp1 < *exp2*	`true` if *exp1* is less than *exp2*
exp1 <= *exp2*	`true` if *exp1* is less than or equal to *exp2*
exp1 != *exp2*	`true` if *exp1* is not equal to *exp2* (`!` is read as "not")

In addition to the relational operators, Java provides **logical operators** that can be used to combine/modify logical expressions. The most commonly used logical operators are listed in Table 2-4.

Table 2-4. *Java Logical Operators*

Operator	Description
exp1 && *exp2*	Logical "and"; compound expression is true only if *both exp1 and exp2* are true.
exp1 || *exp2*	Logical "or"; compound expression is true if *either exp1 or exp2* is true.
! *exp*	Logical "not"; `true` if *exp* is false and `false` if *exp* is true.

Here's an example that uses the logical "and" operator to program the compound logical expression "if x is greater than 2.0 and y is not equal to 4.0":

```
if ((x > 2.0) && (y != 4.0)) { ... }
```

Logical expressions are most commonly seen used with flow control structures, discussed later in this chapter.

Evaluating Expressions and Operator Precedence

As mentioned earlier in the chapter, expressions of arbitrary complexity can be built up by layering nested parentheses—for example, `(((8 * (y + z)) + y) * x)`. The compiler generally evaluates such expressions from the innermost to outermost parentheses, left to right. Assuming that `x`, `y`, and `z` are declared and initialized as shown here

```
int x = 1;
int y = 2;
int z = 3;
```

then the expression on the right-hand side of the following assignment statement

```
int answer = ((8 * (y + z)) + y) * x;
```

would be evaluated piece by piece as follows:

```
((8 * (y + z)) + y) * x
   ((8 * 5) + y) * x
     (40 + y) * x
        42 * x
          42
```

In the absence of parentheses, certain operators take precedence over others in terms of when they will be applied in evaluating an expression. For example, multiplication or division is performed before addition or subtraction. Operator precedence can be explicitly altered through the use of parentheses; operations performed inside parentheses take precedence over operations outside of parentheses. Consider the following code snippet:

```
int j = 2 + 3 * 4;     // j will be assigned the value 14
int k = (2 + 3) * 4;   // k will be assigned the value 20
```

On the first line of code, which uses no parentheses, the multiplication operation takes precedence over the addition operation, and so the overall expression evaluates to the value 2 + 12 = 14; it's as if we've explicitly written `2 + (3 * 4)` without having to do so.

On the second line of code, parentheses are explicitly placed around the operation 2 + 3 so that the addition operation will be performed first, and the resultant sum will then be multiplied by 4 for an overall expression value of 5 * 4 = 20.

Returning to an earlier example

```
if ((x > 2.0) && (y != 4.0)) { ... }
```

note that the > and != operators take precedence over the && operator, such that we could eliminate the nested parentheses as follows:

```
if (x > 2.0 && y != 4.0) { ... }
```

However, the extra parentheses certainly don't hurt, and in fact it can be argued that they make the expression's intention clearer.

The Type of an Expression

The **type of an expression** is the Java type of the value to which the expression ultimately evaluates. For example, given the code snippet

```
double x = 3.0;
double y = 2.0;
if ((x > 2.0) && (y != 4.0)) { ... }
```

the expression `(x > 2.0) && (y != 4.0)` evaluates to `true`, and hence the expression `(x > 2.0) && (y != 4.0)` is said to be a `boolean`-type expression. However, in the following code snippet

```
int x = 1;
int y = 2;
int z = 3;
int answer = ((8 * (y + z)) + y) * x;
```

the expression `((8 * (y + z)) + y) * x` evaluates to `42`, and hence the expression `((8 * (y + z)) + y) * x` is said to be an `int`(eger)-type expression.

Automatic Type Conversions and Explicit Casting

Java supports **automatic type conversion**. This means that if we try to assign a value to a variable

```
// Pseudocode.
x = expression;
```

and the expression on the ***right-hand side*** of the assignment statement evaluates to a ***different*** type than the type with which the variable on the ***left-hand side*** of the assignment statement was declared, Java will automatically convert the value of the right-hand expression to match the type of `x`, but ***only if precision won't be lost in doing so***. This is best understood by looking at an example:

```
int x;
double y;
y = 2.7;
x = y;  // We're trying to assign a double value to an int variable.
```

In the preceding code snippet, we're attempting to assign the `double` value of `y`, 2.7, to `x`, which is declared to be an `int`. If this assignment were to take place, the fractional part of `y` would be truncated, and `x` would wind up with an integer value of 2. This represents a loss in precision, also known as a **narrowing conversion**.

A C or C++ compiler will permit this assignment, silently truncating the value. Rather than assuming that this is what we intend to do, however, the Java compiler will generate an error on the last line of code as follows:

```
possible loss of precision
found: double
required: int
```

In order to signal to the Java compiler that we're willing to accept the loss of precision, we must perform an **explicit cast**—that is, we must precede the expression whose value is to be converted with the target type, enclosed in parentheses:

```
// Pseudocode.
x = (type) expression;
```

In other words, we'd have to rewrite the last line of the preceding example as follows in order for the Java compiler to permit the assignment of a more precise floating-point value to a less precise integer variable:

```
int x;
double y;
y = 2.7;
```

```
x = (int) y;  // This will compile now, because we have explicitly
              // informed the compiler that we WANT a
              // narrowing conversion to occur.
```

Of course, if we were to ***reverse*** the direction of the assignment, assigning the `int` value of variable `x` to the `double` variable `y`, the Java compiler would have no problem with the assignment:

```
int x;
double y;
x = 2;
y = x;    // Assign a less precise int value to a double variable that is
             capable of
          // handling more precision - this is fine as is.
```

In this particular case, we're assigning a value of ***less*** precision—2—to a variable capable of ***more*** precision; `y` will wind up with the value of 2.0. This is known as a **widening conversion**. Such conversions are performed automatically in Java and need not be explicitly cast.

Note that there's an idiosyncrasy with regard to assigning constant values to variables of type `float` in Java; the following statement won't compile

```
  float y = 3.5;   // This won't compile!
```

because a numeric constant value with a fractional component like `3.5` is automatically treated by Java as a more precise `double` value and so the compiler will view this as a narrowing conversion and will refuse to carry out the assignment. To ***force*** such an assignment, we must either explicitly cast the floating-point constant into a `float`

```
  float y = (float) 3.5;   // This will compile, thanks to the cast.
```

or, alternatively, we can force the constant on the right-hand side of the assignment statement to be viewed by the Java compiler as a `float` by using the suffix `F` or `f`, as shown here:

```
  float y = 3.5F;   // OK, because we're indicating that the constant
                       is to be
                         // treated as a float, not as a double.
```

We'll typically use `doubles` instead of `floats` whenever we need to declare floating-point variables in our SRS application in Part 3 of the book, just to avoid these hassles of type conversion.

Expressions of type `char` can be converted to any other numeric type, as illustrated in this next example:

```
char c = 'a';

// Assigning a char value to a numeric variable transfers its
// ASCII numeric equivalent value.
int x = c;
double z = c;

System.out.println(x);
System.out.println(z);
```

Here's the output:

```
97
97.0
```

The only Java type that can't be cast, either implicitly or explicitly, into another type is the `boolean` type.

You'll see other applications of casting, involving objects, later in the book.

Loops and Other Flow Control Structures

Very rarely will a program execute sequentially, line by line, from start to finish. Instead, the path of execution through a program's logic will often be ***conditional***:

- It may be necessary to have the program execute a certain block of code if some condition is met or a ***different*** block of code if the condition ***isn't*** met.
- A program may have to repeatedly execute a particular block of code a fixed number of times or until a particular result is attained.

The Java language provides a number of different types of loops and other flow control structures to take care of these situations.

if Statements

The `if` statement is a basic conditional branch statement that executes one or more lines of code if a condition, represented as a logical expression, is satisfied. Alternatively, one or more lines of code can be executed if the condition is ***not*** satisfied by placing that code after the keyword `else`. The use of an `else` clause within an `if` statement is optional.

The basic syntax of the `if` statement is as follows:

```
// Pseudocode.
if (logical-expression) {
    execute whatever code is contained within these braces if
    logical-expression evaluates to true
}
```

Or add an optional `else` clause:

```
// Pseudocode.
if (logical-expression) {
    execute whatever code is contained within these braces if
    logical-expression evaluates to true
}
else {
    execute whatever code is contained within these braces if
    logical-expression evaluates to false
}
```

If only one executable statement follows either the `if` or (optional) `else` keyword, the braces can be omitted, as shown here:

```
// Pseudocode.
if (logical-expression) single statement to execute if logical-expression
is true;
else single statement to execute if logical-expression is false;
```

For example:

```
if (x > 3) y = x;
else z = x;
```

Alternatively, optional line breaks can be inserted:

```
if (x > 3)
    y = x;
else
    z = x;
```

But it's generally considered good practice to always use braces as follows:

```
if (x > 3) {
    y = x;
}
else {
    z = x;
}
```

A single `boolean` variable, as a simple form of Boolean expression, can serve as the logical expression/condition of an `if` statement. For example, it's perfectly acceptable to write the following:

```
// Use boolean variable "finished" as a flag that will get set to true when
// some particular operation is completed.
boolean finished;

// Initialize it to false.
finished = false;

// The details of intervening code, in which the flag may or may not
get set to
// true, have been omitted ...

// Test the flag.
if (finished) { // equivalent to:  if (finished == true) {
    System.out.println("We are finished!  :o)");
}
```

The ! ("not") operator can be used to negate a logical expression, so that the block of code associated with an `if` statement is executed when the expression evaluates to `false` instead:

```
if (!finished) { // equivalent to:  if (finished == false)
    System.out.println("We are NOT finished ... :op");
}
```

When testing for equality of two expressions, remember that we must use ***two*** consecutive equal signs, not just one:

```
if (x == 3) { // Note use of double equal signs (==) to test for equality.
  y = x;
}
```

A common mistake made by beginning Java programmers—particularly those who've previously programmed in C or C++—is to try to use a ***single*** equal sign to test for equality, as in this example:

```
// Note incorrect use of single equal sign below.
if (x = 3) {
    y = x;
}
```

In Java, an `if` test must be based on a valid ***logical*** expression; `x = 3` isn't a logical expression, but rather an ***assignment*** expression. In fact, the preceding `if` statement won't even compile in Java, whereas it ***would*** compile in the C and C++ programming languages, because in those languages, `if` tests are based on evaluating expressions to either the ***integer*** value 0 (interpreted to mean `false`) or nonzero (interpreted to mean `true`).

It's possible to nest `if-else` constructs to test more than one condition. If nested, an inner `if` (plus optional `else`) statement is placed within either the `if` part or the `else` part of an outer `if`. Here is one commonly seen syntax for a two-level nested `if-else` construct:

```
if (logical-expression-1) {
    execute this code
}
else {
    if (logical-expression-2) {
        execute this alternate code
    }
    else {
        execute this code if neither of the above expressions
        evaluate to true
    }
}
```

There's no limit to how many nested `if-else` constructs can be used, although if nested too deeply, the code may become difficult for a human reader to understand and hence maintain.

The nested `if` statement shown in the preceding example may alternatively be written without using nesting as follows:

```
if (logical-expression-1) {
    execute this code
}
else if (logical-expression-2) {
    execute this alternate code
}
else {
    execute this code if neither of the above expressions evaluate to true
}
```

Note that the two forms are logically equivalent.

switch Statements

A `switch` statement is similar to an `if-else` construct in that it allows the conditional execution of one or more lines of code. However, instead of evaluating a logical expression as an `if-else` construct does, a `switch` statement compares the value of an expression against values defined by one or more `case` labels. If a match is found,

the code following the matching `case` label is executed. An optional `default` label can be included to define code that is to be executed if the expression matches ***none*** of the `case` labels.

The general syntax of a `switch` statement is as follows:

```
switch (int-or-char-expression) {
    case value1:
        one or more lines of code to execute if value of expression
        matches value1
        break;
    case value2:
        one or more lines of code to execute if value of expression
        matches value2
        break;
    // more case labels, as needed ...
    case valueN:
        one or more lines of code to execute if value of expression
        matches valueN
        break;
    default:
        default code to execute if none of the cases match
}
```

For example:

```
// x is assumed to have been previously declared as an int.
switch (x) {
    case 1:  // executed if x equals 1
      System.out.println("One ...");
      break;
    case 2:  // executed if x equals 2
      System.out.println("Two ...");
      break;
    default:  // executed if x has a value other than 1 or 2
      System.out.println("Neither one nor two ...");
}
```

Note the following:

- The expression in parentheses following the `switch` keyword must be an expression that evaluates to a `char` or `int` value (or, as we will learn later in the book, to a few other special expression types).
- The values following the `case` labels must be constant `char` or `int` values (or, as we will learn later in the book, to a few other special expression types).
- The break; statement causes the switch to stop executing once a given case has been completed; execution resumes after the closing } of the switch statement.
- Colons (`:`), not semicolons (`;`), terminate the `case` and `default` labels.
- The statements following a given `case` label don't have to be enclosed in braces. They constitute a **statement list** rather than a code block.

Unlike an `if` statement, a `switch` statement isn't automatically terminated when a match is found and the code following the matching `case` label is executed. To exit a `switch` statement, a `break` statement must be used. If a `break` statement isn't included following a given `case` label, execution will "fall through" to the next `case` or `default` label. This behavior can be used to our advantage: when the same logic is to be executed for more than one `case` label, two or more `case` labels can be stacked up back to back, as shown here:

```
// x is assumed to have been previously declared as an int
switch (x) {
    case 1:
        code to be executed if x equals 1
    case 2:
        code to be executed if x equals 1 OR 2
    case 3:
        code to be executed if x equals 1, 2, OR 3
        break;
    case 4:
        code to be executed if x equals 4
}
```

for Statements

A `for` statement is a programming construct that is used to execute one or more statements a certain number of times. The general syntax of the `for` statement is as follows:

```
for (initializer; condition; iterator) {
    code to execute while condition evaluates to true
}
```

A `for` statement defines three elements that are separated by semicolons and placed in parentheses after the `for` keyword: the **initializer**, the **condition**, and the **iterator**. The ***initializer*** is used to provide an initial value for a **loop control variable**. The variable can be declared as part of the initializer, or it can be declared earlier in the code, ahead of the `for` statement, for example:

```
// The loop control variable 'i' is declared within the for statement:
for (int i = 0; condition; iterator) {
    code to execute while condition evaluates to true
}
// Note that i is no longer recognized by the compiler when the 'for'
loop exits,
// because it was effectively declared within the 'for' loop - we'll talk
about the
// scope of a variable later in this chapter.
```

or

```
// The loop control variable 'i' is declared prior to the start of the
'for' loop:
int i;

for (i = 0; condition; iterator) {
    code to execute while condition evaluates to true
}
```

The ***condition*** is a logical expression that typically involves the loop control variable:

```
for (int i = 0; i < 5; iterator) {
    code to execute as long as the value of i remains less than 5
}
```

The ***iterator*** typically increments or decrements the loop control variable:

```
for (int i = 0; i < 5; i++) {
    code to execute as long as the value of i remains less than 5
}
```

Again, note the use of a semicolon (`;`) after the initializer and condition, but ***not*** after the iterator.

Here's a breakdown of how a `for` loop operates:

1. When program execution reaches a `for` statement, the initializer is executed first (and only once).
2. The ***condition*** is then evaluated. If the condition evaluates to `true`, the block of code following the parentheses is executed.
3. After the block of code finishes, the iterator is executed.
4. The condition is then reevaluated. If the condition is still `true`, the block of code is executed once again, followed by execution of the iterator statement.

This process repeats until the condition becomes `false`, at which point the `for` loop terminates.

Here's a simple example that uses nested `for` statements to generate a simple multiplication table. The loop control variables, `j` and `k`, are declared inside their respective `for` statements. As long as the conditions in the respective `for` statements are met, the block of code following the `for` statement is executed. The ++ operator is used to increment the values of `j` and `k` each time the respective block of code is executed:

```
public class ForDemo {
    public static void main(String[] args) {
        // Compute a simple multiplication table.
        for (int j = 1; j <= 4; j++) {
            for (int k = 1; k <= 4; k++) {
                System.out.println(j + " * " + k + " = " + (j * k));
            }
        }
    }
}
```

Here's the output:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
4 * 1 = 4
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
```

As with other flow control structures, if only one statement is specified after the `for` condition, the braces can be omitted:

```
for (int i = 0; i < 3; i++) sum = sum + i;
```

or alternatively

```
for (int i = 0; i < 3; i++)
    sum = sum + i;
```

But it's considered good programming practice to use braces regardless:

```
for (int i = 0; i < 3; i++) {
    sum = sum + i;
}
```

while Statements

A `while` statement is similar in function to a `for` statement in that both are used to repeatedly execute an associated block of code. However, if it's impractical to predict the number of times that the code block is to be executed when the loop first begins, a `while` statement is the preferred choice, because a `while` statement continues to execute as long as a specified condition is met.

The general syntax for the `while` statement is as follows:

```
while (condition) {
    code to repeatedly execute while expression continues to
    evaluate to true
}
```

The condition can be either a simple or a complex logical expression that evaluates to a `true` or `false` value, for example:

```
int x = 1;
int y = 1;

while ((x < 20) || (y < 10)) {
    hopefully we'll do something within this loop body that increments the
    value of
    either x or y, to avoid an infinite loop!
}
```

When program execution reaches a `while` statement, the condition is evaluated first. If the condition is `true`, the block of code following the condition is executed. When the block of code is finished, the condition is evaluated again, and if it's still `true`, the process repeats itself until the condition evaluates to `false`, at which point the `while` loop terminates.

Here's a simple example illustrating the use of a `while` loop to print consecutive integer values from 0 to 3. A `boolean` variable named `finished` is initially set to `false`. The `finished` variable is used as a flag: as long as `finished` is `false`, the block of code following the `while` loop will continue to execute. When the value of `i` reaches 4, the `finished` flag will get set to `true`, at which point the `while` loop will terminate:

```
public class WhileDemo {
    public static void main(String[] args) {
        boolean finished = false;
        int i = 0;

        while (!finished) {
            System.out.println(i);
            i++;
            if (i == 4) {
     finished = true;  // toggle the flag to terminate the loop
            }
        }
    }
}
```

Here's the output:

```
0
1
2
3
```

As with the other flow control structures, if only one statement is specified after the condition, the braces can be omitted:

```
while (x < 20) x = x * 2;
```

or alternatively

```
while (x < 20)
    x = x * 2;
```

But it's considered good programming practice to use braces regardless:

```
while (x < 20) {
    x = x * 2;
}
```

If you always want the loop to execute at least once before the end condition is tested, use a do … while statement instead:

```
do {
    whatever is to be executed at least once;
} while (completion test);
```

Jump Statements

The Java language defines two **jump statements** that are used to redirect program execution to another statement elsewhere in the code. The two types of jump statement are the `break` and `continue` statements.

You've already seen `break` statements in action earlier in this chapter, when they were used in conjunction with `switch` statements. A `break` statement can also be used to abruptly terminate `for` or `while` loops. When a `break` statement is encountered during loop execution, the loop immediately terminates, and program execution is transferred to the line of code immediately following the loop, as in the following example:

```
// We start out with the intention of incrementing x from 1 to 4 ...
for (int x = 1; x <= 4; x++) {
    // ... but when x reaches the value 3, we prematurely terminate this
    // loop with a break statement.
    if (x == 3) {
       break;
    }

    System.out.println(x);
}

System.out.println("Loop finished");
```

The output produced by this code would be as follows:

```
1
2
Loop finished
```

A `continue` statement, on the other hand, is used to exit from the ***current iteration*** of a loop ***without*** terminating ***overall*** loop execution. That is, a `continue` statement transfers program execution back up to the top of the loop without finishing the particular iteration that is already in progress; the loop counter is incremented, in the case of a `for` loop, and execution continues.

Let's look at the same example as before, but we'll replace the `break` statement with a `continue` statement:

```
// We start out with the intention of incrementing x from 1 to 4 ...
for (int x = 1; x <= 4; x++) {
    // ... but when x reaches the value 3, we prematurely terminate
    // this iteration of the loop (only) with a continue statement.
    if (x == 3) {
       continue;
    }

    System.out.println(x);
}

System.out.println("Loop finished");
```

The output produced by this code would be as follows:

```
1
2
4
Loop finished
```

Block-Structured Languages and the Scope of a Variable

Java is a **block-structured language**. As mentioned earlier in the chapter, a "block" of code is a series of zero or more lines of code enclosed within braces { ... }.

Blocks may be nested inside one another to any arbitrary depth, as illustrated by the following code example:

```
public class SimpleProgram {
    // We're inside of the "class" block (one level deep).
    public static void main(String[] args) {
        // We're inside of the "main method" block (two levels deep).
        int x = 3;
        int y = 4;
        int z = 5;

        if (x > 2) {
            // We're now one level deeper (level 3), in an "if" block.
            if (y > 3) {
                // We're one level deeper still (level 4), in a
                // nested "if" block.
                // (We could go on and on!)
            } // We've just ended the level 4 block.
            // (We could have additional code here, at level 3.)
        } // Level 3 is done!
        // (We could have additional code here, at level 2.)
    } // That's it for level 2!
    // (We could have additional code here, at level 1.)
} // Adios, amigos! Level 1 (the "class" block) has just ended.
```

Variables can be declared in any block within a program. The **scope** of a variable is that portion of code in which the variable can be referenced by name—specifically, from the point where the variable name is first declared down to the closing (right) brace for the block of code in which it was declared. A variable is said to be **in scope** as long as the compiler recognizes its name. Once program execution exits a block of code, any variables that were declared inside that block go **out of scope** and will be inaccessible to the program; the compiler effectively forgets that the variable was ever declared.

As an example of the consequences of variable scope, let's look at a simple program called ScopeExample. This program makes use of three nested code blocks: one for the class body, one for the main method body, and one as part of an if statement inside the body of the main method. We in turn declare two variables–x, in the main code block (at level 2), and y, in the if block (level 3):

```
public class ScopeExample { // Start of block level 1.
  public static void main(String[] args) { // Start of block level 2.
    double x = 2.0;  // Declare "x" at block level 2.

    if (x < 5.0) { // Start of block level 3.
      double y = 1.0;  // Declare "y" inside block level 3.
      System.out.println("The value of x = " + x);   // x, declared at
                                                      level 2, is
                                                      // still in scope at
                                                      level 3.
      System.out.println("The value of y = " + y);
    } // Variable "y" goes out of scope when the "if" block (level 3) ends.

    // "y" has gone out of scope, and is no longer recognized by the
       compiler.
    // If we try to reference "y" in a subsequent statement, the
       compiler will
    // generate an error. "x", on the other hand, remains in scope until
       the main
    // method block (level 2) ends.

    System.out.println("The value of x = " + x);    // This will compile.
    System.out.println("The value of y = "  + y);   // This WON'T compile.
  } // Variable "x" goes out of scope when the main method block (level
       2) ends.
}
```

In the preceding example, variable y goes out of scope as soon as the if block ends. If we try to access y later in the program, as we do in the bolded line of the preceding code, the compiler will generate the following error message:

```
cannot resolve symbol
symbol : variable y
System.out.println("The value of y = "  + y);
                                          ^
```

Note that a given variable is accessible to any nested ***inner*** code blocks that follow its declaration. For example, in the preceding `ScopeExample` program, variable `x`, declared at the `main` method block level (level 2), is accessible inside the `if` statement code block (level 3).

Printing to the Console Window

Most applications communicate information to users by displaying messages via an application's graphical user interface. However, it's also useful at times to be able to display simple text messages to the console window from which we're running a program as a quick and dirty way of verifying that a program is working properly.

To print text messages to the screen, we use the following syntax:

```
System.out.println(expression to be printed);
```

The `System.out.println` method can accept very complex expressions and does its best to ultimately turn these into a single `String` value, which then gets displayed on the screen. Here are a few examples:

```
System.out.println("Hi!");          // Printing a String literal/constant.

String s = "Hi!";
System.out.println(s);              // Printing the value of a String
                                       variable.

String t = "foo";
String u = "bar";
System.out.println(t + u + "!");    // Using the String concatenation
                                       operator (+)
                                    // to print "foobar!".
int x = 3;
int y = 4;
```

```
System.out.println(x);              // Prints the String
representation of the
                                    // integer value 3 to the screen.
System.out.println(x + y);          // Computes the sum of x and y, then
                                       prints the
                                    // String representation of the
                                       integer value 7
                                    // to the screen.
```

Note that on the last line of code, the plus sign (+) is interpreted as the ***addition*** operator, not as the `String` ***concatenation*** operator, because it separates two variables that are both declared to be of type `int`. So the sum of 3 + 4 is computed to be 7, which is then printed. In the next example, however, we get different (and arguably undesired) behavior:

```
System.out.println("The sum of x plus y is: " + x + y);
```

The preceding line of code causes the following to be printed:

```
The sum of x plus y is: 34
```

Why is this? Recall that we evaluate expressions from left to right, and so since the first of the two plus signs separates a `String` literal and an `int`

```
System.out.println("The sum of x plus y is: " + x + y);
```

the first plus sign is interpreted as a `String` concatenation operator, producing the intermediate `String` value `"The sum of x plus y is: 3"`. The second plus sign separates this intermediate `String` value from an `int`, and so the second plus sign is ***also*** interpreted as a `String` concatenation operator, producing the final `String` value `"The sum of x plus y is: 34"`, which is printed to the command window.

To print the ***correct*** sum of `x` and `y`, we must force the second plus sign to be interpreted as an integer addition operator by enclosing the addition expression in nested parentheses:

```
System.out.println("The sum of x plus y is: " + (x + y));
```

The nested parentheses cause the innermost expression to be evaluated first, thus computing the sum of `x + y` properly. Hence, this `println` statement displays the correct message on the screen:

```
The sum of x plus y is: 7
```

When writing code that involves complex expressions, it's a good idea to use parentheses liberally to make your intentions clear to the compiler. Extra parentheses, when used correctly, never hurt!

print vs. println

When we call the `System.out.println` method, whatever expression is enclosed inside the parentheses will be printed, followed by a (platform-dependent) **line terminator**. The following code snippet

```
System.out.println("First line.");
System.out.println("Second line.");
System.out.println("Third line.");
```

produces three separate lines of output:

```
First line.
Second line.
Third line.
```

In contrast, the System.out.print() method

```
System.out.print(expression to be printed);
```

causes whatever expression is enclosed in parentheses to be printed ***without*** a trailing line terminator. Using `print` in combination with `println` allows us to generate a ***single*** line of output across a ***series*** of Java statements, as shown by the following example:

```
System.out.print("J");        // Using print here.
System.out.print("AV");       // Using print here.
System.out.println("A!!!");   // Note use of println as the last statement.
```

This code snippet produces a single line of output:

```
JAVA!!!
```

When a single print statement gets too long to fit on a single line, as in this example

```
// Pseudocode.
statement;
another statement;
System.out.println("Here's an example of a single print statement
that is very long ... SO long that it wraps around and makes the
program listing difficult to read.");
yet another statement;
```

we can make a program listing more readable by breaking up the contents of such a statement into multiple concatenated `String` expressions and then breaking the statement along plus-sign boundaries:

```
// Pseudocode.
statement;
another statement;
System.out.println("Here's an example of how " +
                   "to break up a long print statement " +
                   "with plus signs.");
yet another statement;
```

Even though the preceding `System.out.println` call is broken across three lines of code, it will be printed as a single line of output:

```
Here's an example of how to break up a long print statement with
plus signs.
```

Escape Sequences

Java defines a number of **escape sequences** so that we can represent special characters, such as newline and tab characters, within `String` or `char` literals. The most commonly used escape sequences are listed in Table 2-5.

Table 2-5. *Java Escape Sequences*

Escape Sequence	Description
`\n`	Newline
`\b`	Backspace
`\t`	Tab
`\\`	Backslash
`\'`	Single quote, used within single quotes (e.g., `char singleQuote = '\'';`)
`\"`	Double quote, used within double quotes (e.g., `String doubleQuote = "\"";`)

One or more escape sequences can be included in the expression that is passed to the `print` and `println` methods. For example, consider the following code snippet:

```
System.out.println("One ...");
System.out.println("\t... two ...");
System.out.println("\t\t... three ... \"WHEEE!!!\"");
```

When the preceding code is executed, the following output is displayed:

```
One ...
    ... two ...
        ... three ... "WHEEE!!!"
```

The second and third lines of output have been indented one and two tab positions, respectively, by virtue of the use of `\t`, and the expression `"WHEEE!!!"` is printed enclosed in double quotes because of our use of `\"`.

Elements of Java Style

One of the trademarks of good programmers is that they produce ***human-readable*** code, so that their colleagues will be able to work with and modify their programs. The following sections present some guidelines and conventions that will help you produce clear, readable Java code.

Proper Use of Indentation

One of the best ways to make a Java program readable is through proper use of indentation to clearly delineate its block structure. Statements within a block of code should be indented relative to the starting/ending line of the enclosing block (i.e., indented relative to the lines carrying the braces). The standard recommendation is to use four spaces (note that some of the examples in this book vary from that standard).

To see how indentation can make a program readable, consider the following two programs. In the first simple program, proper indentation is used:

```
public class IndentationExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 4; i++) {
            System.out.print(i);

            if ((i == 2) || (i == 4))  {
                if (i == 2) {
                    System.out.print("Two ");
                }
                else {
                    System.out.print("Four ");
                }
                System.out.println("is an even number");
            }
            else if ((i == 1) || (i == 3))  {
                if (i == 1) {
                    System.out.print("One ");
                }
```

```
                else {
                    System.out.print("Three ");
                }
                System.out.println("is an odd number");
            }
        }
    }
}
```

And it's relatively easy to see how the following output would be produced:

```
1 is an odd number
2 is an even number
3 is an odd number
4 is an even number
```

Now let's remove all indentation from the program:

```
public class IndentationExample {
public static void main(String[] args) {
for (int i = 1; i <= 4; i++) {
System.out.print(i);

if ((i == 2) || (i == 4))  {
if (i == 2) {
System.out.print("Two ");
}
else {
System.out.print("Four ");
}
System.out.println("is an even number");
}
else if ((i == 1) || (i == 3))  {
if (i == 1) {
System.out.print("One ");
}
```

```
else {
System.out.print("Three ");
}
System.out.println("is an odd number");
}
}
}
}
```

Both versions of this program are understood by the Java compiler, and both produce the same output when executed, but the first version is much more readable to a human being.

Failure to properly indent not only makes programs unreadable but also makes them harder to debug, particularly if a compilation error arises due to unbalanced/missing braces. In such a situation, the compilation error message often gets reported on a line much later in the program than where the problem actually exists. For example, the following program is missing an opening/left brace on line 9, but the compiler doesn't report an error until line 23:

```
public class IndentationExample2 {
    public static void main(String[] args) {
        int x = 2;
        int y = 3;
        int z = 1;

        if (x >= 0) {
            if (y > x) {
                if (y > 2) // we're missing a left brace here on line 9,
                but ...
                    System.out.println("A");
                    z = x + y;
                }
                else {
                    System.out.println("B");
                    z = x - y;
                }
            }
```

```
            else {
                System.out.println("C");
                z = y - x;
            }
        }
        else System.out.println("D");  // ... compiler first complains here!
                                       // (line 23)
    }
}
```

What's even worse, the error message that the compiler generates in such a situation can be rather cryptic. In this particular example, the compiler (incorrectly) points to line 23 as the problem, with a misleading error message:

```
IndentationExample2.java:23: illegal start of type
else System.out.println("D");
^
```

This error message doesn't help us locate the ***real*** problem on line 9. However, at least we've properly indented our code, and so it will likely be far easier to hunt down the missing brace than it would be if our indentation were sloppy or nonexistent.

> If ever you get a compilation error that makes absolutely no sense whatsoever, consider looking earlier in the program for missing punctuation—that is, a missing brace, parenthesis, or semicolon!

Sometimes, we have so many levels of nested indentation, or individual statements are so long, that lines wrap when viewed in an editor or printed as hard copy:

```
while (a < b) {
    while (c > d) {
        for (int j = 0; j < 29; j++) {
```

```
            x = y + z + a - b + (c * (d / e) + f) -
            g + h + j - l - m - n + o + p * q / r + s;
        }
    }
}
```

To avoid this, it's best to break the line in question along white space or punctuation boundaries, indenting continuation lines relative to the start of the line:

```
while (a < b) {
    while (c > d) {
        for (int j = 0; j < 29; j++) {
            // This is cosmetically preferred.  Note indentation
            // of continuation lines.
            x = y + z + a - b + (c * (d / e) + f) -
                g + h + j - l - m - n + o + p *
                q / r + s;
        }
    }
}
```

Use Comments Wisely

Another important feature that makes code more readable is the liberal use of meaningful comments. Always keep in mind when writing code that ***you*** may know what you're trying to do, but someone else trying to read your code may not. (Actually, we sometimes even need to remind ***ourselves*** of why we did what we did if we haven't looked at code that we've written in a while.)

Here are some basic rules of thumb:

- If there can be any doubt as to what a passage of code does, precede it with a comment.
- Indent each comment to the same level as the block of code or statement to which it applies.

Placement of Braces

For block-structured languages that use braces to delimit the start/end of blocks, there are two general schools of thought as to where the ***left/opening*** brace of a code block should be placed.

The first style is to place an opening brace at the ***end*** of the line of code that starts a given block. Each closing brace goes on its own line, aligned with the ***first*** character of the line containing the opening brace:

```
public class Test {
    public static void main(String[] args) {
        for (int i = 0; i < 3; i++) {
            System.out.println(i);
        }
    }
}
```

An alternative brace placement style is to place every opening brace on a line by itself, aligned with the immediately preceding line. Each closing brace goes on its own line as before, aligned with the corresponding opening brace:

```
  public class Test
  {
      public static void main(String[] args)
      {
          for (int i = 0; i < 3; i++)
          {
              System.out.println(i);
          }
      }
  }
```

Either approach to left/opening brace placement is fine; the first of the two approaches produces code listings that are a bit more compact (i.e., contain less white space) and is the more popular of the two styles. It's a good practice to maintain consistency in your code, however, so pick whichever brace placement style you prefer and stick with it.

Descriptive Variable Names

As with indentation and comments, the goal when choosing variable names is to make a program as readable, and hence self-documenting, as possible. Avoid using single letters as variable names, except for loop control variables (or as parameters to methods, discussed later in the book).

Abbreviations should be used sparingly and only when the abbreviation is commonly used and widely understood by developers. Consider the following variable declaration:

```
int grd;
```

It's not completely clear what the variable name `grd` is supposed to represent. Is the variable supposed to represent a grid, a grade, or a gourd? A better practice would be to spell out the entire word:

```
int grade;
```

At the other end of the spectrum, names that are too long, such as perhaps

```
double averageThirdQuarterReturnOnInvestment;
```

can make a code listing overwhelming to anyone trying to read it. It can sometimes be challenging to shorten a variable name while still keeping it descriptive, but do try to keep the length of your variable names within reason, for example:

```
double avg3rdQtrROI;
```

Summary

In this chapter, you've learned some basic aspects of the Java language. In particular, we covered

- The architecture-neutral nature of Java
- The anatomy of a simple Java program
- The mechanics of how to compile and execute a Java program from the command line
- The eight primitive Java types and the `String` type

- How variables of these types are declared and initialized
- How an expression of one type can be cast into a different type and when it's necessary to do so
- Arithmetic, assignment, logical, and relational expressions and operators
- Loops and several other flow control structures available with Java
- How to define blocks of code and the concept of variable scope
- How to print text messages with the `System.out.println` and `System.out.print` methods
- Some basic elements of good Java programming style

There's a lot more to learn about Java—things you'll need to know in building the SRS application in Part 3 of the book—but I need to explain a number of basic object concepts first. So on to Chapter 3!

EXERCISES

1. Compare what you've learned about Java so far to another programming language that you're already familiar with. What is similar about the two languages? What is different?
2. [*Coding*] Create a program that will print the even numbers from 2 to 10 to the command window using (a) a `for` loop and a `continue` statement and (b) a `while` loop and a `boolean` variable as a flag.
3. Given the following initial variable declarations and value assignments, evaluate the expression on the last line of code manually, without using the Java compiler:

```
int a = 1;
int b = 1;
int c = 1;
((((c++ + --a) * b) != 2) && true)
```

Using nested for or if loops, write a program called Stars.java that produces the following output:

```
1: *
2: **
3: ***
4: ****
5: *****
```

CHAPTER 3

Objects and Classes

Objects are the fundamental building blocks of an object-oriented application. Just as you learned in Chapter 1 that abstraction involves producing a model of the real world, you'll see in this chapter that objects are "mini abstractions" of the various real-world elements that comprise such a model.

In this chapter, you'll learn

- The advantages of an object-oriented approach to software development as compared with the "traditional" non-OO approach
- How to use classes to specify an object's data and behavior
- How to create objects at run time
- How to declare **reference variables** to refer to objects symbolically within a program
- How objects keep track of one another in memory

Software at Its Simplest

At its simplest, every software application consists of two primary components: ***data*** and ***functions*** that operate on (i.e., input, output, calculate, store, retrieve, print, etc.) that data. (See Figure 3-1.)

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_3

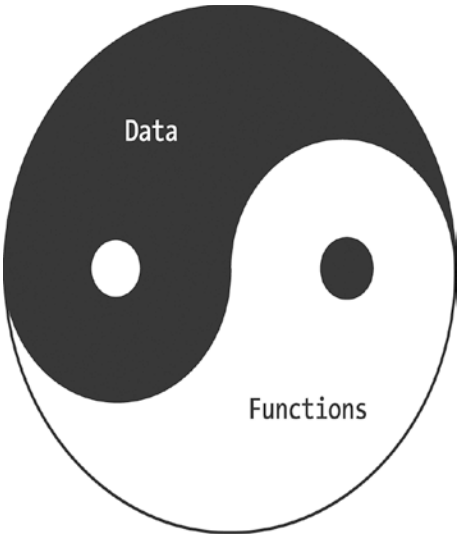


Figure 3-1. *At its simplest, software consists of data and functions that operate on that data*

The pre-OO way of designing software was known as (**top-down**) **functional decomposition**. Let's compare the functional decomposition approach of designing software with the OO approach.

Functional Decomposition

With functional decomposition, we started with a statement of the overall function that a system was expected to perform—for example, "Student Registration." We then broke that function down into subfunctions:

- "Add a New Course to the Course Catalog"
- "Allow a Student to Enroll in a Course"
- "Print a Student's Class Schedule"
- And so forth

We next decomposed those functions into smaller subfunctions. For example, we might decompose "Allow Student to Enroll in a Course" into

- "Display List of Available Courses"
- "Allow Student to Select a Course"
- "Verify That Student Has Met All Prerequisites"
- And so forth

We kept decomposing functions into smaller and smaller logical pieces until we could reasonably subdivide no further, as illustrated in Figure 3-2.

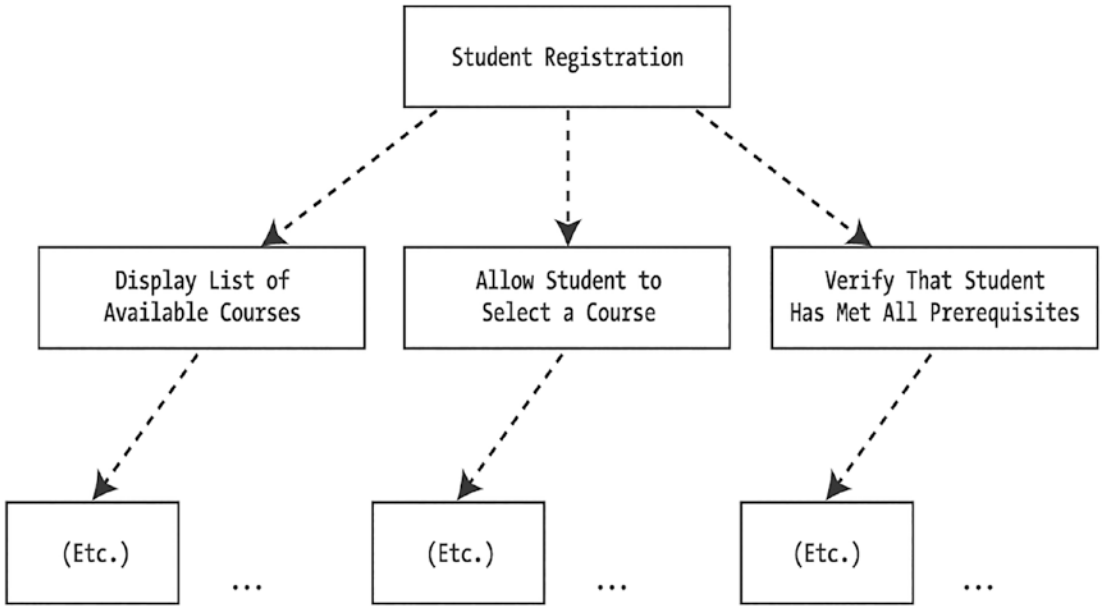


Figure 3-2. *We functionally decomposed an application from the top down…*

We then assigned the lowest-level functions to different programmers to code and **unit test** (i.e., test in isolation). Finally, we assembled these functions in modular fashion from the bottom up, testing the results of each successive stage in the integration process, until we had a complete application built, as illustrated in Figure 3-3.

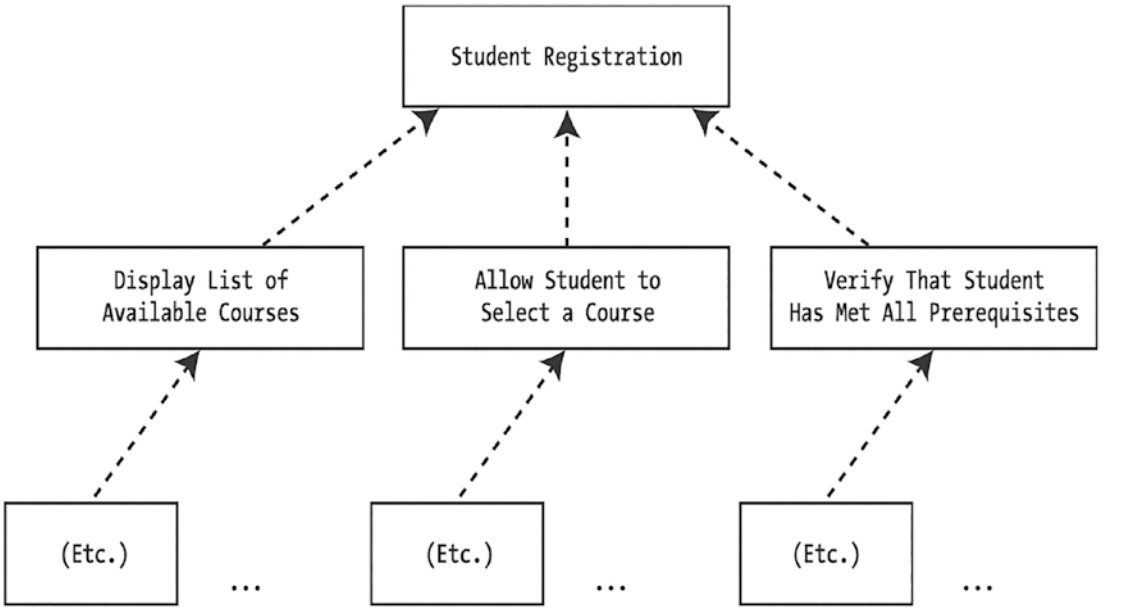


Figure 3-3. *...and assembled the application from the bottom up*

With the functional decomposition approach to software development, our primary focus was on the ***functions*** that an application was to perform; ***data*** was an afterthought. That is,

- Data was passed around from one function to the next, like a car being manufactured via an assembly line process in an automotive plant.
- Data structure thus had to be understood in ***many*** places (i.e., by many functions) throughout an application.
- If an application's data structure had to change after the application was deployed, ***major*** "ripple effects" often arose throughout the application. One of the most dramatic examples of a ripple effect due to a change in data structure was the Y2K crisis, wherein a seemingly simple change in date formats—from a two- to four-digit year—caused a worldwide panic! ***Billions*** of dollars were spent on trying to find and repair what were expected to be disastrous ripple effects before the clock struck midnight on January 1, 2000.

- Despite our best efforts to test an application thoroughly before deploying it, bugs always manage to creep through undetected. If data integrity errors arose as a result of faulty logic after an application had been fully integrated, it was often very difficult to pinpoint precisely where—that is, ***in which specific function(s)***—the error might have occurred, because the data had been passed from function to function so many times.

The Object-Oriented Approach

As you'll see over the next several chapters, the OO approach to software development remedies the vast majority of these shortcomings.

- With OO software development, we focus on designing the application's ***data structure*** first and its functions second.
- Data is ***encapsulated*** inside of objects; thus, data structure has to be understood only by the object to which the data belongs.
- If an object's data structure has to change after the application has been deployed, there are virtually ***no ripple effects***; only the internal logic of the affected object must change.
- ***Each object is responsible for ensuring the integrity of its own data***. Thus, if data integrity errors arise within a given object's data, we can pretty much assume that it was the object ***itself*** that allowed this to happen, and we can therefore focus on that object's internal functional logic to isolate the "bug."

What Is an Object?

Before we talk about software objects, let's talk about real-world objects in general. According to Merriam-Webster's Collegiate Dictionary, an object is "(1) something material that may be perceived by the senses; (2) something mental or physical toward which thought, feeling, or action is directed."

The first part of this definition refers to objects as we typically think of them: as physical "things" that we can see and touch and that occupy space. Because we intend to use the Student Registration System (SRS) case study as the basis for learning about objects throughout this book, let's think of some examples of **physical objects** that make sense in the general context of an academic setting, namely

- The ***students*** who attend classes
- The ***professors*** who teach the students
- The ***classrooms*** in which class meetings take place
- The ***audiovisual equipment*** in these classrooms
- The ***buildings*** in which the classrooms are located
- The ***textbooks*** students use
- And so forth

Of course, while all of these types of objects are commonly found on a typical college campus, not all of them are relevant to registering students for courses, nor are they all necessarily called out by the SRS case study, but we won't worry about that for the time being. In Part 2 of this book, you'll learn a technique for using a requirements specification as the basis for identifying which types of objects are relevant to a particular abstraction.

Now, let's focus on the second half of the definition, particularly on the phrase ***"something mental… toward which thought, feeling, or action is directed."*** There are a great many **conceptual objects** that play important roles in an academic setting; some of these are

- The **courses** that students attend
- The **departments** that faculty work for
- The **degrees** that students receive

And, of course, there are many others. Even though we can't see, touch, taste, smell, or hear them, conceptual objects are every bit as important as physical objects in an overall abstraction.

From a software perspective, a (software) **object** is a software construct/module that bundles together **state** (**data**) and **behavior** (**functions**), which, taken together, represent an abstraction of a real-world (physical or conceptual) object. This is illustrated conceptually in Figure 3-4.

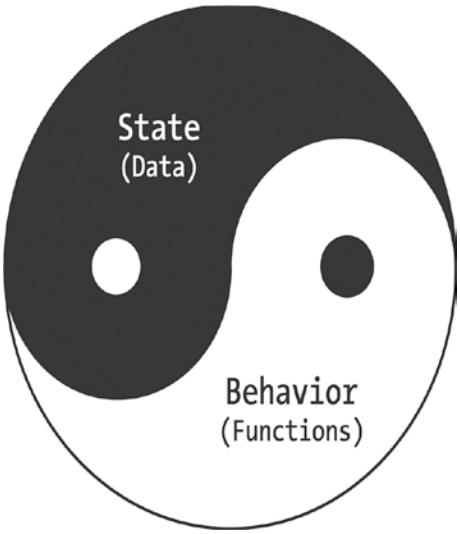


Figure 3-4. *A software object bundles state (data) and behavior (functions)*

Let's explore the two sides of objects—their state and behavior—separately, in more depth.

State/Data/Attributes

If we wish to record information about a student, what data might we require? Some examples might be

- The student's name
- Their student ID number
- The student's birth date
- Their address
- The student's designated major field of study
- Their cumulative grade point average (a.k.a. GPA)
- Whom the student's faculty advisor is
- A list of the courses that the student is currently enrolled in this semester (i.e., the student's current ***course load***)

- A history of all of the courses that the student has taken to date, the semester/year in which each was taken, and the grade that was earned for each (i.e., the student's ***transcript***)
- And so on

Now, how about for an academic course? Perhaps we'd wish to record

- The course number (e.g., "ART 101")
- The course name (e.g., "Introductory Basketweaving")
- A list of all of the courses that must have been successfully completed by a student prior to registering for this course (i.e., the course's ***prerequisites***)
- The number of credit hours that the course is worth
- A list of the professors who have been approved to teach this course
- And so on

In object nomenclature, the data elements used to describe an object are referred to as the object's **attributes**.

An object's attribute values, when taken collectively, are said to define the **state**, or condition, of the object. For example, if we wanted to determine whether or not a student is "eligible to graduate" (a ***state***), we might look at a combination of the following

- The student's transcript (an ***attribute value***)
- The list of courses the student is currently enrolled in (a second ***attribute value***)

to determine if the student indeed is expected to have satisfied the course requirements for their chosen major field of study (a third attribute value) by the end of the current academic year.

A given attribute may be simple—for example, GPA, which can be represented as a simple floating-point number (perhaps a `double` in Java)—or complex, for example, "transcript," which represents a rather extensive collection of information with no simple representation.

Programmers new to the object paradigm often ask, "Why not represent a transcript as a `String`? A ***long*** `String`, no doubt, but a `String` nonetheless?" You'll learn over successive chapters that there is a far more elegant way to represent the notion of a student's transcript in object-oriented terms.

Behavior/Operations/Methods

Now, let's revisit the same two types of object—a student and a course—and talk about these objects' respective behaviors.

A student's behaviors (relevant to academic matters, at any rate!) might include

- Enrolling in a course
- Dropping a course
- Designating a major field of study
- Selecting a faculty advisor
- Telling us their GPA
- Telling us whether or not they have taken a particular course and, if so, when the course was taken, which professor taught it, and what grade the student received

It's a bit harder to think of an inanimate, conceptual object like a course as having behaviors, but if we were to imagine a course to be a living thing, we could envision that a course's behaviors might include

- Permitting a student to register
- Determining whether or not a given student is already registered
- Telling us how many students have registered so far or, conversely, how many seats remain before the course is full
- Telling us what its prerequisite courses are
- Telling us how many credit hours the course is worth

- Telling us which professor is assigned to teach the course this semester
- And so on

When we talk about software objects specifically, we define an object's behaviors, also known as its **operations**, as both the things that an object does to ***access*** its attribute values (data) and the things that an object does to ***modify/maintain*** its attribute values (data).

If we take a moment to reflect back on the behaviors we expect of a student as listed previously, we see that each operation involves one or more of the student's attributes, for example:

- Telling you their GPA involves ***accessing*** the value of the student's "GPA" attribute.
- Choosing a major field of study involves ***modifying*** the value of the student's "major field of study" attribute.
- Enrolling in a course involves ***modifying*** the value of the student's "course load" attribute.

Since you recently learned that the collective set of attribute values for an object defines its state, you now can see that operations are capable of ***changing an object's state***. Let's say that we define the state of a student who hasn't yet selected a major field of study as an "undeclared" student. Asking such a student object to perform its "choosing a major field of study" method will cause that object to update the value of its "major field of study" attribute to reflect the newly selected major field. This, then, changes the student's state from "undeclared" to "declared."

Yet another way to think of an object's operations are as ***services*** that can be requested of the object on behalf of the application. For example, one service that we might ask a course object to perform is to provide us with a list of all of the students who are currently registered for the course (i.e., a ***student roster***).

When we actually get around to programming an object's behaviors in a language like Java, we refer to the programming language representation of an operation as a **method**, whereas, strictly speaking, the term "operation" is typically used to refer to a behavior conceptually.

What Is a Class?

A **class** is an abstraction describing the common features of all objects in a group of similar objects. For example, a class called "Student" could be created to describe all student objects recognized by the SRS.

A class defines

- The ***data structure*** (i.e., the names and types of ***attributes***) of each and every object belonging to that class
- The operations/***methods*** to be performed by such objects: specifically, what these operations are, how an object is formally called upon to perform them, and what behind-the-scenes actions an object has to take to actually carry them out

For example, the `Student` class might be designed to have the nine attributes listed in Table 3-1.

Table 3-1. *Proposed Attributes of the Student Class*

Attribute	Java Type
`name`	`String`
`studentId`	`String`
`birthDate`	`Date`
`address`	`String`
`major`	`String`
`gpa`	`Double`
`advisor`	`???`
`courseLoad`	`???`
`transcript`	`???`

Note that many of the `Student` attributes can be represented by built-in Java types (e.g., `String`, `double`, and `Date`) but that a few of the attributes—`advisor`, `courseLoad`, and `transcript`—are too complex for built-in Java types to handle. You'll learn how to tackle such attributes a bit later on in the book.

In terms of operations, the `Student` class might define five methods whose names are as follows:

- `registerForCourse`
- `dropCourse`
- `chooseMajor`
- `changeAdvisor`
- `printTranscript`

You'll learn how to formally declare methods in Java in Chapter 4.

Note that an object can only do those things for which methods have been defined by the object's class. In that respect, an object is like an appliance: it can do whatever it was designed to do (a DVD player provides buttons to play, pause, stop, and seek a particular movie scene) and nothing more (you can't ask a DVD to toast a bagel—at least not with much chance of success!). So an important aspect of successfully designing an object is making sure to anticipate all of the behaviors it will need to perform in order to carry out its "mission" within the system. You'll learn how to formally determine what an object's mission, data structure, and behaviors should be, based on the overall requirements for the application that it is to support, in Part 2 of the book.

The term **feature** is used to collectively refer to both the attributes and methods of a class. That is, a class definition that includes nine attribute declarations and five method declarations is said to have 14 features.

A Note Regarding Naming Conventions

All object-oriented programming languages (OOPLs), including Java, uphold the following naming conventions:

- When naming classes, we begin with an ***uppercase*** letter, but use mixed case for the name overall: `Student`, `Course`, `Professor`, and so on. When the name of a class would ideally be stated as a multiword phrase, such as "course catalog," we start each word with a capital letter and concatenate the words without using spaces, dashes, or underscores to separate them—for example, `CourseCatalog`. This style is known as **Pascal casing**.

- The convention for attribute and method names is to start with a ***lowercase*** letter, but to capitalize the first letter of any subsequent words in the name. Typical attribute names might thus be `name`, `studentId`, or `courseLoad`, while typical method names might thus be `registerForCourse` and `printTranscript`. This style is known as **camel casing**.

Declaring a Class, Java Style

Once we've determined what common data structure and behaviors we wish to impart to a set of similar objects, we must formally declare them as attributes and methods in the context of a Java class. For example, we'd program the `Student` class data structure as presented in Table 3-1 as follows:

```
public class Student {
    // Attribute declarations typically appear first in a class
    declaration ...
    String name;
    String studentId;
    Date birthDate;
    String address;
    String major;
    double gpa;
    // type? advisor – we'll declare this attribute later
    // type? courseLoad - ditto
    // type? transcript - ditto

    // ... followed by method declarations (details omitted – you'll
    // learn how to program methods in Java in Chapter 4.)
}
```

As with all Java class definitions that you've seen thus far in the book, this class definition would reside in a source file named *ClassName*`.java` (`Student.java`, to be specific) and would be subsequently compiled into bytecode form as a file named `Student.class`. Note that the reserved word public appears in the class declaration. We will be discussing the notion of public visibility later in the book.

Note that, for the preceding code to compile, we'd need to insert the statement

```
import java.util.Date;
```

ahead of the declaration

```
public class Student { ... }
```

We'll discuss `import` directives in Chapter 6.

Note that our `Student` class is not required to declare a `main` method. Unlike the classes shown previously in the book, which served to encapsulate a program's `main` method

```
public class Simple {
    public static void main(String[] args) {
        System.out.println("I love Java!!!");
    }
}
```

the Student class serves a different purpose: namely, we're defining what the data structure and behaviors of `Student` objects should be. Because the `Student` class does not contain a main method, it makes no sense to try to execute its bytecode with the JVM–typing

```
java Student
```

results in the error message

```
Error: main method not found in class Student, please define the main
method as:
public static void main(String [] args)
```

Instantiation

A class definition may be thought of as a template for creating software objects—a "pattern" used to

- Allocate a prescribed amount of memory within the JVM to house the ***attributes*** of a new object
- Associate a certain set of ***behaviors*** with that object

The term **instantiation** is used to refer to the process by which an object is created in memory at run time based upon a class definition. From a single class definition—for example, `Student`—we can create many objects with identical data structures and behaviors, in the same way that we use a single cookie cutter to make many cookies all of the same shape. Another way to refer to an object, then, is as an **instance** of a particular class—for example, "A `Student` object is an instance of the `Student` class." (We'll talk about the actual process of instantiating objects as it occurs in Java in a bit more detail later in this chapter.)

Classes may thus be differentiated from objects as follows:

- A ***class*** defines the features—attributes, methods—that every object belonging to the class must possess; a class can thus be thought of as serving as an ***object template***, as illustrated in Figure 3-5.

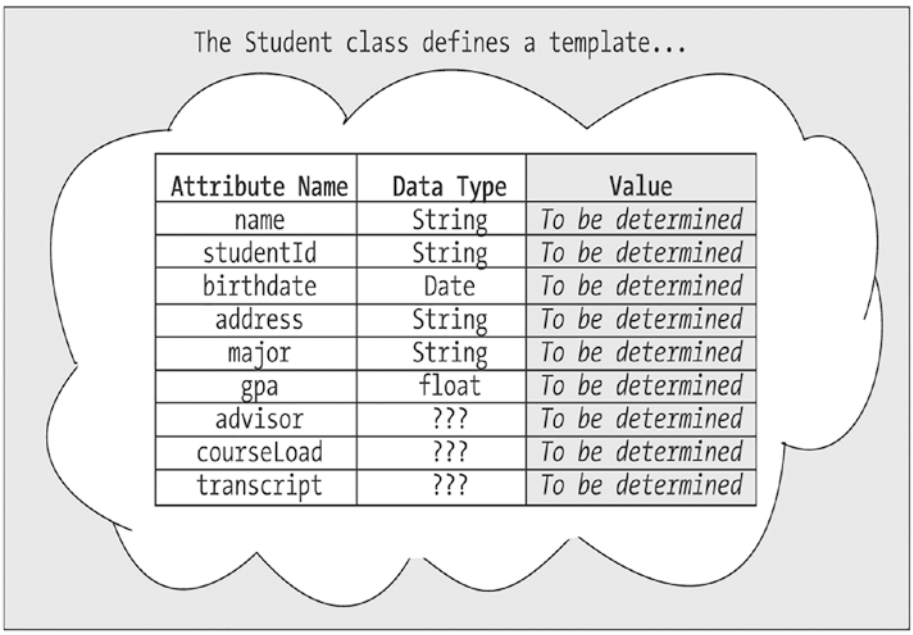


Figure 3-5. *A class prescribes a template for instantiating objects…*

- An ***object***, on the other hand, is a unique instance of a ***filled-in template*** for which attribute values have been provided and upon which methods may be performed, as illustrated in Figure 3-6.

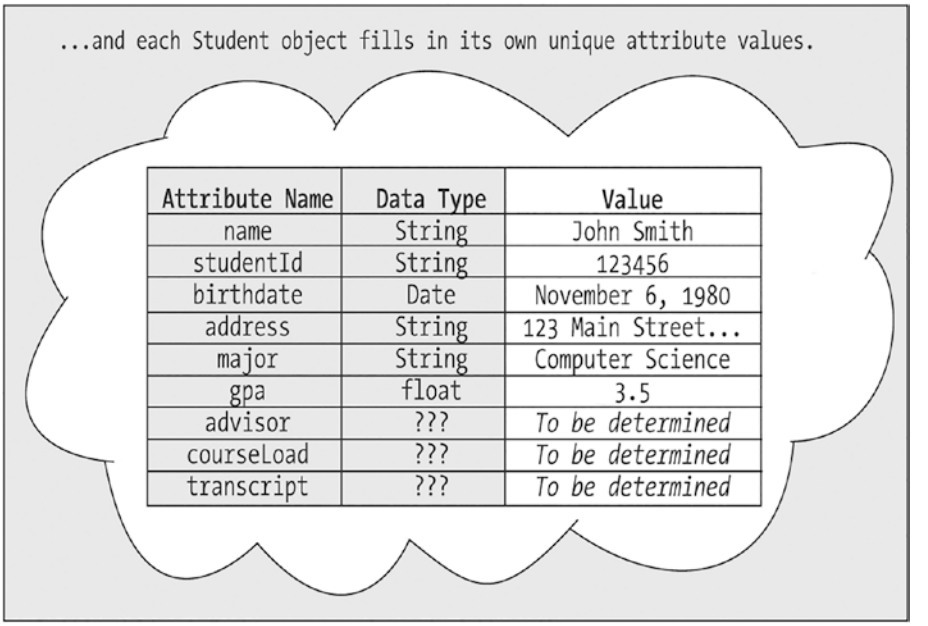


Figure 3-6. *... and an object then fills in its own unique attribute values*

Encapsulation

Encapsulation is a formal term referring to the mechanism that bundles together the state and behavior of an object into a single logical unit. Everything that we need to know about a given student is, in theory, contained within the boundaries of a `Student` object, either

- Directly, as an attribute of that object
- Indirectly, as a method that can answer a question or make a determination about the object's state

---

Encapsulation isn't unique to OO languages, but in some senses it is perfected by them. If you're familiar with C, you know that a C `struct`(ure) encapsulates data

```
struct employee {
    char name[30];
    int age;
```

```
}
```

and a C function encapsulates logic—data is passed in and operated on, and an answer is optionally returned:

```
float average(float x, float y) {
    return (x + y)/2.0;
}
```

But only with OO programming languages is the notion of encapsulating data and behavior in a single class construct, to represent an abstraction of a real-world entity, truly embraced.

User-Defined Types and Reference Variables

In a non-OO programming language such as C, the statement

```
int x;
```

is a **declaration** that variable `x` is an `int`(eger), one of several primitive data types defined to be part of the C language. What does this ***really*** mean? It means that

- `x` is a ***symbolic name*** that we've invented to refer to an `int`(eger) value that is stored somewhere in the computer's memory. We don't care where this value is stored, however, because
- Whenever we want to operate on this particular integer value in our program, we refer to it via its symbolic name `x`, for example:

  ```
  if (x > 17) x = x + 5;
  ```

- Furthermore, the "thing" that we've named `x` understands how to respond to a number of different operations, such as addition (+), subtraction (–), multiplication (*), division (/), logical comparisons (>, <, =), and so on, as defined by the `int` type.

In an OO language like Java, we can define a class such as `Student` and then declare a variable to be of type `Student`, as follows:

```
Student y;
```

What does this ***really*** mean? It means that

- `y` is a symbolic name that we've invented to refer to a `Student` object (i.e., an instance of the `Student` class) that is stored somewhere in the computer's memory. We don't care where this object is stored, however, because
- Whenever we want to operate on this particular object in our program, we refer to it via its symbolic name `y`, for example:

  ```
  if (y.isOnAcademicProbation()) {
       System.out.println ("Uh oh ...");
  }
  ```

- Furthermore, the "thing" that we've named `y` understands how to respond to a number of different service requests—how to register for a course, drop a course, and so on—that have been defined by the `Student` class.

Note the parallels between `y` as a `Student` and `x` as an `int` in the preceding examples. Just as `int` is said to be a ***predefined type*** in Java (and other languages), the `Student` class is said to be a **user-defined type**. And, because `y` in the preceding example is a variable that ***refers to*** an instance (object) of class `Student`, `y` is known as a **reference variable**.

In contrast, variables declared to be one of the eight primitive types in Java—`int`, `double`, `float`, `byte`, `short`, `long`, `char`, and `boolean`—are ***not*** reference variables, because in Java, primitive types are ***not*** **reference types**; that is, they do not refer to ***objects***:

```
// x is NOT a reference variable, because in Java,
// an int is NOT an object.
int x;
```

```
// yesNo is NOT a reference variable, because in Java,
// a boolean is NOT an object.
boolean yesNo;

// etc.
```

Various OO languages differ in their treatment of simple types. In some OO languages (e.g., Smalltalk), ***all*** types, including "simple" types such as `int` and `char`, are reference types, whereas in other languages (e.g., Java and C++), they are not.

The fact that Java contains a mix of reference types and nonreference types affects the way that we manipulate variables under certain circumstances, such as when we're placing primitive values into **collections**, a subject that we'll explore in Chapter 6.

Naming Conventions for Reference Variables

Names for reference variables follow the same convention as method and attribute names—that is, they use camel casing. Some sample reference variable declarations are as follows:

```
Course prerequisiteOfThisCourse;
Professor facultyAdvisor;
Student student;
```

The last of these, `Student student;`, might take some getting used to if you are new to case-sensitive programming languages. However, it is considered good programming practice to pattern reference variable names after the name of the class to which they belong if there is only one such variable within scope in a particular body of code. Because the class name starts with an uppercase "S" and the reference variable name starts with a lowercase "s," `Student` and `student` are ***completely different symbols*** as far as the compiler is concerned.

Instantiating Objects: A Closer Look

Different OO languages differ in terms of when an object is actually instantiated (created). In Java, when we declare a variable to be of a user-defined type, as in

```
Student y;
```

we haven't actually created an object in memory yet. Rather, we've simply declared a reference variable of type `Student` named `y`. This reference variable has the ***potential*** to refer to a `Student` object, but it doesn't refer to one just yet; rather, as with variables of the various simple types, `y`'s value is undefined as far as the compiler is concerned (it is null) until we explicitly assign it a value.

If we want to instantiate a ***brand-new*** `Student` object for `y` to refer to, we have to take the distinct step of using a special Java keyword, `new`, to allocate a new `Student` object within the JVM's memory at run time. We associate the new object with the reference variable `y` via an assignment statement, as follows:

```
y = new Student();
```

Don't worry about the parentheses at the end of the preceding statement. I'll talk about their significance in Chapter 4, when we discuss the notion of **constructors**.

Think of the newly created object as a helium balloon, as shown in Figure 3-7, and a reference variable as the hand that holds a string tied to the balloon so that we may access the object whenever we'd like.

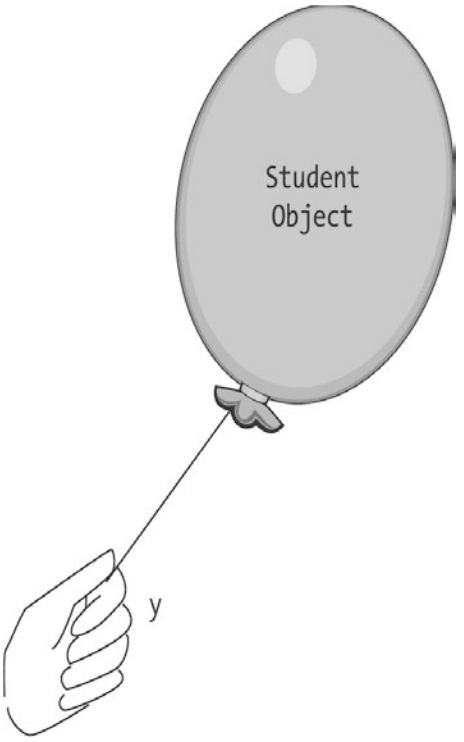


Figure 3-7. *Using a reference variable to keep track of an object in memory*

Because a reference variable is sometimes informally said to "hold on to" an object, we often use the informal term **handle**, as in the expression "Reference variable y maintains a handle on a `Student` object."

If you're familiar with the concept of ***pointers*** from languages such as C and C++, a reference is similar to a pointer in that it refers behind the scenes to the memory location/address where a particular object is stored. Java references ***differ*** from pointers, however, in that references can't be manipulated arithmetically the way that pointers can.

We could also create a new object without immediately assigning it to a reference variable, as in the following line of code:

```
new Student();
```

But such an object would be like a helium balloon without a string: it would indeed exist, but we'd never be able to access this object in our program. It would, in essence, "float away" from us in memory immediately after being "inflated."

Note that we can combine the two steps—declaring a reference variable and actually instantiating an object for that variable to refer to—in a single line of code:

```
Student y = new Student();
```

Another way to initialize reference variable `y` is to use an assignment statement to "hand" `y` a reference to an ***already existing*** object, that is, an object ("helium balloon") that is already being referenced by a ***different*** reference variable `x`. Let's look at an example:

```
// We declare a reference variable, and instantiate our first
   Student object.
Student x = new Student();

// We declare a second reference variable, but do *not* instantiate
// a second Student object.
Student y;

// We assign y a reference to the SAME object that x is referring to
// (x continues to refer to it, too).  We now, in essence,
// have two "strings" tied to the same "balloon."
y = x;
```

The conceptual outcome of the preceding assignment statement is illustrated in Figure 3-8: two "strings," being held by two different "hands," tied to the same "balloon"—that is, two ***different*** reference variables referring to the ***same*** physical object in memory.

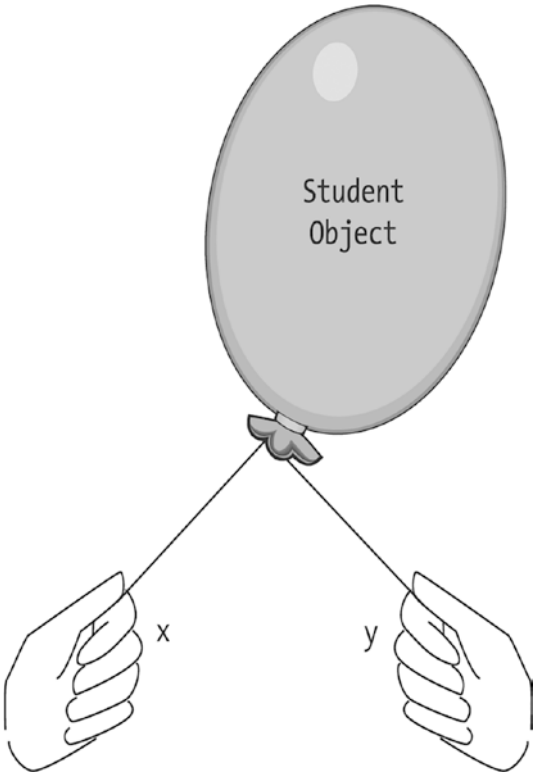


Figure 3-8. *Maintaining multiple handles on the same object*

We therefore see that ***multiple*** reference variables may simultaneously refer to the ***same*** object. However, any ***one*** reference variable may only hold on to/refer to ***one*** object at a time. Therefore, if a reference variable is already holding on to an object, it must let go of that object to reference a different object. If there comes a time when all of the handles for a particular object have been released, then that object is no longer accessible to our program, like a helium balloon that has been let loose.

Let's expand our previous example to illustrate these concepts (note the **highlighted** code that has been added):

```
// We declare a reference variable, and instantiate our first
   Student object.
Student x = new Student();

// We declare a second reference variable, but do not instantiate a
// second object.
Student y;
```

```
// We assign y a reference to the SAME object that x is referring to
// (x continues to refer to it, too).
y = x;

// We declare a THIRD reference variable and instantiate a SECOND
// Student object.
Student z = new Student();
```

At this point in time, we now have two references to the ***first*** `Student` object, `x` and `y`, and one reference, `z`, to the ***second*** `Student` object, as illustrated in Figure 3-9.

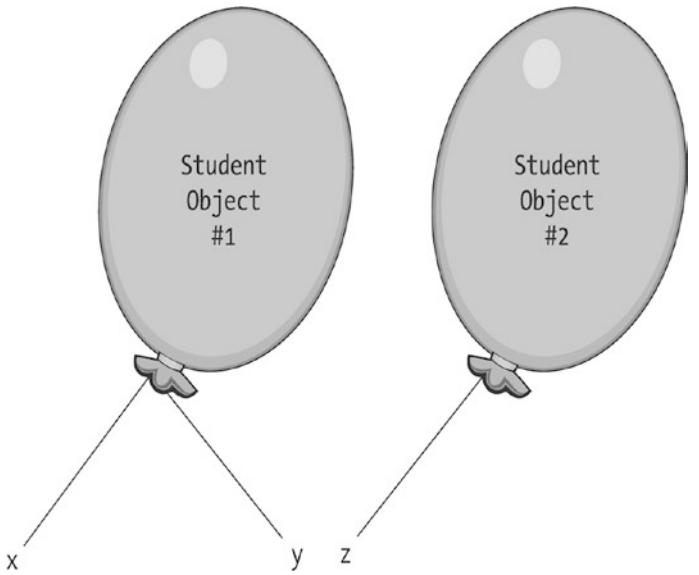


Figure 3-9. *A second object comes into existence*

Let's expand our example yet again (note the **highlighted** code that has been added):

```
// We declare a reference variable, and instantiate our first
   Student object.
Student x = new Student();

// We declare a second reference variable, but do *not* instantiate
// a second object.
Student y;
```

```
// We assign y a reference to the SAME object that x is referring to
// (x continues to refer to it, too).
y = x;

// We now declare a third reference variable and instantiate a second
// Student object.
Student z = new Student();

// We reassign y to refer to the same object that z is referring to;
// y therefore lets go of the first Student object and grabs on to
   the second.
y = z;
```

Because we've now asked y to refer to the same object that z is referring to—namely, the ***second*** `Student` object—y must release its handle on the ***first*** `Student` object. This is illustrated in Figure 3-10. (Note that x is still holding on to the first `Student` object, however.)

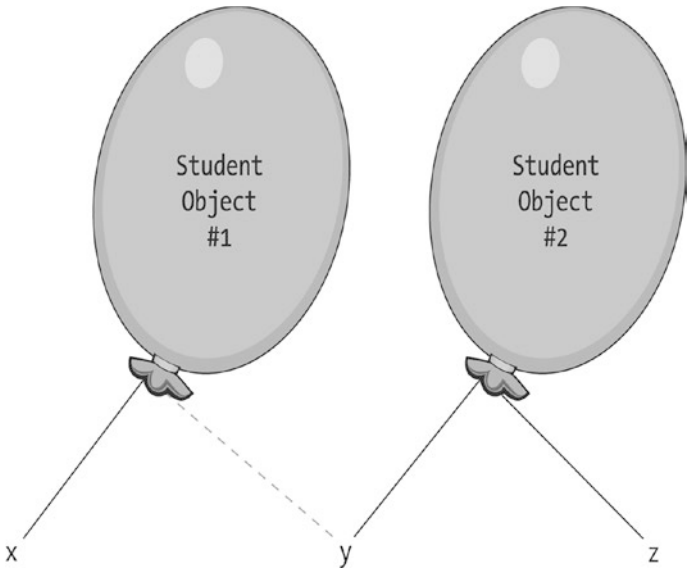


Figure 3-10. *Transferring object handles*

We'll now complete our example with the code that we've **highlighted**:

```
// We declare a reference variable, and instantiate our first
// Student object.
Student x = new Student();

// We declare a second reference variable, but do *not* instantiate
// a second object.
Student y;

// We assign y a reference to the SAME object that x is referring to
// (x continues to refer to it, too).
y = x;

// We now declare a third reference variable and instantiate a second
// Student object.
Student z = new Student();

// We reassign y to refer to the same object that z is referring to.
y = z;

// We reassign x to refer to the same object that z is referring to.
// x therefore lets go of the first Student object, and grabs on to
// the second, as well.
x = z;
```

Because we've now asked `x` to refer to the same object that `z` is referring to—namely, the ***second*** `Student` object—`x` must release its handle on the ***first*** `Student` object. Since we're no longer maintaining any references to the first `Student` object whatsoever—`x`, `y`, and `z` are now all referring to the ***second***—the first `Student` object is now lost to the program, as illustrated in Figure 3-11.

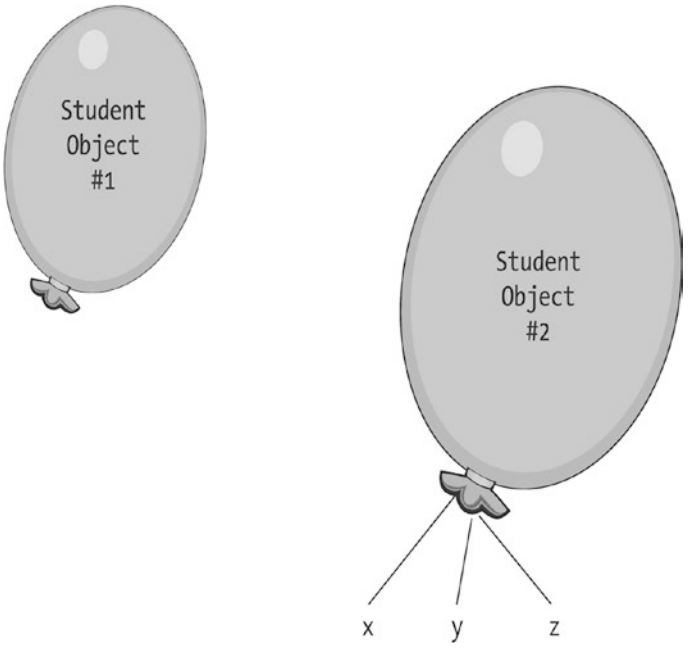


Figure 3-11. *The first* `Student` *object is now lost to our program*

Another way to get a reference variable to release its handle on an object is to set the reference variable to the value `null`, which as we discussed in Chapter 2 is the Java keyword used to represent a nonexistent object. Continuing with our previous example

- Setting `x` to `null` gets `x` to release its handle on the second `Student` object, as illustrated in Figure 3-12:

  ```
  x = null;
  ```

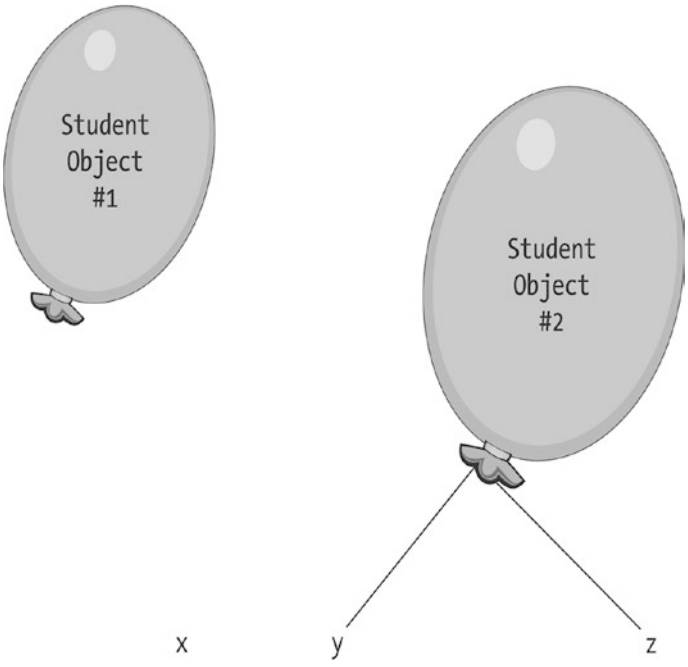


Figure 3-12. *Only two handles remain on the second* `Student` *object*

- Setting y to `null` gets y to release its handle on the second `Student` object, as illustrated in Figure 3-13:

```
x = null;
y = null;
```

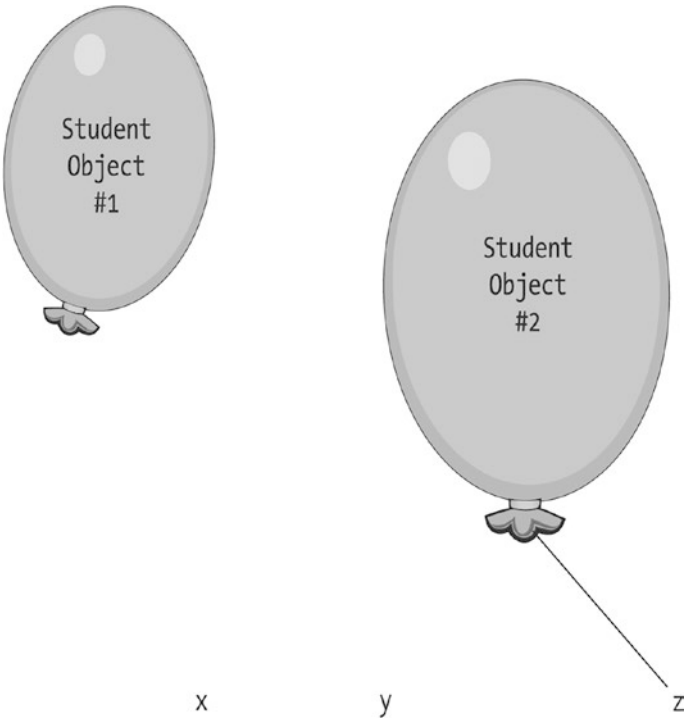


Figure 3-13. *Only one handle remains on the second* `Student` *object*

- Ditto for z, as illustrated in Figure 3-14:

```
x = null;
y = null;
z = null;
```

Now, ***both*** `Student` objects have been lost to our program!

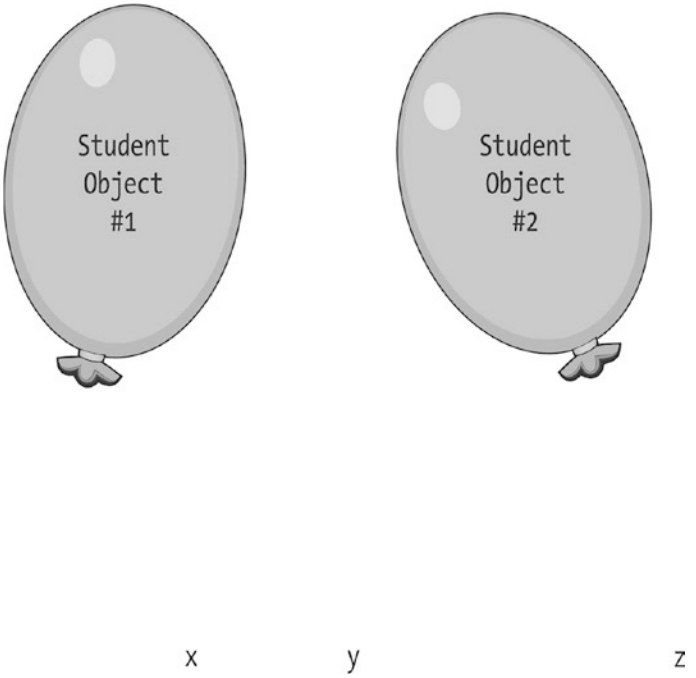


Figure 3-14. *The second* `Student` *object is now lost to our program, as well*

Garbage Collection

As it turns out, if all of an object's handles are released, it might seem as though the memory that the object occupies within the JVM would be permanently wasted. In a language like C++, this can indeed happen if programmers don't explicitly take care to reclaim the memory of an object that is no longer needed before all of its handles are dropped. Failure to do so is a chronic source of problems in C++ programs and is commonly known as a **memory leak**.

In C++, this is accomplished via the statement

```
x.delete();
```

but such a statement is unnecessary in Java, for reasons that we are about to discuss.

With Java, on the other hand, the JVM periodically performs **garbage collection**, a process that automatically reclaims the memory of "lost" objects for us while an application is executing. Here's how the Java garbage collector works:

- If there are no remaining active references to an object, it becomes a ***candidate*** for garbage collection.
- The garbage collector doesn't ***immediately*** recycle the object, however; rather, garbage collection occurs whenever the JVM determines that the application is getting low on free memory or when the JVM is otherwise idle.
- So, for some period of time, the "orphaned" object will still exist in memory—we simply won't have any handles/reference variables with which to access it.

The inclusion of garbage collection in Java has virtually eliminated memory leaks of the type that arose in C++. Note that it is still possible for the JVM itself to run out of memory, however, if we maintain handles on too many active objects simultaneously. Thus, a Java programmer cannot be totally oblivious to memory management; managing memory is just less error-prone in Java than it is in C/C++.

Objects As Attributes

When I first discussed the attributes and methods associated with the `Student` class, I stated that some of the attributes could be represented by predefined types provided by the Java language, whereas the types of a few others (`advisor`, `courseLoad`, and `transcript`) were temporarily left unspecified. Let's now put what you've learned about user-defined types to good use with respect to one of these attributes: the `Student` class's `advisor` attribute.

Rather than declaring the `advisor` attribute as simply a `String` representing the advisor's name, we'll declare it to be of user-defined type—namely, type `Professor`, another class that we've invented. This attribute type is reflected in Table 3-2.

Table 3-2. *Student Class Attributes Revisited*

Attribute	Type
name	String
studentID	String
birthDate	Date
address	String
major	String
gpa	double
advisor	**Professor**
courseLoad	???
transcript	???

This is reflected as well as in our Student class declaration:

```
public class Student {
    // Attribute declarations typically appear first ...
    String name;
    String studentId;
    Date birthDate;
    String address;
    String major;
    double gpa;
    // A class is a user-defined type, and so we may declare attributes
    // to be of such a type.
    Professor advisor;
    // type? courseLoad - we'll declare this attribute later.
    // type? transcript – ditto.

    // ... followed by method declarations (details omitted for now).
}
```

By having declared the `advisor` attribute to be of type `Professor`—that is, by having the `advisor` attribute serve as a ***reference variable***—we've just enabled a `Student` object to maintain a handle on the actual `Professor` object that represents the professor who is advising the student.

The `Professor` class, in turn, might be defined to have the attributes listed in Table 3-3.

Table 3-3. *Professor Class Attributes*

Attribute	Type
`Name`	`String`
`employeeID`	`String`
`birthdate`	`Date`
`Address`	`String`
`worksFor`	`String (or Department)`
`Advisee`	**`Student`**
`teachingAssignments`	`???`

Again, by having declared the `advisee` attribute to be of type `Student`—that is, by making the `advisee` attribute a reference variable—we've just given a `Professor` object a way to hold on to/refer to the actual `Student` object that represents the student whom the professor is advising.

The methods of the `Professor` class might be as follows:

- `transferToDepartment`
- `adviseStudent`
- `agreeToTeachCourse`
- `assignGrades`
- And so forth

Just as we did for the `Student` class earlier in the chapter, we'd render this `Professor` class design in Java code as follows:

```
public class Professor {
    // Attributes.
    String name;
    String employeeId;
    Date birthDate;
    String address;
    String worksFor;
    Student advisee;
    double gpa;
    // type? teachingAssignments – we'll declare this attribute later.

    // ... followed by method declarations (details omitted for now).
}
```

This class definition would reside in a source file named `Professor.java` and would be subsequently compiled into bytecode form as a file named `Professor.class`.

Note that, as mentioned earlier for the `Student` class, we'd need to insert the declaration

```
import java.util.Date;
```

ahead of the declaration

```
public class Professor { ... }
```

in order for the code to compile properly.

The following are a few noteworthy points about the `Professor` class:

- It's likely that a professor will be advising several students simultaneously, so having an attribute like `studentAdvisee` that can reference only a ***single*** `Student` object is not terribly useful. We'll discuss techniques for handling this in Chapter 6, when we discuss

collections, which we'll also see as being useful for defining the `teachingAssignments` attribute of `Professor` and the `courseLoad` and `transcript` attributes of `Student`.

- The `worksFor` attribute represents the department to which a professor is assigned. We can choose to represent this as either a simple `String` representing the department name, for example, "MATH", or as a reference variable that maintains a handle on a `Department` object—specifically, the `Department` object representing the "real-world" Math Department. Of course, to do so would require us to define the attributes and methods for a new class called `Department`.

As we'll discuss further in Part 2 of this book, the decision of whether or not we need to invent a new user-defined type/class to represent a particular real-world concept/abstraction isn't always clear-cut.

A Compilation "Trick": "Stubbing Out" Classes

If we were to want to program the `Student` and `Professor` classes in Java as shown earlier, neither one could be compiled in isolation; that is, if we merely wrote the code for the `Student` class

```
// Student.java

public class Student {
    // Attribute declarations typically appear first ...
    String name;
    String studentId;
    Date birthDate;
    String address;
    String major;
    double gpa;
    Professor advisor;
    // etc.
}
```

without having yet written the code for the `Professor` class (`Professor.java`), we'd get a compilation error on `Student` as follows

```
Student.java: cannot find symbol
symbol   : class Professor
location : class Student
Professor advisor;
^
```

because we haven't yet defined "`Professor`" as a type to the Java compiler.

We could wait until we've programmed both the `Student` and `Professor` classes before attempting to compile either one of them, but what if we were to introduce a third class into the mix

```
public class Student {
    String name;
    Professor advisor;
    Department major;
    // etc.
}

public class Professor {
    String name;
    Student advisee;
    Department worksFor;
    // etc.
}

public class Department {
    String name;
    Professor chairman;
    // etc.
}
```

or a fourth class or a fifth? Must we program ***all*** of them before compiling any ***one*** of them?

Fortunately, we can use the technique of **stubbing out** a class to temporarily work around issues related to compiling a class X that refers to a class Y, which we haven't yet programmed. Going back to our `Student` class as originally written

```
// Student.java

public class Student {
    // Attribute declarations typically appear first ...
    String name;
    // etc.
    Professor advisor;
    // etc.
}
```

we can temporarily code a "bare-bones" `Professor` class as follows:

```
// Professor.java

// A "stub" class:  note that the body consists of a pair of empty braces!
public class Professor { }
```

Trivial as this `Professor` class definition may be, it is nonetheless considered to be a ***legitimate*** class definition by the Java compiler that, when compiled, will yield a `Professor.class` bytecode file.

When we now attempt to compile our `Student.java` file, the compiler will indeed deem "`Professor`" to be a valid symbol—specifically, the name of a user-defined type—and `Student` will compile properly, as well.

Recall that we can compile the `Student.java` and `Professor.java` ("stub") files simultaneously with the single command

```
javac *.java
```

That is, `Professor.java` needn't be compiled separately first.

Composition

Whenever we create a class, such as `Student` or `Professor`, in which one or more of the attributes are themselves references to other objects, we are employing an OO technique known as **composition**. The number of levels to which objects can be conceptually bundled inside one another is endless, and so composition enables us to model very sophisticated real-world concepts. As it turns out, most "interesting" classes employ composition.

With composition, it may conceptually seem as though we're physically nesting objects one inside the other, as depicted in Figure 3-15.

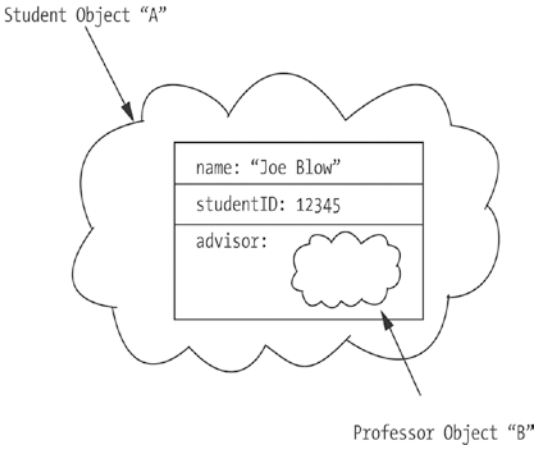


Figure 3-15. *Conceptual object "nesting"*

Actual object nesting (i.e., declaring one class inside of another) is possible in many OO programming languages, and it does indeed sometimes make sense—namely, if an object A doesn't need to have a life of its own from the standpoint of an OO application and it exists only for the purpose of serving enclosing object B**.**

- Think of your brain, for example, as an object that exists only within the context of your body (another object).
- As an example of object nesting relevant to the SRS, let's consider a grade book used to track student performance in a particular course. If we were to define a `GradeBook` class and then create `GradeBook` objects as attributes—one per `Course` object—then it might be reasonable for each `GradeBook` object to exist wholly within the context of its associated `Course` object. No other objects would need

> to communicate with the `GradeBook` directly; if a `Student` object wished to ask a `Course` object what grade the `Student` has earned, the `Course` object might internally consult its embedded `GradeBook` object and simply hand a letter grade back to the `Student`.

However, we often encounter the situation—as with the sample `Student` and `Professor` classes—in which an object A needs to refer to an object B, object B needs to refer back to object A, and ***both*** objects need to be able to respond to requests independently of each other as made by the application as a whole. In such a case, handles come to the rescue!

In reality, we are ***not*** storing whole objects as attributes inside of other objects; rather, we are storing ***references*** to objects. When an attribute of an object A is defined in terms of an object reference B, the two objects exist separately in memory and simply have a convenient way of finding one another whenever it's necessary for them to interact. Think of yourself as an object and your cellular phone number as your reference. Other people—"objects"—can reach you via your cell phone number to speak with you whenever they need to, even though they don't know where you're physically located; and conversely, if you have their cell phone numbers, you can call them whenever you like.

Memory allocation using handles might look something like Figure 3-16 conceptually.

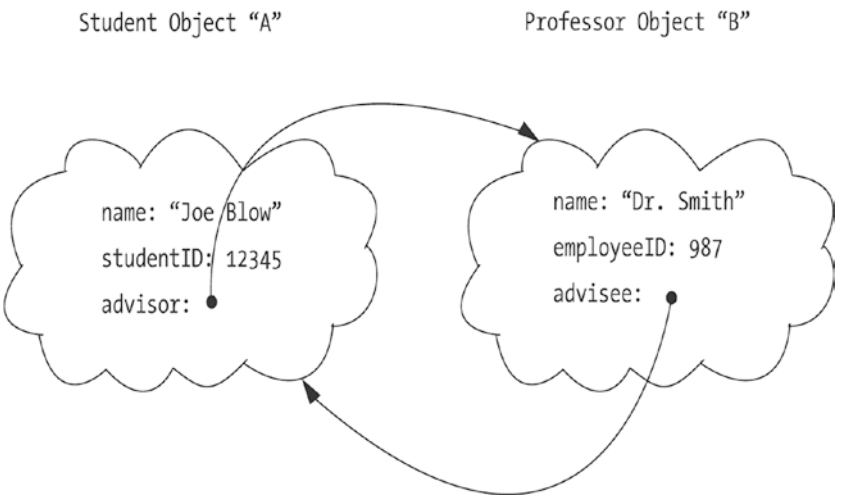


Figure 3-16. *Objects exist separately in memory and maintain handles on one another*

With this approach, each object is allocated in memory only once:

- The `Student` object knows how to find and communicate with its advisor (`Professor`) object whenever it needs to through its `advisor` attribute/handle/***reference***.
- The `Professor` object knows how to find and communicate with its advisee (`Student`) object whenever it needs to through its `advisee` attribute/handle/***reference***.

You'll learn how to actually code such object intercommunication in Java in Chapter 4.

The Advantages of References As Attributes

What do we gain by defining the `Student`'s `advisor` attribute as a reference to a `Professor` object, instead of merely storing the name of the advisor as a `String` attribute of the `Student` object? ***We avoid data redundancy and the associated potential loss of data integrity***. Let's see how this works.

By encapsulating the name of each professor inside of the corresponding `Professor` object, each name will be represented in only one place within an application: namely, ***within the object to which the name belongs***, which is ***precisely*** where it belongs! (You'll learn in Chapter 4 how to ask a `Professor` object for its name whenever you need to know it.) Then, if a given professor's name changes for some reason, we have only one copy of that name to change in our application—the name that is encapsulated inside of the corresponding `Professor` object.

If we were to instead design our application such that we redundantly stored the `Professor`'s name both as a `String` attribute of the `Professor` object and as a `String` attribute of every `Student` object that the professor advises, we'd have a lot more work to do! We'd have to remember to update the professor's name not only in the `Professor` object but also in potentially many different `Student` objects. If we were to forget to update all such objects, then the name of the `Professor` might wind up being inconsistent from one `Student` instance to another.

Just as important, by maintaining a handle on the `Professor` object via the `advisor` attribute of `Student`, the `Student` object can also ***request other services*** of this `Professor` object via whatever methods are defined for the `Professor` class. In addition

to asking for the advisor's (`Professor`'s) name, a `Student` object may, for example, ask its advisor (`Professor`) object where the `Professor`'s office is located or what courses the `Professor` is teaching so that the `Student` can sign up for one of them.

Three Distinguishing Features of an Object-Oriented Programming Language

In order to be considered truly object-oriented, a programming language must provide support for three key mechanisms:

- User-defined (reference) types
- Inheritance
- Polymorphism

You've just learned about the first of these mechanisms; we'll discuss the other two in chapters to follow.

Summary

In this chapter, you've learned that

- An ***object*** is a software abstraction of a physical or conceptual real-world object.
- A ***class*** serves as a template for creating objects. Specifically, a class defines (a) the data that an object will encapsulate, known as the object's ***attributes***, and (b) the behaviors that an object will be able to perform, known as an object's operations/***methods***.
- An object may be thought of as an ***instance*** of a class to which attribute ***values*** have been assigned—in essence, a ***filled-in*** template.
- Just as we can declare variables to be of primitive types such as `int`, `double`, and `boolean`, we can also declare variables to be of ***user-defined types*** such as `Student` and `Professor`. User-defined types are declared as classes.

- When we create a new object at run time (a process known as ***instantiation***), we typically store a reference to ("handle" on) that object in a ***reference variable***. We can then use the reference variable as a symbolic name for accessing and communicating with the object.
- We can define attributes of a class A to serve as references to objects belonging to another class B. In doing so, we allow each object to encapsulate the information that rightfully belongs to that object, but enable objects to find one another in memory at run time so that they can contact one another to share information whenever necessary.

EXERCISES

1. From the perspective of an academic setting (but not necessarily the SRS case study specifically), think about what the appropriate attributes and methods of the following classes might be:
   - `Classroom`
   - `Department`
   - `Degree`
2. [*Coding*] Render the `Student` and `Professor` classes as presented in this chapter in Java code. In so doing, (a) omit method declarations, and (b) declare any `Date` attributes to be of type `String` instead. Finally, create a third class called `MainClass` to serve as a "wrapper" for a `main` method that instantiates one `Student` object and one `Professor` object.
3. [*Coding*] Revise the code that you wrote for Exercise 2 to include a fourth class, `Department`; declare whatever `Department` attributes seem reasonable but, once again, omit method declarations for now. Then, go back and modify the `major` attribute of `Student` to refer to a `Department` and the `worksFor` attribute of `Professor` to refer to a `Department`. Finally, modify the `MainClass`'s `main` method to instantiate a `Department` object along with a `Student` and a `Professor`.

4. For the problem area whose requirements you defined for Exercise 3 in Chapter 1, list the classes that you might need to create in order to model it properly.
5. List the classes that you might need to create in order to model the Prescription Tracking System (PTS) discussed in the Appendix.
6. Would `Color` be a good candidate for a user-defined type/class? Why or why not?

CHAPTER 4

Object Interactions

As you learned in Chapter 3, objects are the building blocks of an object-oriented software system. In such a system, objects collaborate with one another to accomplish common system goals, similar to the ants in an anthill or the employees of a corporation or the cells in your body. Each object has a specific structure and mission; these respective missions complement one another in accomplishing the overall mission of the system as a whole.

In this chapter, you'll learn

- How methods are used to specify an object's behaviors
- The various code elements that make up a method
- How objects publicize their methods as services to one another
- How objects communicate with one another to request one another's services in order to collaborate
- How objects maintain their data and how they guard their data to ensure its integrity
- The power of an OO language feature known as **information hiding** and how information hiding can be used to limit ripple effects on an application's code when the private implementation details of a class inevitably change
- How a special type of function known as a **constructor** can be used to initialize the state of an object when it is first instantiated

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_4

Events Drive Object Collaboration

At its simplest, the process of object-oriented software development involves the following four basic steps:

1. Properly establishing the functional requirements for, and overall mission of, an application
2. Designing the appropriate classes—their data structures, behaviors, and relationships with one another—necessary to fulfill these requirements and mission
3. Instantiating these classes to create the appropriate types and number of object instances
4. Setting these objects in motion through **external triggering events**

Think of an anthill: At first glance, you may see no apparent activity taking place. But if you drop a candy bar nearby, a flurry of activity suddenly begins as ants rush around to gather up the "goodies," as well as to repair any damage that may have been caused if you dropped the candy bar ***too close*** to the anthill!

Within an OO application (the "anthill"), the objects ("ants") may be set in motion by an external event such as

- The click of a button on the SRS graphical user interface, indicating a student's desire to register for a particular course
- The receipt of information from some other automated system, such as when the SRS receives a list of all students who have paid their tuition from the university's Billing System

As soon as such a triggering event has been noted by an OO system, the appropriate objects react, performing services themselves and/or requesting services of other objects in chain reaction fashion, until some overall goal of the application has been accomplished. For example, the request to register for a course as made by a student user via the SRS application's GUI may involve the collaboration of many different objects, as illustrated in Figure 4-1:

- A `Student` object (an abstraction of the ***real*** student user)
- A `DegreeProgram` object, to ensure that the requested course is truly required for the student to graduate

- The appropriate `Course` object, to make sure that there is a seat available for the student in that course
- A `Classroom` object, representing the room in which the course will be meeting, to verify its seating capacity
- A `Transcript` object—specifically, the `Transcript` of the `Student` of interest—to ensure that the student has met all prerequisites for the course

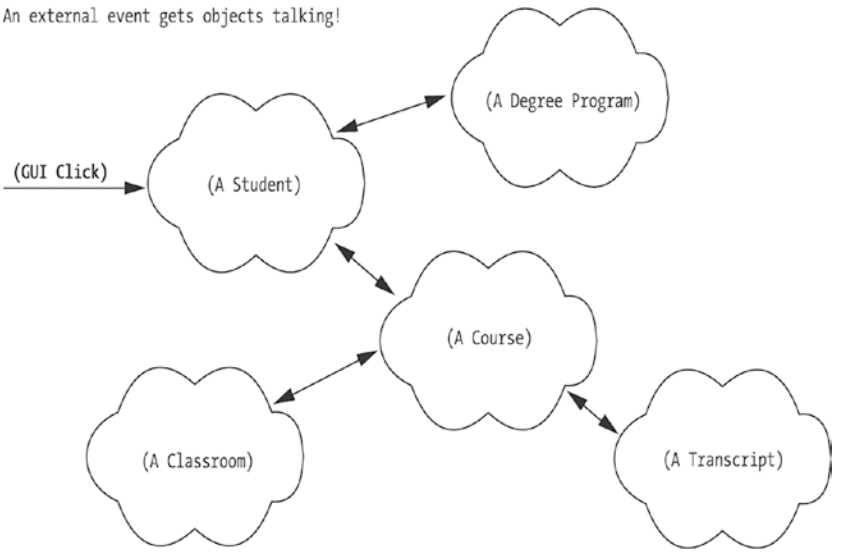


Figure 4-1. *SRS objects must collaborate to accomplish the overall SRS mission*

Meanwhile, a student user of the SRS is blissfully ignorant of all the objects that are "scurrying around" behind the scenes to accomplish their goal. The student merely fills in a few fields and clicks a button on the SRS GUI and a few moments later sees a message that either confirms or rejects their registration request.

Once the ultimate goal of an event chain has been achieved (e.g., registering a student for a course), an application's objects effectively become idle and may remain so until the next such triggering event occurs. An object-oriented application is in some ways similar to a game of billiards: hit the cue ball with your cue, and it (hopefully!) hits another ball, which might collide with three other balls, and so on. Eventually, however, all balls will come to a standstill until the cue ball is hit again.

Declaring Methods

Let's talk in a bit more detail about how we formally specify an object's behaviors as Java methods. Recall from Chapter 3 that an object's behaviors may be thought of as services that the object can perform. In order for an object A to request some service of an object B, A needs to know the specific language with which to communicate with B. That is,

- ***Object A needs to be clear as to exactly which of B's methods/services A wants B to perform***. Think of yourself as object A and a pet dog as object B. Do you want your dog to sit? Stay? Heel? Fetch?
- ***Depending on the service request, object A may need to give B some additional information so that B knows exactly how to proceed***. If you tell your dog to fetch, the dog needs to know ***what*** to fetch: a ball? A stick? The neighbor's cat?
- ***Object B in turn needs to know whether object A expects B to report back the outcome of what it has been asked to do***. In the case of a command to fetch something, your dog will hopefully bring the requested item to you as an outcome. However, if your dog is in another room and you call out the command "Sit!" you won't see the result of your command; you have to trust that the dog has done what you have asked it to do.

We take care of specifying/defining these three aspects of each method by declaring a **method header**. We must then program the behind-the-scenes logic for ***how*** B will perform the requested service in the **method body**.

For readers familiar with the C programming language, a Java method declaration is virtually the same syntactically as a C function declaration. The only philosophical difference between a function in a non-OO language like C and a method in an OO language like Java is the context in which they are executed: a non-OO function is executed by the programming environment as a whole, whereas a method in an OO language is executed by a particular object. We'll explore this difference in more detail as this chapter unfolds.

Let's look at method headers first.

Method Headers

A **method header** is a formal specification (from a programming standpoint) of how that method is to be invoked. A method header, at minimum, consists of

- A method's **return type**—that is, the type of information that is going to be returned by object B to object A, if any, when B's method finishes executing.
- A method's name.
- An optional list of comma-separated **formal parameters** (specifying their types and names) to be passed to the method, enclosed in parentheses. If no parameters need be passed in, an empty set of parentheses is used; such methods are said to "take no parameters," and we'll refer to them as **parameterless**.

As an example, here is a typical method header that we might define for the `Student` class:

```
  boolean  registerForCourse(String courseID, int secNo)
return type  method name    comma-separated list of formal parameters,
                              enclosed in parentheses
                            (parentheses may be left empty)
```

When casually referring to a method such as `registerForCourse` in narrative text, some authors attach an empty set of parentheses, `()`, to the method name, for example, `registerForCourse()`. This doesn't necessarily imply that the ***formal*** header is parameterless, however.

Method Naming Conventions

Java method names are crafted using the ***camel casing*** style; recall from Chapter 2 that variable names are also crafted using camel casing. By way of review, with camel casing

- The first letter of the method name is in lowercase.
- The first letter of each subsequent concatenated word in the method name is in uppercase, and the remaining characters are in lowercase.
- We don't use any "punctuation" characters—dashes, underscores, etc.—to separate these words.

As an example, `chooseAdvisor` is an appropriate method name, whereas none of the following would be appropriate: `ChooseAdvisor` (uppercase "C"), `chooseadvisor` (lowercase "a"), `choose_advisor` (separating underscore).

Passing Arguments to Methods

The purpose of passing arguments into a method is twofold:

- To provide it with the (optional) "fuel" necessary to do its job
- To otherwise guide its behavior in some fashion

With the `registerForCourse` method shown previously, for example, it's necessary to tell the specific `Student` object performing the method which course we want it to register for; we'll do so by passing in two arguments, a course ID (e.g., "MATH 101") and a section number (e.g., `10`, which happens to meet Monday nights from 8:00 to 10:00 p.m.), as illustrated here:

```
boolean registerForCourse(String courseID, int secNo)
```

Had we instead declared the `registerForCourse` method header with an ***empty*** parameter list

```
boolean registerForCourse()
```

the request would be ambiguous, because the `Student` object performing this method would have no idea as to which course/section it's expected to register for.

Not all methods require such "fuel," however; some methods are able to produce results solely based on the information stored internally within an object as attribute values, in which case no additional guidance is needed in the form of arguments. For example, the method

```
int getAge()
```

is designed to be parameterless because a `Student` object can presumably tell us its age without having to be given any qualifying information, perhaps by comparing the value of its `birthDate` attribute with the system date. Let's say, however, that we wanted a `Student` object to be able to report its age expressed either in years or in months; in such a case, we might wish to declare the `getAge` method as follows

```
int getAge(int ageType)
```

allowing us to pass in an `int`(eger) argument to serve as a control flag for informing the `Student` object of how we want the answer to be returned. That is, we might program the `getAge` method so that

- If we pass in a value of 1, it means that we want the answer to be returned in terms of years.
- If we pass in a value of 2, we want the answer to be returned in terms of months (e.g., a 21-year-old student would respond that it is 252 months old).

An alternative way of handling the requirement to retrieve the age of a `Student` object in two different forms would be to define two separate methods, such as perhaps the following:

```
int getAgeInYears()
int getAgeInMonths()
```

But in object-oriented programming, it's common practice to control a method's behavior through the values (and types) of arguments.

Method Return Types

The `registerForCourse` method as previously declared is shown to have a return type of `boolean`, which implies that this method will return one of the following two values:

- A value of `true`, to signal "mission accomplished"—namely, that the `Student` object has successfully registered for the course that it was instructed to register for.
- A value of `false`, to signal that the registration request has been denied for some reason. Perhaps the desired section was full, or the student didn't meet the prerequisites of the course, or the requested course/section has been canceled, etc.

> In Chapter 13, you'll learn techniques for communicating and determining precisely ***why*** the mission of a method has failed when we discuss **exception handling**.

Note that a method may be designed so as to not return anything—that is, it may go about its business silently, without needing to report the outcome of its efforts. If so, it is declared to have a return type of `void` (another of Java's keywords). As an example, consider the `Student` method header

```
void setName(String newName)
```

This method requires one argument—a `String` representing the new name that we want this `Student` object to assume—and performs "silently" by setting the `Student` object's internal `name` attribute to whatever value is being passed into the method, returning no answer in response.

Here's an additional example of a method header that we might declare for the `Student` class with a `void` return type:

```
void switchMajor(String newDepartment, Professor newAdvisor)
```

This method represents a request for a `Student` object to change its major field of study, which involves designating both a new academic department (e.g., "BIOLOGY") and a reference to the `Professor` object that is to serve as the student's advisor in this new department.

The preceding example demonstrates that we can declare parameters to be of any type, including user-defined types such as `Professor`. The same is true for the return type of a method; for example, a method with the following header

```
Professor getAdvisor()
```

could be used to ask a `Student` object who its advisor is. Rather than merely returning the ***name*** of the advisor, the `Student` object returns a reference to the `Professor` object as a whole (as recorded by the `Student`'s internal `facultyAdvisor` attribute; you'll learn how to inform a `Student` object of ***which*** `Professor` object is to serve as its `facultyAdvisor` a bit later in the chapter).

Note that a method can return at most one result, which may seem limiting. What if, for example, we want to ask a `Student` object for a list of ***all*** of the `Courses` that the `Student` has ever taken—must we ask for these one by one through multiple method

calls? Fortunately not. The result handed back by a method can actually be a reference to an object of arbitrary complexity, including a special type of object called a **collection** that can contain references to ***multiple*** other objects. We'll talk about collections in depth in Chapter 6.

An Analogy

Let's use an analogy to help illustrate what we've discussed so far about methods. With respect to household chores, let's say that a person is capable of

- Taking out the trash
- Mowing the lawn
- Washing the car

Expressing this notion in Java code, we'd perhaps declare three methods for the `Person` class, one for each chore (service):

- `takeOutTheTrash`
- `mowTheLawn`
- `washTheCar`

In the case of the `takeOutTheTrash` method, we needn't provide any qualifying details in the form of arguments, nor do we expect the person performing this service (method) to report back to us, so we declare the method header with a return type of `void` and an empty parameter list:

```
void takeOutTheTrash()
```

In the case of the `mowTheLawn` method, we'd like whoever is mowing the lawn to report back to us as to whether or not they see any crabgrass, but again, we needn't provide any qualifying details in the form of arguments, so we declare the method header with a return type of `boolean` (where `true` means the person saw crabgrass and `false` means the person did not) and an empty parameter list:

```
boolean mowTheLawn()
```

Finally, in the case of the `washTheCar` method, we might own several different cars, and so we'd need to specify which car is to be washed by passing in a reference to the car of interest. We needn't get any sort of response from the person doing the washing, however, and so we might craft the following method header:

```
void washTheCar(Car c)
```

We'll revisit this "chores" analogy to expand upon it later in this chapter.

Method Bodies

When we design and program a class's methods, declaring method headers alone is not enough: we must also program the internal details of how each method should behave when invoked. These internal programming details, known as the **method body**, are enclosed within braces { ... } immediately following the method header, as follows:

```
public class Student {
    // Attributes.
    String name;
    double gpa;
    // Other Student attribute declarations have been omitted from this
    example ...

    // We declare a method header ...
    boolean isHonorsStudent() {
        // ... and program the details of what this method is to do
        // within enclosing braces ... this is the method body.

        // Here, we're accessing the value of "gpa", declared as an
        // attribute of the Student class above.
        if (gpa >= 3.5) {
            // Returning the value "true" indicates "yes, this is
            // an honors student".
            return true;
        }
        else {
            // Returning the value "false" indicates "no, this isn't
            // an honors student".
```

```
            return false;
        }
    }

    // Other method declarations for the Student class would follow, e.g.,
    // getName(), setName(), getGpa(), setGpa() ... details omitted.
}
```

We can thus see that ***a method is a function***—a function that is ***performed by a specific object***, but a function nonetheless.

Features May Be Declared in Any Order

Note that the relative order in which features are declared within a Java class doesn't matter. That is, we're permitted to reference a feature A from within method B even though the declaration of feature A comes ***after*** the declaration of method B in the overall class declaration.

For example, in the following simple class, we declare two methods, `foo` and `bar`, and one attribute, `x`. The `foo` method is able to invoke the `bar` method, despite the fact that the declaration of `bar` comes ***after*** the declaration of `foo` in the class:

```
public class Simple {
    // Attributes.
    int x;

    // Methods.
    void foo() {
        // Invoke bar() from within foo.
        bar();
    }

    // bar() is declared AFTER foo().
    void bar() {
        System.out.println(x);
    }
}
```

> All languages are not created equal in this regard; in C++, for example, you may only reference a feature if it has been ***previously declared***. Hence, invoking `bar` from `foo` would generate a compilation error if the preceding example were a C++ vs. Java example.

Similarly, attribute declarations needn't precede method declarations for a class; it is thus permissible (but not common) to rewrite our `Student` class as follows:

```
public class Student {
    // Here, we BEGIN with method declarations ...

    void foo() {
        bar();
    }

    void bar() {
        // We are able to reference attribute 'x' despite the fact
        // that its declaration hasn't
        // been 'seen' by the compiler yet.
        System.out.println(x);
    }

    // ... and END with attribute declarations.
    int x;
}
```

However, it is common practice to consolidate all attribute declarations at the beginning of a class, prior to declaring any of its methods.

return Statements

A `return` statement is a jump statement that is used to exit a method:

```
void doSomething() {
  // Pseudocode.
```

```
  do whatever is required by this method ...

    return;
}
```

Whenever a `return` statement is encountered, the method stops executing as of that line of code, and execution control immediately returns to the code that invoked the method in the first place.

For methods with a return type of `void`, the `return` keyword is used by itself, as a complete statement:

```
return;
```

However, it turns out that for methods with a return type of `void`, the use of a `return;` statement is ***optional.*** If omitted, a `return;` statement is implied as the last line of the method. That is, the following two versions of method `doSomething` are equivalent:

```
void doSomething() {
    int x = 3;
    int y = 4;
    int z = x + y;
}
```

and

```
void doSomething() {
    int x = 3;
    int y = 4;
    int z = x + y;
    return;
  }
```

The bodies of methods with a ***non***-`void` return type, on the other hand, ***must*** include at least one explicit `return` statement. The `return` keyword in such a case must be followed by an expression that evaluates to a value compatible with the method's declared return type. For example, if a method is defined to have a return type of `int`, then any of the following `return` statements would be acceptable:

```
return 0;          // returning a constant integer value

return x;          // returning the value of x (assuming that x
                   // has previously been declared to be an int)

return x + y;      // returning the value of the expression "x + y" (here,
                   // we're assuming that "x + y" evaluates to an int value)

return (int) z;    // casting the value of z (assume z was declared as
                      a double)
                   // to an int value
```

and so forth. As another example, if a method is defined to have a return type of `boolean`, then any of the following `return` statements would be acceptable:

```
return false;          // returning a boolean constant value

return outcome;        // returning the value of variable outcome
                       // (assuming that outcome has previously been
                       // declared to be of type boolean)

return (x < 3);        // returning the boolean value that results when
                       // the (numeric) value of x is compared to 3:
                       // if x is less than 3, this method returns a
                       // value of true; otherwise, it returns false.
```

A method body is permitted to include more than one `return` statement. Good programming practice, however, is to have only ***one*** `return` statement in a method, at the very end. Let's look once again at the `isHonorsStudent` method discussed previously, which has two `return` statements:

```
boolean isHonorsStudent() {
  if (gpa >= 3.5) {
    return true;   // first return statement
  }
  else {
    return false;  // second return statement
  }
}
```

Let's rewrite this method to use a locally declared `boolean` variable, `result`, to capture the `true`/`false` answer that is to ultimately be returned. We'll return the value of `result` with a single `return` statement at the very end of the method:

```
boolean isHonorsStudent() {
    // Declare a local variable to keep track of the outcome; arbitrarily
    // initialize it to false.
    boolean result = false;

    if (gpa >= 3.5) {
        // Instead of returning true, we record the value in our "result"
        // variable:
        result = true;
    }
    else {
        // Instead of returning false, we record the value in our "result"
        // variable:
        result = false;
    }

    // We now have a single return statement at the end of our method to
    return the
    // result.
    return result;
}
```

As it turns out, since we initially assigned the value `false` to `result`, setting it to `false` explicitly in the `else` clause is unnecessary; we could therefore simplify the `isHonorsStudent` method as follows:

```
boolean isHonorsStudent() {
    // Declare a local variable to keep track of the outcome; arbitrarily
    // initialize it to false.
    boolean result = false;

    if (gpa >= 3.5) {
      result = true;
    }
```

```
    // Note that we've removed the 'else' clause ... if the "if" test
    // fails, variable "result" already has a value of false.

    return result;
}
```

There is, however, one situation in which multiple `return` statements are considered acceptable, and that is when a method needs to perform a series of operations where failure at any step along the way constitutes failure as a whole. This situation is illustrated via pseudocode:

```
// Pseudocode.
boolean someMethod() {
    // Perform a test … if it fails, we wish to abort the method as
    // a whole.
    if (first test fails) {
        return false;
    }

    // If we pass the first test, we do some additional processing ...
    do something interesting ...

    // Then, perhaps we perform a second test, where again failure of the
    // test warrants immediately giving up in our 'quest'.
    if (second test fails) {
        return false;
    }

    // If we pass the second test, we do some additional processing ...
    // details omitted.

    // If we reach this point in our code, we return a value of true
    // to signal that we made it to the finish line!
    return true;
}
```

Note that the Java compiler will verify that ***all*** logical pathways through a method return an appropriately typed result. For example, the following method will generate a compiler error because a proper `return` statement will only be reached if the `if` test succeeds; if the `if` test fails, the `return` statement is bypassed:

```
boolean xGreaterThanThree(int x) {
    if (x <= 3) {
        return false;
    }
}
```

The specific compiler error message in this case would be as follows:

```
missing return statement:
      boolean xGreaterThanThree(int x) {
```

Methods Implement Business Rules

The logic contained within a method body defines the **business logic**, also known as **business rules**, for an abstraction. In the `isHonorsStudent` method, for example

```
boolean isHonorsStudent() {
    boolean result = false;

    if (gpa >= 3.5) {
      result = true;
    }

    return result;
}
```

a single business rule is expressed for determining whether or not a student is an honors student, namely,

If a student has a grade point average (GPA) of 3.5 or higher, then they are an honors student.

If the business rules underlying this method were more complex—say, if the rules were as follows

In order for a student to be considered an honors student, the student must

(a) *Have a grade point average (GPA) of 3.5 or higher*

(b) *Have taken at least three courses*

(c) *Have received no grade lower than "B" in any of these courses*

then our method's logic would of necessity be more complex:

```
boolean isHonorsStudent() {
    boolean result = false;

    // Pseudocode.
    if ((gpa >= 3.5) &&
        (number of courses taken >= 3) &&
        (no grades lower than a B have been received)) {
            result = true;
    }

    return result;
}
```

In a sense, even a method ***header*** expresses a simple form of business rule/requirement–in this particular case, that there is such a notion as an "honors student" in the first place. But the ***details*** of an application's business rule(s) are encoded in its various classes' method ***bodies***.

Objects As the Context for Method Invocation

As mentioned in passing a bit earlier in the chapter, methods in an OO programming language (OOPL) differ from functions in a non-OOPL in that

- Functions are executed by the programming environment as a whole.
- Methods are executed by specific objects.

That is, we are able to invoke a C function "in a vacuum" as follows

```
// A C program.
void main() {
    sqrt(42.0);  // invoke the sqrt (square root) function ...
    // etc.
}
```

whereas in an OOPL like Java, we typically must **qualify** the method call by prefixing it with the name of the reference variable representing the object that is to perform the method, followed by a period (**dot**). This is illustrated for the `registerForCourse` method as follows:

```
// Instantiate two Student objects.
Student x = new Student();
Student y = new Student();

// Invoke the registerForCourse method on Student object x, asking it to
// register for course MATH 101, section 10; Student y is unaffected.
x.registerForCourse("MATH 101", 10);
```

We refer to an expression of the form `referenceVariable.methodName(args)` as a **message**–that is, this line of code

```
x.registerForCourse("MATH 101", 10);
```

can be interpreted as either "invoking a method on object `x`" or "sending a message to object `x`." Either way, such code should be viewed as ***requesting object*** `x` ***to perform a method as a service, on behalf of the application to which the object belongs***.

The terminology "sending a message to an object" originated with the Smalltalk language and is used when speaking of OOPLs generically. When talking about Java specifically, the terminology "invoking a method on an object" is preferred. Similarly, the Java-specific alternative for the generic OOPL term "message" is "method invocation." Throughout the book, I'll alternate between the generic and Java-specific forms for referring to these notions, but tend to favor the generic "message" nomenclature.

Because we use a "dot" to append a method call to a particular reference variable, we informally refer to the notation `referenceVariable.methodName(args)` as **dot notation**.

Another informal way to think of the notation `x.methodName(args)` is that we are "**talking to" object** `x`; specifically, we are "talking to" object `x` to request it to perform a particular method/service. Let's return to the analogy of household chores introduced earlier in the chapter to illustrate this point.

Recall that a person is capable of the following household chores:

- Taking out the trash
- Mowing the lawn
- Washing the car

Here's an expression of this abstraction as Java code:

```
public class Person {
    // Attributes omitted from this snippet ...

    // Methods.
    void takeOutTheTrash() {  ... }
    boolean mowTheLawn() { ... }
    void washTheCar(Car c)  { ... }
}
```

We decide that we want our teenaged sons Larry, Moe, and Curly to each do one of these three chores. How would we ask them to do this? If we were to simply say

- "Please wash the Camry."
- "Please take out the trash."
- "Please mow the lawn, and let me know if you see any crabgrass."

chances are that ***none*** of the chores would get done, because we haven't tasked a ***specific*** son with fulfilling any of these requests! Larry, Moe, and Curly will probably all stay glued to the TV, because none of them will acknowledge that a request has been directed toward them specifically.

On the other hand, if we were to instead say

- "***Larry***, please wash the Camry."
- "***Moe***, please take out the trash."
- "***Curly***, please mow the lawn, and let me know if you see any crabgrass."

we'd be directing each request to a ***specific*** son; again, using Java syntax, this might be expressed as follows:

```
// We declare and instantiate three Person objects:
Person larry = new Person();
Person moe = new Person();
Person curly = new Person();

// And, while we're at it, a Car object, as well!
Car camry = new Car();

// We send a message to each son, indicating the service that we wish
// each of them to perform:
larry.washTheCar(camry);
moe.takeOutTheTrash();
boolean crabgrassFound = curly.mowTheLawn();
```

By applying each method call to a specific "son" (`Person` object reference), there is no ambiguity as to which object is being asked to perform which service.

Assuming that `takeOutTheTrash` is a method defined for the `Person` class as previously illustrated, the following code won't compile in Java (or, for that matter, in any OOPL) because the method call is **unqualified**—that is, the dot notation is missing:

```
public class BadCode {
  public static void main(String[] args) {
    // This next line won't compile -- where's the "dot"?  That is,
    which object
    // are we talking to???
    takeOutTheTrash();
  }
}
```

The following compilation error would be reported:

```
cannot find symbol
symbol  :    method takeOutTheTrash()
location:  class BadCode
```

However, in a ***non***-OOPL like C, there is no notion of objects or classes, and so functions in such languages are ***always*** invoked "in a vacuum" (i.e., in the programming environment as a whole):

```
// A C program.
void main() {
    sqrt(42.0);
    // etc.
}
```

Java Expressions, Revisited

When we discussed Java expressions in Chapter 2, there was one form of expression that was omitted from the list—namely, ***messages***—because we hadn't yet talked about objects. I've repeated the list of what constitutes Java expressions here, adding message expressions to the mix:

- ***A constant***: `7`, `false`
- ***A char(acter) literal***: `'A'`, `'&'`
- ***A String literal***: `"foo"`
- ***The name of any variable declared to be of one of the predefined types that we've seen so far***: `myString`, `x`
- ***Any one of the preceding that is modified by one of the Java unary operators***: `i++`
- ***A method invocation ("message")***: `z.length()`
- ***Any two of the preceding that are combined with one of the Java binary operators***: `z.length() + 2`
- ***Any of the preceding simple expressions enclosed in parentheses***: `(z.length() + 2)`

The ***type*** of a message expression is the type of the result that the method returns. For example, if `length()` is a method with a return type of `int`, then the expression `z.length()` is an expression of type `int`, and, if `registerForCourse` is a method with a return type of `boolean`, then the expression `s.registerForCourse(...)` is an expression of type `boolean`.

Capturing the Value Returned by a Method

Whenever we invoke a method with a non-`void` return type, it's up to us to choose to either ignore or react to the value that the method returns. In an earlier example, we declared the `Student` class's `registerForCourse` method to have a return type of `boolean`:

```
boolean registerForCourse(String courseID, int secNo)
```

But we didn't pay any attention to what `boolean` value was returned when we invoked the method:

```
x.registerForCourse("MATH 101", 10);
```

If we wish to react to the value returned by a non-`void` method, we may choose to capture the value in a variable declared to be of the appropriate type, as in the following example:

```
boolean successfullyRegistered = x.registerForCourse("MATH 101", 10);

if (!successfullyRegistered) { // or:  if (successfullyRegistered == false)
    // Pseudocode.
    action to be taken if registration failed ...
}
```

However, if we only plan on using the returned value from a method once in our code, then going to the trouble of declaring an explicit variable such as `successfullyRegistered` to capture the result is overkill. We can instead react to the result simply by ***nesting*** a message expression within a more complex statement. For example, we can rewrite the preceding code snippet to eliminate the variable `successfullyRegistered` as follows:

```
// Register for a course and react to the value that is returned by
the method.
if (!(x.registerForCourse("MATH 101", 10))) {
    // Pseudocode.
    action to be taken if registration failed ...
}
```

Because the `registerForCourse` method returns a `boolean` value, the message `x.registerForCourse(...)` is a `boolean` expression and can be used within the `if` clause of an `if` statement. Furthermore, we can apply the `!` ("not") operator to the expression, as in the preceding example.

We often combine method calls with other types of statements when developing object-oriented applications—for example, when printing to the command window:

```
Student s = new Student();
// Details omitted.
System.out.println("The student named " + s.getName() +
    " has a GPA of " + s.getGPA());
```

Method Signatures

We've already learned that a method header consists at a minimum of the method's return type, name, and formal parameter list:

```
void switchMajor(String newDepartment, Professor newAdvisor)
```

From the standpoint of the code used to invoke a method on an object, however, the return type and parameter names aren't immediately evident upon inspection:

```
Student s = new Student();
Professor p = new Professor();
// Details omitted ...
s.chooseMajor("MATH", p);
```

We can infer from inspecting the last line of code that

- `chooseMajor` is the name of a method defined for the `Student` class; otherwise, the compiler would reject this line.
- The `chooseMajor` method declares two parameters of types `String` and `Professor`, respectively, because those are the types of the arguments that we're passing in: specifically, a `String` literal and a reference to a `Professor` object.

However, what we ***cannot*** determine from inspecting this code is (a) how the formal parameters were ***named*** in the corresponding method header or (b) what the ***return type*** of this method is declared to be; it may be `void`, or the method may be returning a non-`void` result that we've simply chosen to ignore.

For this reason, we refer to a method's **signature** as those aspects of a method header that are "discoverable" by inspecting the code used to invoke the method, namely

- The method's ***name***
- The order, types, and number of ***parameters declared by*** the method

but ***excluding***

- The parameter names
- The method's return type

Furthermore, we'll introduce the informal terminology **argument signature** to refer to that ***subset*** of a method signature consisting of the order, types, and number of arguments, but excluding the method ***name***.

"Argument signature" isn't an industry-standard term, but one that is nonetheless useful. We'll use it throughout the book.

Some examples of method headers and their corresponding method/argument signatures are as follows:

- ***Method header***: int getAge(int ageType)
  - ***Method signature***: `getAge(int)`
  - ***Argument signature***: `(int)`

- ***Method header***: `void chooseMajor(String newDepartment, Professor newAdvisor)`
  - ***Method signature***: `chooseMajor(String, Professor)`
  - ***Argument signature***: `(String, Professor)`
- ***Method header***: `String getName()`
  - ***Method signature***: `getName()`
  - ***Argument signature***: `()`

Choosing Descriptive Method Names

Assigning intuitive, descriptive names to our methods helps make an application's code self-documenting. When combined with carefully crafted variable names such as those chosen in the following code example, comments are (virtually) unnecessary:

```
public class IntuitiveNames {
    public static void main(String[] args) {
        Student student;
        Professor professor;
        Course course1;
        Course course2;
        Course course3;

        // Later in the program ...

        // This code is fairly straightforward to understand!
        // A student chooses a professor as its advisor ...
        student.chooseAdvisor(professor);
        // ... and registers for the first of three courses.
        student.registerForCourse(course1);

        // etc.
```

Now, contrast the preceding code with the much "fuzzier" code that follows:

```
public class FuzzyNames {
    public static void main(String[] args) {
        Student s;
```

```
Professor p;
Course c1;
Course c2;
Course c3;

// Later in the program ...

// Without comments, this next bit of code is not nearly as
intuitive.
s.choose(p);
s.reg(c1);

// etc.
```

Method Overloading

Overloading is a language mechanism that allows two or more different methods belonging to the same class to have the ***same*** name as long as they have ***different*** argument signatures. Overloading is supported by numerous non-OO languages like C as well as by OO languages like Java.

For example, the `Student` class may legitimately define the following five different `print` method headers:

```
void print(String fileName) { ... // version #1
void print(int detailLevel) { ... // version #2
void print(int detailLevel, String fileName) { ... // version #3
int print(String reportTitle, int maxPages) { ... // version #4
boolean print() { ... // version #5
```

Hence, the `print` method is said to be **overloaded**. Note that all five of the methods differ in terms of their argument signatures:

- The first takes a single `String` as an argument.
- The second takes a single `int`.
- The third takes two arguments—an `int` followed by a `String`.

- The fourth takes two arguments—a `String` followed by an `int` (although these are the same parameter types as in the previous header, they are in a different order).
- The fifth takes no arguments at all.

Thus, all five of these headers represent valid, different methods, and all can coexist happily within the `Student` class without any complaints from the compiler.

We can then choose which of these five "flavors" of the `print` method we'd like a `Student` object to perform based on what form of message we send to a `Student` object:

```
Student s = new Student();

// Invoking the version of print that takes a single String argument.
s.print("output.rpt");

// Invoking the version of print that takes a single int argument.
s.print(2);

// Invoking the version that takes two arguments, an int followed by
a String.
s.print(2, "output.rpt");

// etc.
```

The compiler is able to unambiguously match up which version of the `print` method is being called in each instance based on the argument signatures.

This example illustrates why overloaded methods must have unique argument signatures: if we ***were*** permitted to introduce the following additional `print` method as a sixth method of `Student`

```
boolean print(int levelOfDetail) { ... // version #6
```

despite the fact that its argument signature—a single `int`—duplicates the argument signature of one of the other five `print` methods

```
void print(int detailLevel) { ... // version #2
```

then the compiler would be unable to determine which version of the `print` method, #2 or #6, we are trying to invoke with the following line of code:

```
  s.print(3);  // Which version to we want to execute:  #2 or #6?  HELP!!!
```

So, to make life simple, the compiler prevents this type of ambiguity from arising in the first place by preventing classes from declaring like-named methods with identical argument signatures. The compiler error we'd generate if we were to try to declare version #6 of the `print` method along with the other five versions would be as follows:

```
print(int) is already defined in Student
boolean print(int levelOfDetail) {
        ^
```

The ability to overload methods allows us to create an entire family of similarly named methods that do essentially the same job. Think back to Chapter 2 where we discussed the `System.out.println` method, which is used to display printed output to the command window. As it turns out, there is not one but ***many*** versions of the `System.out.println` method; each overloaded version accepts a different argument type (`println(int)`, `println(String)`, `println(double)`, etc.). Using an overloaded `System.out.println` method is much simpler and neater than having to use separate methods named `printlnString`, `printlnInt`, `printlnDouble`, and so on.

Note that there is no such thing as ***attribute*** overloading; that is, if a class tries to declare two attributes with the same name

```
public class Student {
  private String studentId;
  private int studentId;
  // etc.
```

the compiler will generate an error message on the second declaration:

```
studentId is already defined in Student
```

Message Passing Between Objects

Let's now look at a message passing example involving two objects. Assume that we have two classes defined—`Student` and `Course`—and that the following methods are defined for each.

- For the `Student` class:

  ```
  boolean successfullyCompleted(Course c)
  ```

  Given a reference `c` to a particular `Course` object, we're asking the `Student` object receiving this message to confirm that the student has indeed taken the course in question and received a passing grade.

- For the `Course` class:

  ```
  boolean register(Student s)
  ```

  Given a reference `s` to a particular `Student` object, we're asking the `Course` object receiving this message to do whatever is necessary to register the student. In this case, we expect a `Course` to ultimately respond `true` or `false` to indicate success or failure of the registration request.

Figure 4-2 reflects one possible message interchange between a `Course` object `c` and a `Student` object `s`; each numbered step in the diagram is narrated in the text that follows. Solid-line arrows represent messages being passed/methods being invoked; dashed-line arrows represent values being returned from methods.

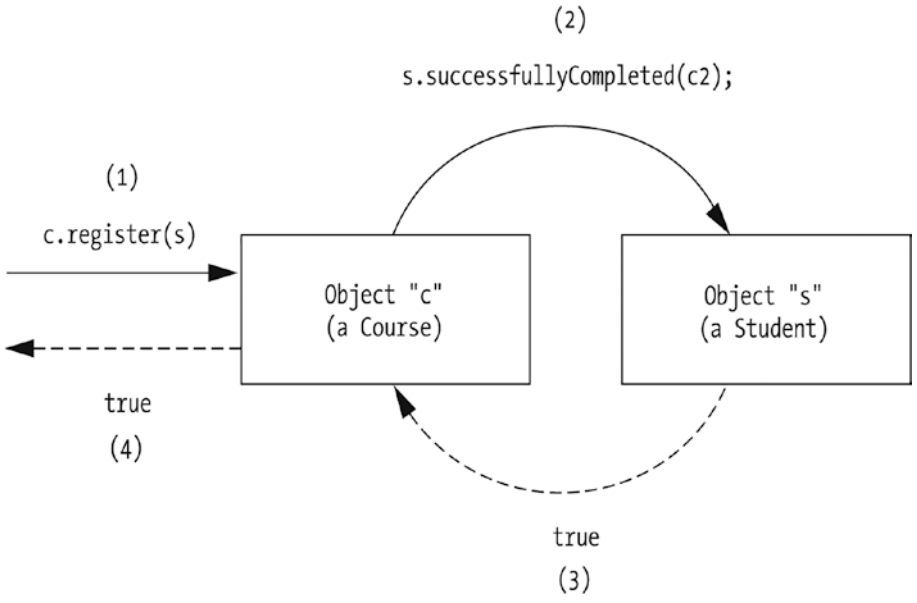


Figure 4-2. *Message passing between* `Student` *and* `Course` *objects*

(Please refer back to Figure 4-2 when reading through steps 1–4.)

1. A `Course` object `c` receives the message

   ```
   c.register(s);
   ```

   where `s` represents a particular `Student` object. (For now, we won't worry about the origin of this message; it was most likely triggered by a user's interaction with the SRS GUI. We'll see the complete code context of how all of these messages are issued later in this chapter, in the section entitled "Objects As Clients and Suppliers.")

2. In order for `Course` object `c` to officially determine whether or not `s` should be permitted to register, `c` sends the message

   ```
   s.successfullyCompleted(c2);
   ```

   to `Student s`, where `c2` represents a reference to a ***different*** `Course` object that happens to be a prerequisite of `Course c`. (Don't worry about how `Course c` knows that `c2` is one of its prerequisites; this involves interacting with `c`'s internal `prerequisites` attribute, which we haven't talked about yet. Also, `Course c2` isn't depicted in Figure 4-2 because, strictly speaking, `c2` isn't engaged in this "discussion" between objects `c` and `s`. `c2` is being talked ***about***, but isn't doing any talking itself!)

3. `Student` object `s` replies to `c` with the value `true`, indicating that `s` has successfully completed the prerequisite course. (We will for the time being ignore the details as to how `s` determines this; it involves interacting with `s`'s internal `transcript` attribute, which we haven't covered the structure of just yet.)

4. Convinced that the student has complied with the prerequisite requirements for the course, `Course` object `c` finishes the job of registering the student (internal details omitted for now) and confirms the registration by responding with a value of `true` to the originator of the service request.

This example was overly simplistic; in reality, `Course c` may have had to speak to numerous other objects as well:

- A `Classroom` object (the room in which the course is to be held, to make sure that it has sufficient room for another student)
- A `DegreeProgram` object (the degree sought by the student, to make sure that the requested course is indeed required for the degree that the student is pursuing)
- And so forth—before sending a `true` response to indicate that the request to register `Student s` has been fulfilled

We'll see a slightly more complex version of this message exchange later in the chapter.

Delegation

If a request is made of an object A and, in fulfilling the request, A in turn requests assistance from another object B, this is known as **delegation** by A to B. The concept of delegation among objects is exactly the same as delegation between people in the real world: if your "significant other" asks you to mow the lawn while they are out running errands and you in turn hire a neighborhood teenager to mow the lawn, then, as far as your partner is concerned, the lawn has been mowed. The fact that you delegated the activity to someone else is (hopefully!) irrelevant.

The fact that delegation has occurred between objects is often transparent to the initiator of a message, as well. In our previous message passing example, `Course c` delegated part of the work of registering `Student s` ***back to*** `s` when `c` asked `s` to verify having taken a prerequisite course. However, from the perspective of the originator of the registration request—`c.register(s);`—this seems like a simple interaction: namely, the requestor asked c to register a student, and it did so! All of the behind-the-scenes details of what c had to do to accomplish this are hidden from the requestor (see Figure 4-3).

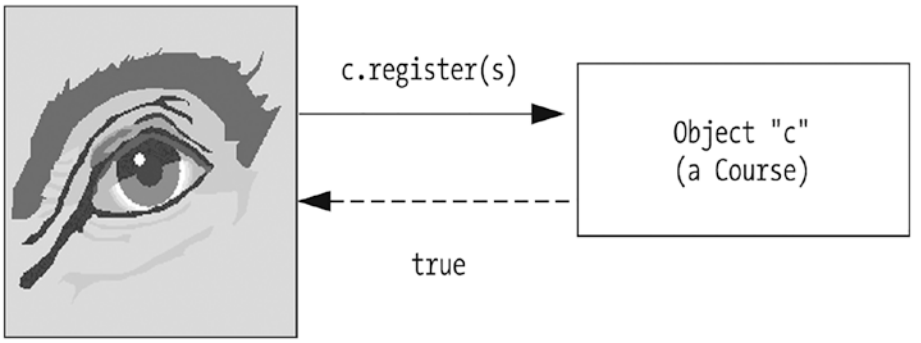


Figure 4-3. *A requestor sees only the external details of a message exchange*

Obtaining Handles on Objects

The only way that an object A can pass a message to an object B is if A has access to a reference to/handle on B. This can happen in several different ways.

- ***Object A might maintain a reference to B as one of A's attributes***. For example, here's the example from Chapter 3 of a `Student` object having a `Professor` reference as an attribute:

  ```
  public class Student {
      // Attributes.
      String name;
      Professor facultyAdvisor;
      // etc.
  ```

  (Again, you'll learn how to inform a `Student` object of ***which*** `Professor` object is to serve as its `facultyAdvisor` a bit later in the chapter.)

By way of analogy, this is like a person A "permanently" recording the phone number for person B in their address book so that A can look up and call B whenever A needs to interact with B.

- ***Object A may be handed a reference to B as an argument of one of A's methods***. This is how `Course` object `c` obtained access to `Student` object `s` in the preceding message passing example, when `c`'s `register` method was called:

```
c.register(s);
```

This is analogous to person A being handed a slip of paper with person B's phone number on it, so that A may call B.

- ***A reference to object B may be made "globally available" to the entire application***, such that all other objects can access it. We'll discuss techniques for doing so later in the book and employ such techniques in building the SRS.

This is analogous to advertising person B's phone number on a billboard for ***anyone*** to call!

- ***Object A may have to explicitly request a handle on/reference to B by calling a method on some third object C***. Since this is potentially the most complex way for A to obtain a handle on B, we'll illustrate this with an example.

This is analogous to person A having to call person C to ask C for person B's phone number.

Going back to the example interaction between `Course` object `c` and `Student` object `s` from a few pages ago, let's complicate the interaction a bit:

- First, we'll introduce a third object: a `Transcript` object `t`, which represents a record of all courses taken by `Student` object `s`.

- Furthermore, we'll assume that `Student s` maintains a handle on `Transcript t` as one of `s`'s attributes (specifically, the `transcript` attribute) and, conversely, that `Transcript t` maintains a handle on its "owner," `Student s`, as one of `t`'s attributes:

```
public class Student {
    // Attributes.
    Transcript transcript;
    // etc.
}

public class Transcript {
    // Attributes.
    Student owner;
    // etc.
}
```

Figure 4-4 reflects this more elaborate message interchange between `Course c`, `Student s`, and `Transcript t`; each numbered step in the diagram is narrated in the text that follows. Again, solid-line arrows represent messages being passed/methods being invoked; dashed-line arrows represent values being returned from methods.

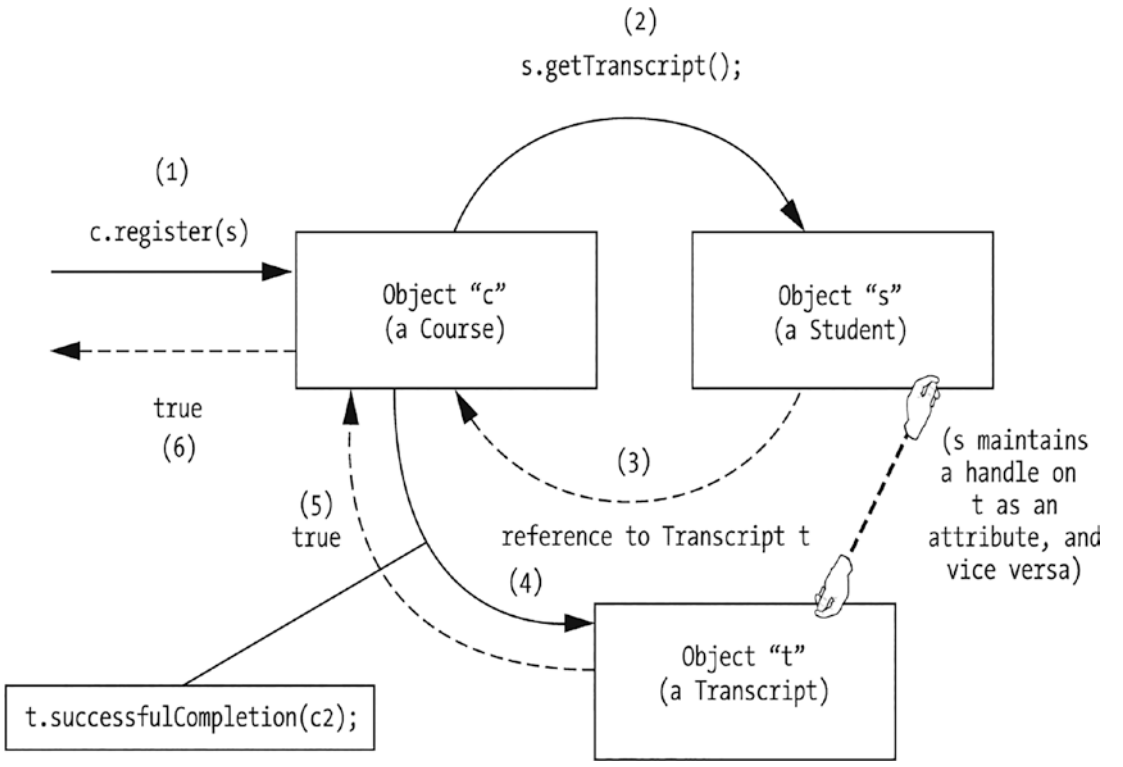


Figure 4-4. *A more complex message passing example involving three objects*

(Please refer back to Figure 4-4 when reading through steps 1–6.)

1. In this enhanced object interaction, the first step is exactly as previously described: namely, a `Course` object `c` receives the message

   ```
   c.register(s);
   ```

   where `s` represents a `Student` object.

2. Now, instead of `Course c` sending the message `s.successfullyCompleted(c2)` to `Student s` as before, where `c2` represents a prerequisite `Course`, `Course` object `c` instead sends the message

   ```
   s.getTranscript();
   ```

to the `Student`, because `c` wants to check `s`'s transcript firsthand. This message corresponds to a method on the `Student` class whose header is declared as follows:

```
Transcript getTranscript()
```

Note that this method is defined to return a `Transcript` object reference—specifically, a handle on the `Transcript` object belonging to this student.

3. Because `Student s` maintains a handle on its `Transcript` object as an attribute, it's a snap for `s` to respond to this message by passing a handle on `t` back to `Course` object `c`.

4. Now that `Course c` has its ***own*** temporary handle on `Transcript t`, object `c` can talk directly to `t`. Object `c` proceeds to ask `t` whether `t` has any record of `c`'s prerequisite course `c2` having successfully been completed by `Student s` by passing the message

   ```
   t.successfulCompletion(c2);
   ```

   This implies that there is a method defined for the `Transcript` class with the header

   ```
   boolean successfulCompletion(Course c)
   ```

5. `Transcript` object `t` responds with the value `true` to `Course c`, indicating that `Student s` has indeed successfully completed the prerequisite course in question. (Note that `Student s` is unaware that `c` is talking to `t`; `s` knows that it was asked by `c` to return a handle on `t` in an earlier message, but `s` has no insights as to ***why*** `c` asked for the handle.)

This is not unlike the real-world situation in which person A asks person C for person B's phone number, without telling C ***why*** they want to call B.

---

6. Satisfied that `Student s` has complied with its prerequisite requirements, `Course` object `c` finishes the job of registering the student (internal details omitted for now) and confirms the registration by responding with a value of `true` to the originator of the registration request that first arose in step 1. Now that `c` has finished with this transaction, it discards its (temporary) handle on `t`.

Note that, from the perspective of whoever sent the original message

```
c.register(s);
```

to `Course c`, this more complicated interaction appears ***identical*** to the earlier, simpler interaction, as shown in Figure 4-5. All the sender of the original message knows is that `Course c` eventually responded with a value of `true` to the request.

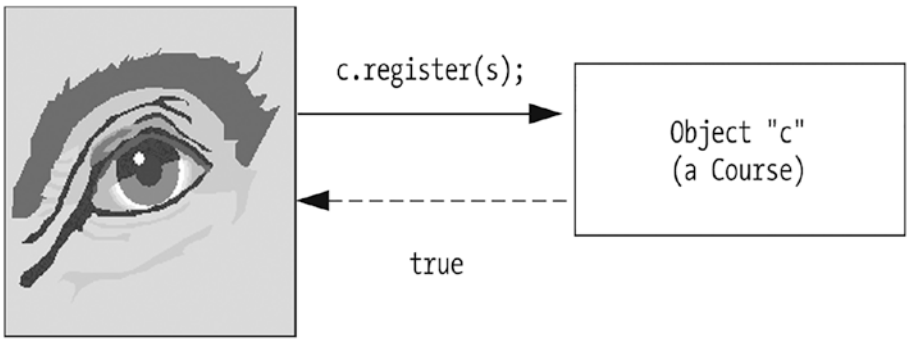


Figure 4-5. *The external details of this more complex interaction appear identical from the requestor's standpoint*

Objects As Clients and Suppliers

In the preceding example of message passing between a `Course` object and a `Student` object, we can consider `Course` object `c` to be a **client** of `Student` object `s`, because `c` is requesting that `s` perform one of its methods—namely, `getTranscript`—as a ***service*** to `c`. This is identical to the real-world concept of ***you***, as a client, requesting the services of an accountant or an attorney or an architect. Similarly, `c` is a client of `Transcript t` when `c` asks `t` to perform its `successfulCompletion` method. We therefore refer to code that invokes a method on an object X as **client code** relative to X because such code benefits from the services performed by X.

Let's look at a few examples of client code corresponding to the message passing example involving a `Course`, `Student`, and `Transcript` object from a few pages back. This first code example, taken from the `main` method of an application, instantiates two objects—`Course c` and `Student s`—and invokes a method on one of them, which gets them "talking":

```
public class MyApp {
    public static void main(String[] args) {
        Course c = new Course();
        Student s = new Student();

        // details omitted ...

        // Invoke a method on Course object c.
        // (This is labeled as message (1) in the earlier figure; the
        returned
        // value, labeled as (6) in that figure, is being captured
        in boolean
        // variable "success".)
        boolean success = c.register(s);

        // etc.
    }
}
```

In this example, the `main` method body is considered to be ***client code*** relative to `Course` object `c` because the `main` method calls upon `c` to perform its `register` method as a service.

Let's now look at the code that implements the body of the `register` method, inside of the `Course` class:

```
    public class Course {
      // Attribute details omitted ...

      public boolean register(Student s) {
        boolean outcome = false;

        // Request a handle on Student s's Transcript object.
        // (This is labeled as message (2) in the earlier figure.)
```

```
    Transcript t = s.getTranscript();
    // (The return value from this method is labeled as (3) in
    // the earlier figure.)

    // Now, request a service on that Transcript object.
    // (Assume that c2 is a handle on some prerequisite Course ...)
    // (This is labeled as message (4) in the earlier figure.)
    if (t.successfulCompletion(c2)) {
      // (This next return value is labeled as (5) in the earlier
      figure.)
      outcome = true;
    }
    else {
      outcome = false;
    }

    return outcome;
  }

  // etc.
```

We see that the `register` method body of the `Course` class is considered to be ***client code*** relative to ***both*** `Student` object `s` and `Transcript` object `t` because this code calls upon both `s` and `t` to each perform a service: `s.getTranscript()` and `t.successfulCompletion(c2)`.

Whenever an object A is a client of object B, object B in turn can be thought of as a **supplier** to A. Note that the roles of client and supplier are not absolute between two objects; such roles are only relevant for the duration of a particular message passing event. If I ask you to pass me the bread, I am your client, and you are my supplier; and if a moment later you ask me to pass you the butter, then you are my client, and I am your supplier.

The notion of objects as clients and suppliers is discussed further in *Object-Oriented Software Construction* by Bertrand Meyer (Prentice Hall, 2000).

Information Hiding/Accessibility

Just as we've been using dot notation to formulate messages to objects, we can also use dot notation to refer to an object's attributes. For example, if we declare a reference variable `x` to be of type `Student`, we can refer to any of `Student x`'s attributes from client code via the following notation

`x.`*`attribute_name`*

where the dot is used to qualify the name of the ***attribute*** of interest with the name of the reference variable representing the object to which it belongs: `x.name`, `x.gpa`, and so forth.

Here are a few additional examples:

```
// Instantiate two objects.
Student x = new Student();
Student y = new Student();

// Use dot notation to access attributes as variables.

// Assign student x's name ...
x.name = "John Smith";

// ... and student y's name.
y.name = "Joe Blow";

// Compare the ages of the two students.
if (x.age == y.age) { ... }
```

However, just because we ***can*** access attributes this way doesn't mean that we ***should***. There are many reasons we'll want to ***restrict*** access to an object's data so as to give the object complete control over when and how its data is altered and several mechanisms for how we can get the Java compiler's help in enforcing such restrictions.

In practice, objects often restrict access to some of their features (attributes or methods). Such restriction is known as **information hiding**. In a well-designed object-oriented application, a class typically publicizes ***what*** its objects can do—that is, the services the objects are capable of providing, as declared via the class's method headers—but ***hides*** the internal details both of ***how*** they perform these services and of the data (attributes) that they maintain internally in order to ***support*** these services.

By way of analogy, think of a Yellow Pages advertisement for a dry cleaner. Such an ad will promote the services that the dry cleaner provides—that is, ***what*** they can do for you: "We clean formal wear," "We specialize in cleaning area rugs," and so forth. However, the ad typically ***won't*** disclose the details of ***how*** they do the cleaning—for example, what specific chemicals or equipment that they use—because you, the potential customer, needn't know such details in order to determine whether a particular dry cleaner can provide the services that you need.

We use the term **accessibility** to refer to whether or not a particular feature of an object can be accessed outside of the class in which it is declared—that is, whether it is accessible from client code via dot notation. The accessibility of a feature is established by placing an **access modifier** keyword at the beginning of its declaration:

```
public class MyClass {
    // Attributes.
    access-modifier int x;
    // etc.

    // Methods.
    access-modifier void foo() { ... }
    // etc.
}
```

Java defines several different access modifiers. Let's explore the implications of using the two primary access modifiers: `private` and `public`.

There is a third access modifier, `protected`, as well as default `package`-level access that we'll defer discussing until later in the book.

Public Accessibility

When a feature is declared to have **public accessibility**, it's freely accessible from client code using dot notation. For example, if we were to declare the `name` attribute of the `Student` class as being publicly accessible by placing the keyword `public` just ahead of the attribute's type in the declaration

```
public class Student {
    public String name;
    // etc.
```

we've granted client code permission to directly access the `name` attribute of a `Student` object via dot notation; that is, it would be perfectly acceptable to write client code as follows:

```
public class MyProgram {
    public static void main(String[] args) {
        Student x = new Student();

        // Because name is a public attribute of the Student class, we
        may access
        // it via dot notation from client code.
        x.name = "Fred Schnurd";  // assign a value to x's name attribute
        // or:
        System.out.println(x.name);  // retrieve the value of x's name
                                        attribute

        // etc.
    }
}
```

Similarly, if we were to declare the `isHonorsStudent` method of `Student` as having public accessibility, which we do by adding the keyword `public` to the beginning of the method header declaration

```
public class Student {
    // Attribute details omitted from this example.

    // Methods.
    public boolean isHonorsStudent() { ... }

    // etc.
}
```

we've granted client code permission to invoke the `isHonorsStudent` method on a `Student` object via dot notation; that is, it would be perfectly acceptable to write client code as follows:

```
public class MyProgram {
    public static void main(String[] args) {
        Student x = new Student();

        // Because isHonorsStudent is a public method, we may access it
        // via dot notation from client code.
        if (x.isHonorsStudent()) { ... }

        // etc.
```

Private Accessibility

When a feature is declared to have **private accessibility**, on the other hand, it's ***not*** accessible outside of the class in which it's declared—that is, we may ***not*** use dot notation to access such a feature from client code. For example, if we were to declare the `ssn` attribute of the `Student` class to have `private` accessibility

```
public class Student {
  public String name;
  private String ssn;
  // etc.
```

then we would not be permitted to access `ssn` directly via dot notation from client code. In the following code example, a compiler error would arise on the line that is **bolded**:

```
public class MyProgram {
  public static void main(String[] args) {
    Student x = new Student();

    // Not permitted from client code!  ssn is private to the
    // Student class, and so this will not compile.
    x.ssn = "123-45-6789";

    // etc.
```

The resultant error message would be

```
ssn has private access in Student
```

The same is true for ***methods*** that are declared to be `private`—that is, such methods can't be invoked from client code. For example, if we were to declare the `printInfo` method of `Student` as being `private`

```
public class Student {
    // Attribute details omitted from this example.

    // Methods.
    public boolean isHonorsStudent() { ... }
    private void printInfo() { ... }

    // etc.
}
```

then it would not be possible to invoke the `printInfo` method on a `Student` object from within client code. In the following code snippet, a compiler error would arise on the line that is **bolded**:

```
public class MyProgram {
    public static void main(String[] args) {
        Student x = new Student();

        // Because printInfo() is a private method, we may not access it
        // via dot notation from client code; this won't compile:
        x.printInfo();

        // etc.
```

The resultant error message would be

```
printInfo() has private access in Student
```

Publicizing Services

As it turns out, methods of a class are typically declared to be `public` because an object (class) needs to publicize its services (as in the Yellow Pages advertisement analogy) so that client code may request these services. By contrast, most attributes are typically declared to be `private` (and effectively "hidden"), so that an object can maintain

ultimate control over its data. We'll look at several detailed examples later in this chapter of how an object goes about doing so.

Although it isn't explicitly declared as such, the internal code that implements each method (i.e., the method body) is also, in a sense, ***implicitly*** private. When a client object A asks another object B to perform one of its methods, A doesn't need to know the behind-the-scenes details of ***how*** B is doing what it's doing; object A needs simply to trust that object B will perform the "advertised" service. This is depicted conceptually in Figure 4-6, where those aspects of a class/object deemed to be private are depicted as being sealed off from client code by an impenetrable brick wall.

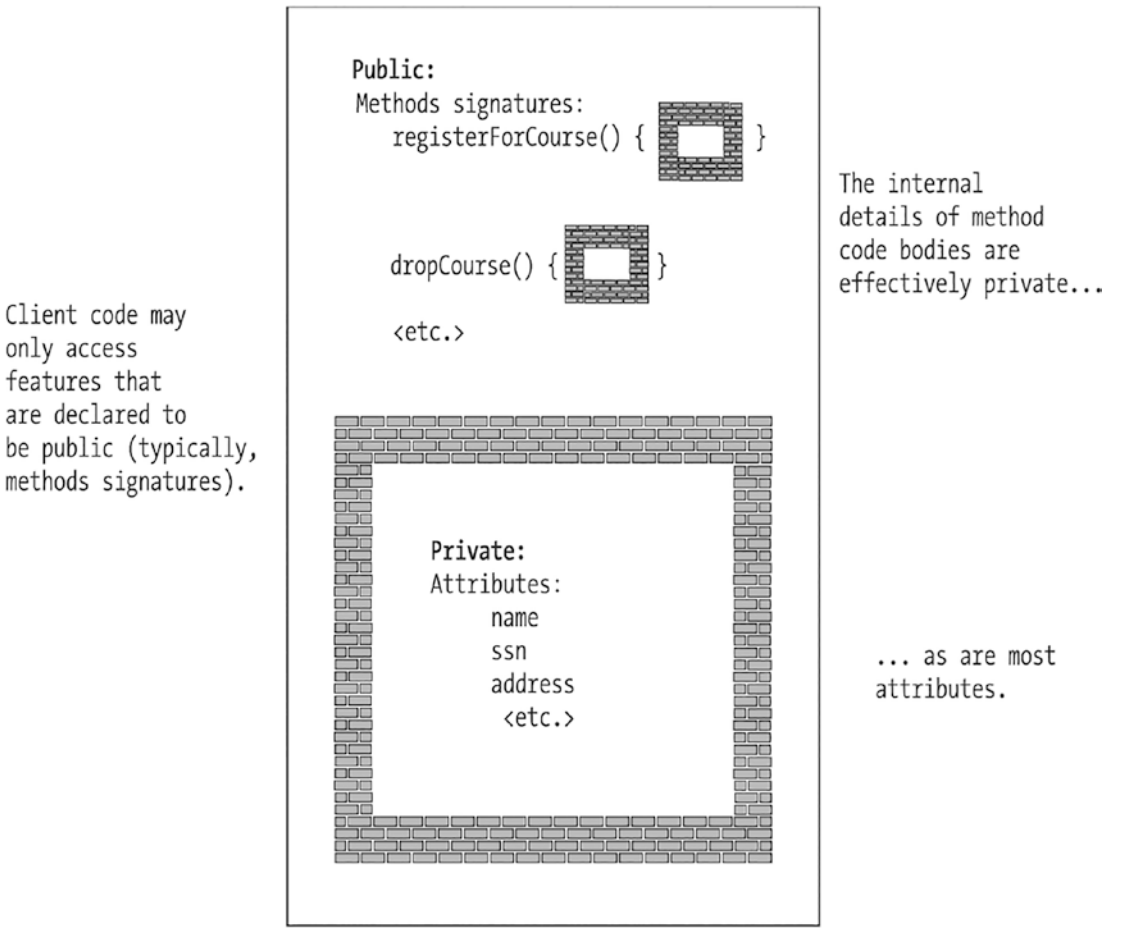


Figure 4-6. *Public vs. private visibility*

Method Headers, Revisited

Let's amend the definition of a method header from a bit earlier in the chapter. A method header actually consists of the following:

- ***A method's access modifier***
- A method's return type—that is, the data type of the information that is going to be passed back by object B to object A, if any, when the method is finished executing
- A method's name
- An optional list of comma-separated formal parameters (specifying their types and names) to be passed to the method, enclosed in parentheses

As an example, here is a typical method header that we might define for the `Student` class, with the access modifier included:

```
 public     boolean    registerForCourse (String courseID, int secNo)
 access    return type   method name    comma-separated list of formal
modifier                                  parameters, enclosed in parentheses
                                        (parentheses may be left empty)
```

Accessing the Features of a Class from Within Its Own Methods

Note that we can access all of a given class's features, ***regardless of their accessibility***, from ***within*** any of that class's ***own*** method bodies; that is, `public`/`private` designations only affect access to a feature ***from outside the class itself*** (i.e., ***from client code***).

Let's study the following example to see how one feature of a class may be accessed from within another:

```
public class Student {
    // A few private attributes.
    private String name;
    private String ssn;
```

```
    private double totalLoans;
    private double tuitionOwed;

    // Get/set methods would be provided for all of these attributes;
    // details omitted ...

    public void printStudentInfo() {
        // Accessing attributes of the Student class.
        System.out.println("Name:  " + name);
        System.out.println("Student ID:  " + ssn);
        // etc.
    }

    public boolean allBillsPaid() {
        boolean answer = false;
        // Accessing another method of the Student class.
        double amt = moneyOwed();

        if (amt == 0.0) {
          answer = true;
        }
        else {
          answer = false;
        }

        return answer;
    }

    private double moneyOwed() {
        // Accessing attributes of the Student class.
        return totalLoans + tuitionOwed;
    }
}
```

The first thing we observe is that we needn't use dot notation to access any of the features of the `Student` class from ***within*** `Student` methods. It's automatically understood by the compiler that a class is accessing one of its own features when a **simple name**—that is, a name without a dot notation prefix, also known as an **unqualified name**—is used, for example:

```
    public void printStudentInfo() {
        // Here, we're accessing the "name" attribute without dot notation.
        System.out.println("Name:  " + name);
        // etc.
    }
```

and

```
    public boolean allBillsPaid() {
        boolean answer = false;
        // Here, we're accessing the "moneyOwed" method without dot
        notation.
        double amt = moneyOwed();
        // etc.
    }
```

That being said, the Java keyword `this` can be used in dot notation fashion—`this.featureName`—within any of a class's methods to emphasize the fact that we're accessing another feature of ***this same class***. I've rewritten the `Student` example from earlier to take advantage of the `this` keyword:

```
public class Student {
    // A few private attributes.
    private String name;
    private String ssn;
    private double totalLoans;
    private double tuitionOwed;

    // Get/set methods would be provided for all of these attributes;
    // details omitted ...

    public void printStudentInfo() {
        // We've added the prefix "this.".
        System.out.println("Name:  " + this.name);
        System.out.println("Student ID:  " + this.ssn);
        // etc.
    }
```

```
    public boolean allBillsPaid() {
        boolean answer = false;
        // We've added the prefix "this.".
        double amt = this.moneyOwed();

        if (amt == 0.0) {
          answer = true;
        }
        else {
          answer = false;
        }

        return answer;
    }

    private double moneyOwed() {
        // We've added the prefix "this.".
        return this.totalLoans + this.tuitionOwed;
    }
}
```

Either approach—prefixing internal feature references with `this.` or omitting such a qualifying prefix—is acceptable; common practice is to forego the use of the `this.` prefix except when necessary to disambiguate a method parameter from a similarly named attribute. That is, it is permissible to declare a method parameter with the same name as an attribute, as illustrated by the following code:

```
public class Student {
    private String major;
    // Other attributes omitted.

    // Note that we've used "major" as the name of a parameter
    // to the following method - this duplicates the name of
    // the "major" attribute above.  This is OK, however, if
    // we use "this." within the method body below to disambiguate
    // the two.
    public void updateMajor(String major) {
        // In the next line of code, "this.major" on the left side
```

```
        // of the assignment statement refers to the ATTRIBUTE
        // named "major", whereas "major" on the right side of
        // the assignment statement refers to the PARAMETER
        // named "major".
        this.major = major;
    }

    // etc.
}
```

Of course, we could avoid having to use `this.` as a prefix simply by choosing an alternative name for our method parameter:

```
public class Student {
    private String major;
    // Other attributes omitted.

    public void updateMajor(String m) {
        // No ambiguity!
        major = m;
    }

    // etc.
}
```

It's important to avoid ***accidentally*** giving parameters/local variables names that duplicate the names of attributes, as this can lead to bugs that are hard to diagnose. For example, in the `Student` class that follows are both an ***attribute*** and a ***local variable*** named `major`. Please refer to the comments in the code example for an explanation of why this is problematic:

```
public class Student {
    // Attributes.
    private String major;

    public void updateMajor() {
        // We've inadvertantly declared a local variable, "major", with
        // the SAME name as an attribute of this class.  This is a
        BAD IDEA!
```

```
        // Note that this code will compile WITHOUT ERROR ...
        String major = null;

        // Later in the method:

        // We THINK we're updating the value of ATTRIBUTE "major" below,
        // but we're instead updating LOCAL VARIABLE "major", which will
        // go out of scope as soon as this method ends;
        // meanwhile, the value of ATTRIBUTE "major" is unchanged!
        major = major.toUppercase();
        // etc.
    }
}
```

We'll see other uses for the `this` keyword, involving code reuse and object self-referencing, later in the book.

Accessing Private Features from Client Code

If private features can't be accessed outside of an object's own methods, how does client code ever manipulate them? Through ***public*** features, of course!

Good OO programming practice calls for providing public **accessor methods** by which clients of an object can effectively manipulate selected private attributes to read or modify their values. Why is this? ***So that we may empower an object to have the "final say" in whether or not what client code is trying to do to its attributes is valid***. That is, we want an object to be involved in determining whether or not any of the business rules defined by its class are being violated. Before looking at specific examples that illustrate ***why*** this is so important, let's first discuss the mechanics of ***how*** we declare accessor methods.

Declaring Accessor Methods

The following code, excerpted from the `Student` class, illustrates the conventional accessor methods—informally known as **"get"** and **"set" methods**—that we might write for reading/writing the value of two `private` attributes of the `Student` class called `name` and `facultyAdvisor`, respectively:

```
public class Student {
    // Attributes are typically declared to be private.
    private String name;
    private Professor facultyAdvisor;
    // other attributes omitted from this example ...

    // Provide public accessor methods for reading/modifying
    // private attributes from client code.

    // Client code will use this method to read ("get") the value of the
    // "name" attribute of a particular Student object.
    public String getName() {
        return name;
    }

    // Client code will use this method to modify ("set") the value of the
    // "name" attribute of a particular Student object.
    public void setName(String newName) {
        name = newName;
    }

    // Client code will use this method to read ("get") the value of the
    // facultyAdvisor attribute of a particular Student object.
    public Professor getFacultyAdvisor() {
        return facultyAdvisor;
    }

    // Client code will use this method to modify ("set") the value of the
    // facultyAdvisor attribute of a particular Student object.
    public void setFacultyAdvisor(Professor p) {
        facultyAdvisor = p;
    }

    // etc.
}
```

The nomenclature "get" and "set" is stated from the standpoint of ***client code***: think of a "set" method as the way that ***client code*** stuffs a value ***into*** an object's attribute (see Figure 4-7)...

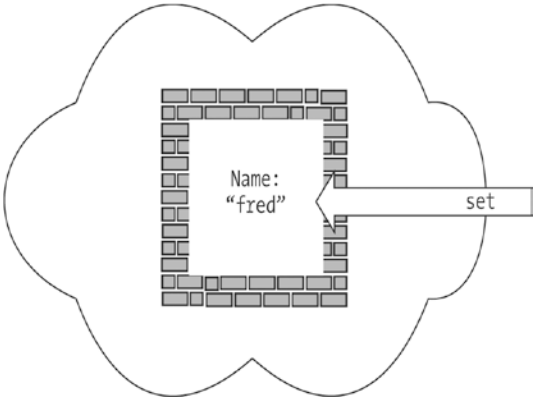


Figure 4-7. *A "set" method is used to pass data* ***into*** *an object*

... and the "get" method as the way that ***client code*** retrieves an attribute value ***from*** an object (see Figure 4-8).

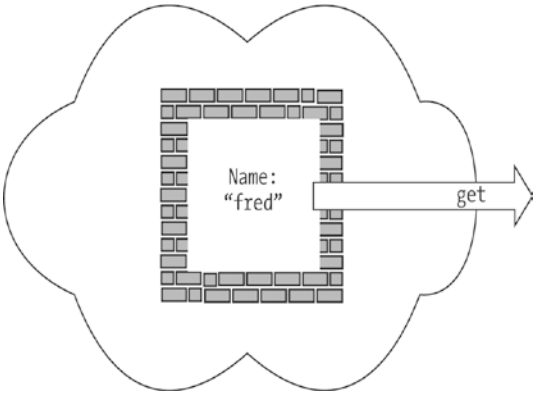


Figure 4-8. *A "get" method is used to retrieve data* ***from*** *an object*

Recommended "Get"/"Set" Method Headers

For an attribute declaration of the form

```
accessibility*  attribute-type attributeName;     * typically private
```

for example,

```
private String majorField;
```

the rules for formulating conventional accessor method headers are as follows.

For a "get" method, the formula is as follows:

```
public attribute-type getAttributeName()
```

For example:

```
public String getMajorField()
```

- The name of the method is formulated by capitalizing the first letter of the attribute name in question (e.g., `majorField`) and sticking "get" in front (e.g., `getMajorField`).
- Note that we don't typically pass any arguments into a "get" method, because all we want an object to do is to hand us back the value of one of its attributes; we don't typically need to tell the object anything special for it to know how to do this.
- Also, because we're expecting an object to hand back the value of a specific attribute, the return type of the "get" method must match the type of the attribute of interest. If we're "getting" the value of an `int` attribute, then the return type of the corresponding "get" method must be `int`; if we're "getting" the value of a `Professor` attribute, then the return type of the corresponding "get" method must be `Professor`; and so forth.
- Here's a typical "get" method in its entirety, shown in the context of the `Student` class:

```
public class Student {
    private String majorField;
    // Other attributes omitted from this example.

    public String getMajorField() {
        // Return the value of the majorField attribute.
```

```
            return majorField;
        }

        // etc.
    }
```

For a "set" method, the formula is as follows:

```
public void setAttributeName(attributeType parameterName)
```

For example:

```
public void setMajorField(String major)
```

- The name of the method is formulated by capitalizing the first letter of the attribute name in question (e.g., `majorField`) and sticking "set" in front (e.g., `setMajorField`).
- In the case of a "set" method, we must pass in the value that we want the object to use when setting its corresponding attribute value, and the type of the value that we're passing in must match the type of the attribute being set. If we're "setting" the value of an `int` attribute, then the argument that is passed into the corresponding "set" method must be an `int`; if we're "setting" the value of a `Professor` attribute, then the argument that is passed into the corresponding "set" method must be a `Professor`; and so forth.
- Since simple "set" methods are typically expected to perform their mission silently, without returning a value to the client, we typically declare "set" methods to have a return type of `void`.
- Here's a typical "set" method in its entirety, shown in the context of the `Student` class:

```
public class Student {
    private String majorField;
    // Other attributes omitted from this example.

    public String getMajorField() {
        // Return the value of the majorField attribute.
```

```
            return majorField;
        }

        public void setMajorField(String major) {
            // Assign the value passed in as an argument as the new
            value of
            // the majorField attribute.
            majorField = major;
        }
    }
```

There is one exception to the "get" method naming convention: when an attribute is of type `boolean`, it's recommended to name the "get" method starting with the verb `is` instead of with `get`. The "set" method for a `boolean` attribute would still follow the standard naming convention, however, for example:

```
public class Student {
  private boolean honorsStudent;
  // other attributes omitted from this example ...

  // Get method.  For a boolean, the method name starts with "is" vs. "get".
  public boolean isHonorsStudent() {
    return honorsStudent;
  }

  // Set method.
  public void setHonorsStudent(boolean x) {
    honorsStudent = x;
  }

  // etc.
}
```

All of the "get"/"set" method bodies that we've seen thus far are simple "one-liners": we're either returning the value of the attribute of interest with a simple `return` statement in a "get" method or copying the value of the passed-in argument to the internal attribute in a "set" method so as to store it. This isn't to imply that all "get"/"set" methods need be this simple; in fact, there are endless possibilities for what actually gets

coded in accessor methods, because, as we discussed earlier, methods must implement business rules, not only about how an object behaves but also what valid states its data can assume.

As a simple example, let's say that we always want to store a Student's name such as "Steve Barker" in the format "S. BARKER," where we abbreviate the first name to a single letter and represent the entire name in all uppercase. We might therefore wish to write the setName method of the Student class as follows:

```
public void setName(String newName) {
  // First, reformat the newName, as necessary ...
  // Pseudocode.
  if (newName contains full first name) {
    // Amend the value of newName.
    newName = newName with first name converted to a single character
              followed by a period;
  }

  // Next, convert newName to all uppercase.
  // Pseudocode.
  newName = upper case version of newName;

  // Only then do we update the name attribute with the (modified) value.
  name = newName;
}
```

IDE-Generated Get/Set Methods

With most IDEs, get and set methods are generated automatically. In such cases, the name of the parameter being passed into the set method will typically be identical to the name of the attribute, for example:

```
public class Student {
    String majorField;
// other details  omitted
public void setMajorField(String majorField) {
    this.majorField = majorField;
}
```

The use of the keyword this followed by a dot (.) in front of majorField in the preceding code indicates to the compiler that the ***local variable*** majorField's value is to be transferred to the ***attribute*** by the same name.

The "Persistence" of Attribute Values

Because I haven't explicitly stated so before, and because it may not be obvious to everyone, I'd like to call attention now to the fact that an object's attribute values persist as long as the object itself persists in memory. That is, once we instantiate a `Student` object in our application

```
Student s = new Student();
```

then any values that we assign to `s`'s attributes

```
s.setName("Mel");
```

will persist until such time as either the value is explicitly changed

```
// Renaming Student s.
s.setName("Klemmie");
```

or the object as a whole is garbage collected by the Java Virtual Machine (JVM), a process that we discussed in Chapter 3. So, to return to our analogy of objects as helium balloons from Chapter 3, as long as the "helium balloon" representing `Student s` stays "inflated," whenever we ask `s` for its name, it will "remember" whatever value we've ***last*** assigned to its `name` attribute.

Using Accessor Methods from Client Code

We already know how to use dot notation to invoke methods on objects from client code, and so we'll do the same when invoking accessor methods on object references:

```
Student s = new Student();

// Modify ("set") the attribute value.
s.setName("Joe");

// Read ("get") the attribute value.
System.out.println("Name: " +  s.getName());
```

I promised earlier in this chapter to discuss how a given `Student` can be informed as to which particular `Professor` is its `facultyAdvisor`; now that you know about "set" methods, doing so is a snap! Assuming that (a) `facultyAdvisor` is an attribute of the `Student` class declared to be of type `Professor` and (b) we've written a "set" method for this attribute with the "standard" header `public void setFacultyAdvisor(Professor p)`, here's the client code for "acquainting" students with their advisors:

```
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();
Student s4 = new Student();
// etc.

Professor p1 = new Professor();
Professor p2 = new Professor();
// etc.

// Details omitted ...

s1.setFacultyAdvisor(p1);
s2.setfacultyAdvisor(p1);
s3.setFacultyAdvisor(p2);
s4.setFacultyAdvisor(p2);
// etc.
```

The Power of Encapsulation Plus Information Hiding

You learned earlier that encapsulation is the mechanism that bundles together the state (attribute values) and behavior (methods) of an object. Now that you've gained some insights into public/private accessibility, encapsulation warrants a more in-depth discussion.

It's useful to think of an object as a "fortress" that "guards" its data—namely, the values of all of its attributes. Rather than trying to march straight through the walls of a fortress, which typically results in death and destruction (!), we ideally would approach the guard at the gate to ask permission to enter. Generally speaking, the same is true for objects: we can't directly access the values of an object's privately declared attributes without an object's permission and knowledge—that is, without using one of an object's publicly accessible methods to access the attribute's value.

Assume that you've just met someone for the first time and wish to know their name. One way to determine their name would be to reach into their pocket, pull out their wallet, and look at their driver's license—essentially, accessing their private attribute values without their permission! The more socially acceptable way would be to simply ask them for their name—akin to using their `getName` method—and to allow them to respond accordingly. They may respond with their formal name or a nickname or an alias, or they may say, "It's none of your business!"—but the important point is that you're giving the person (object) ***control*** over their response based on how the logic of the accessor method is coded.

By restricting access to an object's private attributes through public accessors, we derive three important benefits:

- Preventing unauthorized access to encapsulated data
- Helping ensure data integrity
- Limiting "ripple effects" that can otherwise occur throughout an application when the private implementation details of a class must change

Let's discuss each of these benefits in detail.

Preventing Unauthorized Access to Encapsulated Data

Some of the information that a `Student` object maintains about itself—say, the student's identification number—may be highly confidential. A `Student` object may choose to selectively pass along this information when necessary—for example, when registering for a course—but may not wish to hand out this information to any object that happens to casually ask for it.

Simply by making the attribute private, and intentionally omitting a public "get" method with which to request the attribute's value, there'd be no way for another object to request the `Student` object's identification number.

Helping Ensure Data Integrity

As mentioned previously, one of the arguments against declaring public attributes is that the object loses control over its data, for as we saw earlier, a public attribute's value can be changed by client code without regard to any business rules that the object's class

may wish to impose. On the other hand, when an accessor method is used to change the value of a private attribute, value checking can be built into the "set" method to ensure that the attribute value won't be set to an "improper" value.

As an example, let's say that we've declared a `Student` attribute as follows:

```
private String birthDate;
```

Our intention is to record birth dates in the format "mm/dd/yyyy". By requiring that client code invoke methods to manipulate the `birthDate` attribute (instead of permitting direct public access to the attribute), we can provide logic within those methods to validate the format of any newly proposed birth date and reject those that are invalid. We'll illustrate this concept by declaring an `updateBirthDate` method for the `Student` class as shown in the following code:

```
public class Student {
    private String birthDate;
    // other attributes omitted from this example ...

    public boolean updateBirthDate(String newBirthDate) {
        boolean newDateApproved;

        // Perform appropriate validations.
        // Remember, italics represent pseudocode ...
        if (date is not in the format mm/dd/yyyy) {
            newDateApproved = false;
        }
        else if (mm not in the range 01 to 12) {
            newDateApproved = false;
        }
        else if (the day number isn't valid for the selected month) {
            newDateApproved = false;
        }
        else if (the year is NOT a leap year, but 2/29 was specified) {
            newDateApproved = false;
        }
```

```
        // etc. for other validation tests.
        else {
            // If we've gotten this far in the code, all is well with
            what was
            // passed in as a value to this method, and so we can go
            ahead and
            // update the value of the birthDate attribute with this value.
            birthDate = newBirthDate;

            // Set our flag to indicate success!
            newDateApproved = true;
        }

        return newDateApproved;
    }

    // etc.
}
```

If an attempt is made to pass an improperly formatted birth date to the method from client code, as in

```
s.updateBirthDate("foo");
```

the change will be rejected and the value of `s`'s `birthDate` attribute will be unchanged. In fact, we'd probably insert the attempt to update the birth date within an "if" statement so that we could detect and react to such a rejection:

```
// Somewhere along the line, the newDate variable takes on an
invalid value.
String newDate = "Jan 1 1990";

// Later in the application ...
if (!(s.updateBirthDate(newDate)) {
    // Pseudocode.
    do whatever we need to do if value is rejected ...
}
```

On the other hand, if `birthDate` had been declared to be a ***public*** attribute of the `Student` class, then setting the attribute directly as follows would be permitted by the compiler:

```
s.birthDate = "Jan 1 1990";
```

Hence, it would be possible to corrupt the attribute's value by bypassing the error checking, based on business rules, that a "set" method would normally perform for us.

Limiting "Ripple Effects" When Private Features Change

Despite our best attempts to avoid such situations, we often have a need to go back and modify the design of an application after it has been deployed, either when an inevitable change in requirements occurs or if we unfortunately discover a design flaw that needs attention. Unfortunately, in a non-OO (or poorly designed OO) application, this can open us up to "ripple effects," wherein dozens or hundreds or ***thousands*** of lines of code throughout an application have to be changed, retested, etc.

One of the most dramatic examples of the negative impact of a design change was the notorious **Y2K problem**. When the need to change date formats to accommodate a four-digit year arose as the year 2000 approached, the burden to hunt through ***billions*** of lines of code in ***millions*** of applications worldwide to find all such cases—and to ***fix*** them without unintentionally ***breaking*** anything else—was mind-boggling. Many folks were convinced at the time that the world would actually melt down as a result, and in fact, it's quite amazing that it didn't!

Perhaps the most ***dramatic*** benefit of encapsulation combined with information hiding, therefore, is that the ***hidden implementation details of a class***—that is, its private data structure and/or its (effectively private) accessor code—***can change without affecting how client code interacts with objects belonging to that class***. To illustrate this principle, we'll craft an example.

Let's say that an attribute is declared in the `Student` class as follows

```
private int age;
```

and that we declare a corresponding `getAge()` method as follows:

```
public int getAge() {
  return age;
}
```

(We've chosen not to declare a `setAge` method because we've decided that we want `age` to be a read-only attribute.)

We then proceed to use our `Student` class in countless applications; so, in literally ***thousands*** of places within the client code of these applications, we write statements such as the following, relying on the "get" method to provide us with a student's age as an `int` value, for example:

```
int currentAge = s.getAge();
```

A few years later, we decide to modify the data structure of the `Student` class so that, instead of maintaining an `age` attribute explicitly, we instead use the student's `birthDate` attribute to compute a student's age whenever it's needed. We thus modify our `Student` class code as follows:

Table 4-1. *Modifying Private Details of the Student Class: A Before vs. After View*

The "Before" Code	The "After" Code

`public class Student {`
`  // We have an explicit`
`  // age attribute.`
`  private int age;`
 	
`  public int getAge() {`
`  return age;`
`  }`
`  // etc.`
`}`
	`import java.util.Date;`
	`public class Student {`
	`  // We replace age with`
	`  // birthDate.`
	`  private Date birthDate;`
	
	`  public int getAge() {`
	`  // Compute the age on demand`
	`  // (pseudocode).`
	`  return system date - birthDate;`
	`  }`
	`  // etc.`
	`}`

In the "after" version of `Student`, we're computing the student's age by subtracting their birth date (stored as an attribute value) from today's date. This is an example of what can be informally referred to as a **pseudoattribute**—to client code, the presence of a `getAge()` method implies that there is an attribute by the name of age, when in fact there may not be.

The beauty is that ***we don't care*** that the ***private*** details of the `Student` class design have changed! In all of the thousands of places within the client code of countless applications where we've used code such as

```
int currentAge = s.getAge();
```

to retrieve a student's age as an `int` value, this code will continue to work ***as is***, without any changes to client code being necessary, because the expression

```
s.getAge()
```

still evaluates to an `int` value representing `Student s`'s age. Hence, we've avoided "dreaded" ripple effects and have ***dramatically*** reduced the amount of effort necessary to accommodate a design change. Such changes are said to be **encapsulated**, or limited to the internal code of the `Student` class only.

Of course, all bets are off if the developer of a class changes one of its ***public*** features—most often, a public method header—because then all of the client code that passes messages to objects of this type using this method will potentially have to change. For example, if we were to change the `Student` class design so that the `getAge()` method is now declared to return a `double` value, as follows

```
public class Student {
    // We've changed the type of the age attribute from int to double ...
    private double age;

    // ... and the return type of the getAge() method accordingly.
    public double getAge() {
        return age;
    }

    // etc.
```

then much of our client code ***would*** indeed potentially "break," as in the following example:

```
// This will no longer compile!
int currentAge = s.getAge();
```

This particular client code will "break" because we now have a type mismatch. We are getting back a `double` value, but are trying to assign it to an `int` variable, which as we learned in Chapter 2 will generate a compiler error as follows:

```
possible loss of precision
found    : double
required : int
```

We'd have to hunt for all of the countless instances throughout potentially many applications where we are calling the `getAge()` method on a `Student` reference and modify each such line of code to either do an explicit cast from `double` to `int`, as follows:

```
// We're now using a cast.
int currentAge = (int) s.getAge();
```

Thus, we'd potentially incur a significant ripple effect. But again, this ripple effect is due to the fact that we changed a ***public*** feature of our class—a public method header, to be precise.

As long as we restrict our changes to the private features of a class, ripple effects aren't an issue; any client code that was previously written to use `public Student` methods will continue to work as intended.

Using Accessor Methods from Within a Class's Own Methods

Earlier in the chapter, we discussed the fact that a class is permitted to directly access its own attributes by name, as in the following `printStudentInfo` method:

```
public class Student {
    private String name;
    private String ssn;
    // etc.

    // Details omitted.
```

```
    public void printStudentInfo() {
        // We're accessing our own attributes directly.
        System.out.println("Name:  " + this.name);
        System.out.println("Student ID:  " + this.ssn);
        // etc.
    }

    // etc.
```

However, it's considered to be a best practice for a class to ***use its own "get"/"set" methods*** whenever it needs to access one of its own attribute values. Let's revise the `printStudentInfo` method to illustrate this best practice:

```
public class Student {
    private String name;
    private String ssn;
    // etc.

    // "Garden variety" accessor methods.

    public String getName() {
        return name;
    }

    public void setName(String n) {
        name = n;
    }

    public String getSsn() {
        return ssn;
    }

    public void setSsn(String s) {
        ssn = s;
    }

    public void printStudentInfo() {
        // We're now using our own "get" methods to access our own
        // attribute values.
        System.out.println("Name:  " + this.getName());
```

```
        System.out.println("Student ID:  " + this.getSsn());
        // etc.
    }

    // etc.
```

Why is it important to use a class's own "get"/"set" methods rather than accessing attributes directly? Let's say that, at some future date, the `getName` and `getSsn` methods of the `Student` class are modified as follows:

```
    public String getName() {
        // Business rules have changed!  We now want to reformat the name
        // as stored within a Student object before returning it to
        // client code.
        // (Pseudocode.)
        String reformattedName = name reformatted in the form
            "LastName, FirstName";
        return reformattedName;
    }

    public String getSsn() {
        // Business rules have changed!  We now want to reformat the
        // ssn as stored within a Student object to insert dashes
        // before returning it to client code.
        // (Pseudocode.)
        String reformattedSsn = ssn reformatted in the form "xxx-xx-xxxx";
        return reformattedSsn;
    }
```

Because we've redesigned the `printStudentInfo` method to invoke `this.getName()` and `this.getSsn()`, we'll automatically benefit from the changes in business logic within the `getName` and `getSsn` methods:

```
// Client code.
Student s = new Student();
s.setName("Susan Yamate");
s.setSsn("123456789");
s.printStudentInfo();
```

Here's the output:

```
Name:  Yamate, Susan
Student ID: 123-45-6789
```

On the other hand, if we had accessed the `name` and `ssn` attributes directly from within the `printStudentInfo` method

```
public void printStudentInfo() {
    // We're accessing private attributes directly by name rather than
    // using the corresponding get methods.
    System.out.println("Name:  " + name);
    System.out.println("Student ID:  " + ssn);
    // etc.
}
```

we would ***not*** benefit from changes in business logic:

```
// Client code.
Student s = new Student();
s.setName("Susan Yamate");
s.setSsn("123456789");
s.printAllAttributes();
```

Here's the output (***incorrectly*** formatted):

```
Name:  Susan Yamate
Student ID: 123456789
```

The same holds true for using a class's own ***"set"*** methods when ***updating*** the value of an attribute from within another method—for example, this version of the `assignMajor` method of `Student`

```
public class Student {
    private String name;
    private String ssn;
    private String major;
```

```
    private Professor advisor;
    // etc.

    // Set/get methods provided; details omitted.

    public void assignMajor(String m, Professor p) {
        // Preferred.
        this.setMajor(m);
        this.setAdvisor(p);
    }

    // etc.
```

is preferred over this version

```
public class Student {
    private String name;
    private String ssn;
    private String major;
    private Professor advisor;
    // etc.

    // Set/get methods provided; details omitted.

    public void assignMajor(String m, Professor p) {
        // Not as desirable.
        this.major = m;
        this.advisor = p;
    }

    // etc.
```

because the `Student` class's "set" methods may be simple "one-liners" today

```
public void setMajor(String m) {
    major = m;
}
```

but may be enhanced to reflect more sophisticated business logic at some point in the future:

```
public void setMajor(String m) {
    // Pseudocode.
    look up m in a database to verify that it is an "approved" major
    designation
    before updating the major attribute

    if (m is valid) {
        major = m;
    }
}
```

Of course, the one place where we ***cannot*** invoke a class's "get"/"set" methods is ***within*** the "get"/"set" methods themselves. To do so would result in an infinitely recursive method

```
// This method is recursive!
public void setName(String n) {
    this.setName(n);
}
```

which will compile properly, but will produce a run-time error when invoked, as follows:

```
// Client code.
Student s = new Student();
s.setName("Fred Schnurd");
```

Here's the error:

```
Exception in thread "main" java.lang.StackOverflowError
       at Student.setName(Student.java:8)
       at Student.setName(Student.java:8)
       at Student.setName(Student.java:8)
       at Student.setName(Student.java:8)
       at Student.setName(Student.java:8)
       (repeated 1024 times!)
```

Exceptions to the Public/Private Rule

Even though it's often the case that

- Attributes are declared to be `private`
- Methods are declared to be `public`
- Private attributes are accessed through public methods

there are numerous exceptions to this rule.

Exception #1–Internal housekeeping attributes: An attribute may be used by a class strictly for internal housekeeping purposes. (Like the dishwashing detergent you keep under the sink, guests needn't know about it!) For such attributes, we needn't bother to provide public accessors.

One example for the `Student` class might be an `int` attribute `countOfDsAndFs`, used to keep track of how many poor grades a student has received in order to determine whether or not the student is on academic probation. We may in turn provide a `Student` class method `onAcademicProbation` as follows:

```
public class Student {
  // Private housekeeping attribute.
  private int countOfDsAndFs;
  // other attributes omitted from this example ...

  public boolean onAcademicProbation() {
    boolean onProbation = false;

    // If the student received more than three substandard grades,
    // he or she will be put on academic probation.
    if (countOfDsAndFs > 3) {
      onProbation = true;
    }

    return onProbation;
  }

  // other methods omitted from this example ...
}
```

The onAcademicProbation method uses the value of private attribute countOfDsAndFs to determine whether a student is on academic probation, but no ***client code*** need ever know that there is such an attribute as countOfDsAndFs, and so no explicit public accessor methods are provided for this attribute. Such attributes are instead set as a ***side effect*** of performing some ***other*** method, as in the following example, also taken from the Student class:

```
public void completeCourse(String courseName, int creditHours, char
grade) {
  // Updating this private attribute is considered to be a
  // "side effect" of completing a course.
  if (grade == 'D' || grade == 'F') countOfDsAndFs++;

  // Other processing details omitted from this example ...
}
```

Exception #2–Internal housekeeping methods: Some methods may be used strictly for internal housekeeping purposes, as well, in which case these may also be declared private rather than public. (A neighbor needn't know that we have a maid who comes to clean every other week!)

An example of such a Student method might be updateGpa, which recomputes the value of the gpa attribute each time a student completes another course and receives a grade. The only time that this method may ever need to be called is perhaps from within another method of Student—for example, the public completeCourse method—as follows:

```
public class Student {
  private double gpa;
  private int totalCoursesTaken;
  private int totalQualityPointsEarned;
  private int countOfDsAndFs;
  // other details omitted ...

  public void completeCourse(String courseName,
      int creditHours, char grade) {
    if (grade == 'D' || grade == 'F') {
      countOfDsAndFs++;
    }
```

```
    // Record grade in transcript.
    // details omitted ...

    // Update an attribute ...
    totalCoursesTaken = totalCoursesTaken + 1;

    // ... and call a PRIVATE housekeeping method from within this
    // public method to adjust the student's GPA accordingly.
    updateGpa(creditHours, grade);
  }

  // The details of HOW the GPA gets updated are a deep, dark
  // secret! Even the EXISTENCE of this next method is hidden from
  // the "outside world" (i.e., inaccessible from client code) by
  // virtue of its having been declared to be PRIVATE.
  private void updateGpa(int creditHours, char grade) {
    int letterGradeValue = 0;

    if (grade == 'A') letterGradeValue = 4;
    if (grade == 'B') letterGradeValue = 3;
    if (grade == 'C') letterGradeValue = 2;
    if (grade == 'D') letterGradeValue = 1;
    // For an 'F', it remains 0.

    int qualityPoints = creditHours * letterGradeValue;

    // Update two attributes.
    totalQualityPointsEarned =
         totalQualityPointsEarned + qualityPoints;
    gpa = totalQualityPointsEarned/totalCoursesTaken;
  }
}
```

Client code shouldn't be able to directly cause a `Student` object's `gpa` to be updated; this should only occur as a side effect of completing a course. By making the `updateGpa` method private, we've prevented any client code from explicitly invoking this method to manipulate this attribute's value out of context.

Exception #3–"Read-only" attributes: If we provide only a "get" method for an attribute, but no "set" method, then that attribute is rendered effectively read-only from the perspective of client code. We might do so, for example, with a student's ID number that, once set by the system at the time of Student object creation, should remain unchanged:

```
public class Student {
  private String studentId;
  // details omitted

  // We render studentId as a read-only attribute by only writing a
     get method
  // for it.
  public String getStudentId() {
    return studentId;
  }

  // The set method is intentionally omitted from the class.
}
```

How do we ***ever*** set such an attribute's value initially? We've already seen that some attributes' values get modified as a side effect of performing a method (as with the `countOfDsAndFs` attribute that we discussed earlier). We'll also see how to explicitly initialize such a read-only attribute a bit later in this chapter, when we talk about constructors.

Exception #4–Public attributes: On rare occasions, a class may declare selected attributes as `public` for ease of access; ***this is only done when there is no business logic governing the attributes per se***.

One such example is the predefined Java `Point` class, which is used to define an (x, y) coordinate in two-dimensional space; its attributes are declared simply as

```
public class Point {
    // Both attributes are public:
    public double x;
    public double y;

    // etc.
}
```

so that, in client code, we may easily assign values as follows:

```
Point p = new Point();
p.x = 3.7;
p.y = -4.8;
```

That being said, when creating your own classes, ***resist the urge*** to declare attributes with `public` accessibility simply as a lazy way of avoiding having to write "get"/"set" methods! We've seen the many benefits that "get"/"set" methods provide in terms of enforcing business logic when appropriate. As it turns out, the vast majority of attributes will need to be governed by such business logic.

Constructors

When we talked about instantiating objects in the previous chapter, you might have been curious about the interesting syntax involved with the `new` keyword:

```
Student x = new Student();
```

In particular, you might have wondered why there were parentheses tacked onto the end of the statement.

It turns out that when we instantiate an object via the `new` keyword, we're actually invoking a special type of function affiliated with a class called a **constructor**. Invoking a constructor serves as a request to the JVM to construct (instantiate) a brand-new object at run time by allocating enough program memory to house the object's attributes. Returning to our "object as a helium balloon" analogy, we're asking the JVM to inflate a new helium balloon of a particular type.

Default Constructors

If we don't explicitly declare any constructors for a class, Java automatically provides a **default constructor** for that class. The default constructor is parameterless—that is, it takes no arguments—and does the "bare minimum" required to initialize a new object: namely, setting all attributes to their zero-equivalent default values.

Thus, even though we may have designed a class with no explicit constructors whatsoever, as with the following `Student` class

```
public class Student {
  // Attributes.
  private String name;
  // other details omitted ...

  // We've declared methods, but NO EXPLICIT CONSTRUCTORS.

  public String getName() {
    return name;
  }

  public void setName(String newName) {
      name = newName;
  }

  // etc.
}
```

we are still able to write client code to instantiate a "bare-bones" `Student` object as follows

```
Student s1 = new Student();
```

because the JVM uses the default constructor for the `Student` class.

Writing Our Own Explicit Constructors

We needn't rely on Java to provide a default constructor for each of our classes; we can instead write constructors of our own design for a particular class if we wish to do something more interesting/complex to initialize an object when it is first instantiated.

Note that the header syntax for a constructor is a bit different from that of a method:

```
  public          ____________        Student()
  access      NO return type!     constructor name must match
 modifier                          class name, followed by
                                   comma-separated list of formal
                                   parameters enclosed in ()
```

- A constructor's name must be exactly the same as the name of the class for which we're writing the constructor—we have no choice in the matter.
- A parameter list, enclosed in parentheses, is provided for a constructor header as with method headers. And, as with method headers, the parameter list may be left empty if appropriate.
- We ***cannot*** specify a return type for a constructor; by definition, a constructor returns a reference to a newly created object of the type represented by the class to which the constructor belongs. That is, a constructor of the form

  ```
  // Note:  no return type!
  public Student() { ... }
  ```

  returns a newly instantiated `Student` object reference. A constructor of the form

  ```
  // Note:  no return type!
  public Professor() { ... }
  ```

  returns a newly instantiated `Professor` object reference. And so forth.

Another disparity with respect to constructor syntax as compared with that of methods is that invoking a constructor does not involve dot notation:

```
Professor p = new Professor();
```

This is because we aren't requesting a service of a particular object; rather, we're requesting that a brand-new object be crafted by the JVM.

Passing Arguments to Constructors

One of the most common motivations for declaring an explicit constructor for a class is to provide a convenient way to pass in initial values for an object's attributes at the time of instantiation.

If we use a default constructor to instantiate a bare-bones object, we then must invoke the object's "set" methods one by one to initialize its attribute values, as illustrated by this next snippet:

```
  // Create a bare bones Student object.
  Student s = new Student();

  // Initialize the attributes one by one.
  s.setName("Fred Schnurd");
  s.setSsn("123-45-6789");
  s.setMajor("MATH");
  // etc.
```

This can be rather tedious if there are a lot of attributes to initialize.

Alternatively, if we design a constructor that accepts arguments, we can simultaneously instantiate an object and provide meaningful initial attribute values in a single line of code, for example:

```
// This single line of code replaces the previous four lines.
Student s = new Student("Fred Schnurd", "123-45-6789", "MATH");
```

In order to accomplish this, we'd of course have to declare a `Student` class constructor with an appropriate header, as shown here:

```
public class Student {
  // Attributes.
  private String name;
  private String ssn;
  private String major;
  // etc.

  // We've declared a constructor that accepts three arguments, to
  accommodate
  // passing in three attribute values.
  public Student(String s, String n, String m) {
    this.setName(n);
    this.setSsn(s);
    this.setMajor(m);
  }

  // etc.
```

Constructor arguments can also be used as control flags for influencing how a constructor behaves, as illustrated in the next example constructor:

```
public Student(String name, boolean assignDefaults) {
  setName(n);
  if (assignDefaults) {
    this.setSsn("?");
    this.setMajor("UNDECLARED");
  }
}
```

Client code for the preceding might look as follows:

```
// We DO want to assign default values to other attributes.
Student s = new Student("Cynthia Coleman", true);
```

Replacing the Default Parameterless Constructor

If we wish, we can explicitly program a parameterless constructor for our classes to do something more interesting than merely instantiating a bare-bones object, thereby ***replacing*** the default parameterless constructor with one of our own design. This is illustrated by the following class:

```
public class Student {
    // Attributes.
    private String name;
    private String major;
    // etc.

    // We've explicitly programmed a parameterless constructor, thus
    replacing
    // the default version.
    public Student() {
        // Perhaps we wish to initialize attribute values to something
        other than
        // their zero equivalents.
        this.setName("?");
        this.setMajor("UNDECLARED");
```

```
        // etc.
    }

    // Other methods omitted from this example.
}
```

More Elaborate Constructors

We can program a constructor to do whatever makes sense in constructing a new `Student`:

- We may wish to instantiate additional objects related to the `Student` object:

```
public class Student() {
  // Every Student maintains a handle on his/her own
  individual Transcript
  // object.
  private Transcript transcript;

  public Student() {
    // Create a new Transcript object for this new
    Student.
    transcript = new Transcript();
    // etc.
  }

  // etc.
}
```

- We may wish to access a relational database to read in the data needed to initialize the `Student`'s attributes:

```
public class Student {
  // Attributes.
  String studentId;
  String name;
  double gpa;
  // etc.
```

```
  // Constructor.
  public Student(String id) {
    studentId = id;

    // Pseudocode.
    use studentId as a primary key to retrieve data from
    the Student table of a
    relational database;

    if (studentId found in Student table) {
      retrieve all data in the Student record;
      name = name retrieved from database;
      gpa = value retrieved from database;
      // etc.
    }
  }

  // etc.
}
```

- We may wish to communicate with other already existing objects to announce a new Student's existence:

```
public class Student {
  // Details omitted.

  // Constructor.
  public Student(String major) {
    // Alert the student's designated major department that a new
    student has
    // joined the university.
    // Pseudocode.
    majorDept.notify(about this student ...);

    // etc.
  }

  // etc.
}
```

- And so forth—***whatever*** is required of our application.

We'll see examples of such constructors later in the book, when we craft the SRS.

Overloading Constructors

Just as we are permitted to overload methods in Java, we are permitted to overload constructors. That is, we may write as many different constructors for a given class as we wish, as long as they have different argument signatures.

Here is an example of a `Student` class that declares three different constructors:

```
public class Student {
  private String name;
  private String ssn;
  private int age;
  // etc.

  // Constructor #1:  takes no arguments; supercedes the default
  constructor.
  public Student() {
    // Assign default values to selected attributes, if desired.
    this.setSsn("?");
    // Those that aren't explicitly initialized in the constructor will
    // automatically assume the zero-equivalent value for their
    respective type.
  }

  // Constructor #2:  takes a single String argument.
  public Student(String s) {
    this.setSsn(s);
  }

  // Constructor #3:  takes two Strings and an int as arguments.
  public Student(String s, String n, int i) {
    this.setSsn(s);
    this.setName(n);
    this.setAge(i);
  }
```

```
  // Other methods omitted from this example.
}
```

By overloading the constructor for a class, we make the class more versatile by giving client code a variety of constructors to choose from, depending on the circumstances. Here is an example of client code illustrating the use of all three forms of `Student` constructor:

```
// We don't know ANYTHING about our first student, so we use the
// parameterless constructor to instantiate s1.
Student s1 = new Student();

// We know the ssn (only) for our second student, and so we use the second
// form of constructor to instantiate s2.
Student s2 = new Student("123-45-6789");

// We know the ssn, name, and age of our third student, and so we use
// the third form of constructor to instantiate s3.
Student s3 = new Student("987-65-4321", "John Smith", 21);
```

As with overloaded methods, the compiler is able to unambiguously match up which version of constructor is being invoked in each case based on the argument signatures:

- `()`: No arguments tell the compiler that we are invoking constructor #1.
- `("123-45-6789")`: One `String` argument tells the compiler that we are invoking constructor #2.
- `("987-65-4321", "John Smith", 21)`: Two `String`s and an `int` as arguments tell the compiler that we are invoking constructor #3.

This example also reinforces why no two constructors may have the same argument signature. If we ***were*** permitted to introduce a fourth constructor whose argument signature duplicated that of constructor #2, for example

```
// Constructor #4:  takes a single String argument, thereby duplicating the
// argument signature of constructor #2.
public Student(String n) {  // THIS WON'T COMPILE!!!
    this.setName(n);
}
```

then the compiler would not know which constructor—#2 or #4—we're trying to invoke in the following client code:

```
// Pseudocode.
Student x = new Student(aStringExpression);
```

So, to avoid such an ambiguous situation, the compiler generates an error message on the preceding declaration of constructor #4, as follows:

```
Student(java.lang.String) is already defined in Student
        public Student(String n) { }
               ^
```

An Important Caveat Regarding the Default Constructor

There is one very important caveat about default constructors in Java: if we declare ***any*** of our own constructors for a class, with ***any argument signature***, then the default parameterless constructor is ***not*** automatically provided. This is by design, because it is assumed that if we've gone to the trouble to program any constructors whatsoever for a class, then we must have some special initialization requirements for that class that the default constructor could not possibly anticipate.

The implication of this language feature is as follows: if we want or need a constructor that accepts no arguments for a particular class ***along with*** other versions of constructors that ***do*** take arguments, ***we must explicitly program a parameterless constructor***. To illustrate this point, let's consider a `Student` class that only declares one explicit constructor:

```
public class Student {
    // Details omitted.

    // Only one constructor is explicitly declared, and which takes a
    // single String argument.
    public Student(String s) {
        this.setSsn(s);
    }

    // etc.
}
```

In client code, we may instantiate a `Student` based on this class as follows:

```
Student s = new Student("123-45-6789");
```

But if we try to use the (what is now nonexistent) default constructor

```
Student s = new Student();
```

we'll get the following compilation error:

```
cannot find symbol
symbol  : constructor Student()
location: class Student
    Student s = new Student();
                ^
```

Generally speaking, it is considered a best practice to always explicitly provide a parameterless constructor (to replace the lost default) if we are providing ***any*** constructors for a class at all. We'll revisit the importance of this practice when we discuss **inheritance** in Chapter 5.

One common mistake made by beginning Java programmers is to accidentally declare a return type in a constructor header, for example:

```
public void Student() { ... }
```

This is a particularly difficult bug to track down, because while such header declarations will ***compile***, they are viewed by the compiler as ***methods*** and ***not*** as ***constructors*** and cannot be used as such. What's worse, developers will ***think*** that they've programmed a parameterless constructor when in fact they haven't; any attempt to ***use*** such a constructor in their application will meet with the following seemingly cryptic compilation error message:

```
Student s = new Student();
```

This is the compiler error:

```
cannot find symbol
symbol:  constructor Student()
location:  class Student
Student s = new Student();
                          ^
```

Using the "this" Keyword to Facilitate Constructor Reuse

Earlier in this chapter, we covered the `this` keyword and illustrated how it can be used to optionally qualify features of a class when accessed from within methods of the same class, as in

```
public class Student {
    // Details omitted.

    public void printAllAttributes() {
        System.out.println("Name:  " + this.getName());
        System.out.println("Student ID:  " + this.getSsn());
        // etc.
    }
}
```

We're now going to explore a second alternative use of the `this` keyword, related to reusing code from one constructor by another within the same class.

It's conceivable that if we've overloaded the constructor for a class, there will be some common initialization steps required of all versions. For example, let's say that, for all new students, we must

- Alert the registrar's office of this student's existence.
- Create a transcript for this student.

If we were to declare three constructors for the `Student` class, it would be tedious to duplicate the same logic across all three (see the **bolded** lines of code):

```
public class Student {
    // Attribute details omitted.

    // Constructor #1.
    public Student() {
        // Assign default values to selected attributes ... details
        omitted.

        // Pseudocode.
        alert the registrar's office of this student's existence

        // Create a transcript for this student.
        transcript = new Transcript();
    }

    // Constructor #2.
    public Student(String s) {
        this.setSsn(s);

        // This code is duplicated from above!
        // Pseudocode.
        alert the registrar's office of this student's existence

        // Create a transcript for this student.
        transcript = new Transcript();
        // end of code duplication
    }

    // Constructor #3.
    public Student(String s, String n, int i) {
        this.setSsn(s);
        this.setName(n);
        this.setAge(i);

        // DUPLICATION YET AGAIN!!!
        // Pseudocode.
        alert the registrar's office of this student's existence
```

```
        // Create a transcript for this student.
        transcript = new Transcript();
        // end of code duplication
    }

    // etc.
}
```

Worse yet, if the logic needed to change, we'd have to change it in all three constructors. Fortunately, the `this` keyword comes to our rescue. From within any constructor of a class X, we can invoke any other constructor of the same class X via the following syntax:

`this(`*`argument signature`*`);`

Let's rewrite our previous three `Student` constructors so that constructor #2 takes advantage of the logic of #1 and #3 takes advantage of #2:

```
public class Student {
    // Attribute details omitted.

    // Constructor #1.
    public Student() {
        // Assign default values to selected attributes ... details
        omitted.

        // Do the things common to all three constructors in this first
        // constructor.
        // Pseudocode.
        alert the registrar's office of this student's existence

        // Create a transcript for this student.
        transcript = new Transcript();
    }

    // Constructor #2.
    public Student(String s) {
        // REUSE the code of the first constructor within the second!
        this();  // invoking the parameterless constructor
```

```
        // Then, do whatever else extra is necessary for constructor #2.
        this.setSsn(s);
    }

    // Constructor #3.
    public Student(String s, String n, int i) {
        // REUSE the code of the second constructor within the third!
        this(s);  // Invoking the constructor with one String argument

        // Then, do whatever else extra is necessary for constructor #3.
        this.setName(n);
        this.setAge(i);
    }

    // etc.
}
```

By invoking `this();` from within constructor #2 and this(s); from within constructor #3, we were able to eliminate all duplication of code.

When using the `this(...);` syntax to reuse code from one constructor to another, note that the statement must be the ***first*** statement in the constructor; that is, the following code will not compile:

```
    // Constructor #3.
    public Student(String s, String n, int i) {
        // Do whatever extra is necessary for constructor #3;
        // details omitted.
        ...

        // Then, attempt to reuse the code of constructor #2;
        // THIS NEXT LINE WON'T COMPILE!
        this(s);
    }
```

Here's the error:

```
call to this must be first statement in constructor
    this(s);
        ^
```

We'll revisit the `this` keyword yet again in Chapter 13, to discuss a third context in which it can be used.

Software at Its Simplest, Revisited

As we discussed in Chapter 3, software at its simplest consists of two primary components: ***data*** and ***functions*** that operate on that data (see Figure 4-9).

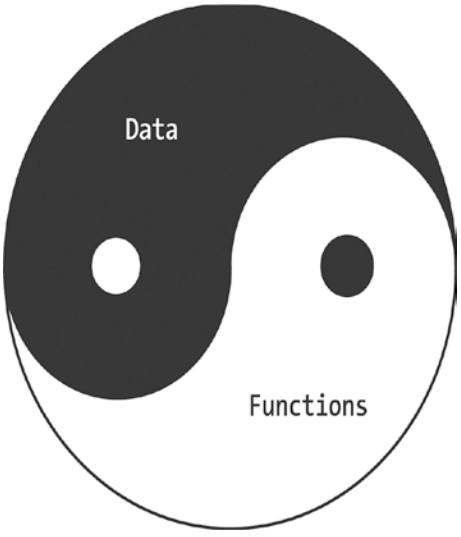


Figure 4-9. *At it simplest, software consists of data and functions that operate on that data*

We also compared the functional decomposition approach of designing software with the object-oriented approach. By way of review

With the functional decomposition approach to software development, our primary focus was on the functions that an application was to perform; data was an afterthought.

That is,

- Data was passed around from one function to the next.
- Data structure thus had to be understood in ***many*** places—that is, by many functions—throughout an application.

- If an application's data structure had to change after the application was deployed, nontrivial ripple effects often arose throughout the application.
- If data integrity errors arose as a result of faulty logic after an application had been fully integrated, it was often very difficult to pinpoint precisely where—that is, ***in which specific function(s)***—the error might have occurred.

  We now know that by taking advantage of the mechanisms of encapsulation plus information hiding, the object-oriented approach to software development remedies the vast majority of these shortcomings:

- Data is ***encapsulated*** inside of objects as ***attributes***, and, if we declare these attributes as having ***private*** accessibility, then the data structure has to be understood only by the object/***class*** to which the data belongs.
- If the (private) attribute declarations of a class have to change after an application has been deployed—as was the case when we modified the data structure of the `Student` class, replacing an `int age` attribute with a `Date birthDate` attribute—there are virtually ***no ripple effects***: only the internal logic of the affected class's methods must change.

  (Recall that we modified the internal workings of the `getAge` method in one of our `Student` class examples, but that ***none*** of the client code that called `getAge` had to change, because we hadn't altered the ***public*** method signature of the method.)

- ***Each class is responsible for ensuring the integrity of its object's data***. Thus, if data integrity errors arise within a given object's data, we can pretty much assume that it is the class to which the object belongs whose method logic is faulty.

  (Recall the `updateBirthdate` method from an earlier `Student` class example. This method contained all sorts of validity checks to ensure that the `String` being passed in as an argument represented a valid birth date. Had an ***invalid*** birth date somehow crept in, we'd know that there was something faulty about the validation logic of the `updateBirthdate` method in particular.)

If every software application consists of data and functions that operate on that data, then ***an object can be thought of as a sort of "mini application"*** whose methods (functions) operate on its attributes (data), as shown in Figure 4-10. You'll learn in Chapter 5 how such objects "join forces" to collaborate on accomplishing the overall mission of an application.

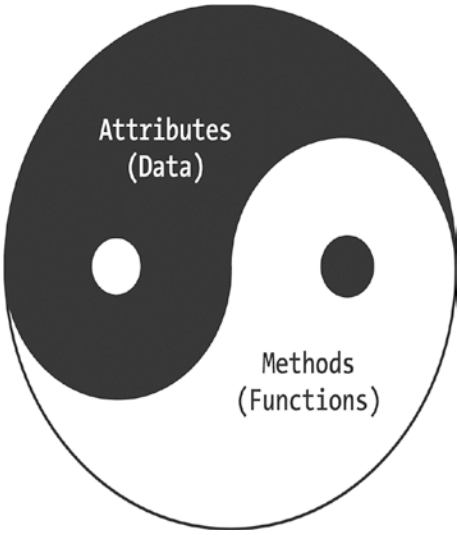


Figure 4-10. *An object is a "mini application" that encapsulates data and functions*

Summary

In this chapter, you've learned

- How to formally specify method headers, the "language" with which services may be requested of an object, and how to formulate messages—using dot notation—to actually request an object to perform such services
- That multiple objects often have to collaborate in carrying out a particular system function, such as registering a student for a course
- That an object A can only communicate with another object B if A has a handle on B and the various ways that such a handle/reference can be obtained

- How classes designate the public/private accessibility of their features (attributes, methods) through a mechanism known as ***information hiding***
- How powerful a language feature information hiding is, both in terms of protecting the integrity of an object's data and preventing ripple effects in client code when private implementation details of an application inevitably change
- How to declare and use accessor ("get"/"set") methods to gracefully access the private attributes of an object from client code
- How a special type of function called a ***constructor*** is specified and used to control what is to occur when we instantiate new objects
- How ***overloading*** enables a class to have multiple methods with the same name and/or multiple constructors as long as their argument signatures are different

EXERCISES

1. Given a class Book defined as having the following attributes

```
Author author;
String title;
int noOfPages;
boolean fiction;
```

   write standard "get"/"set" method headers for each of these attributes.

2. [*Coding*] Actually code and compile the Book class based on the attributes and "get"/"set" methods called for in Exercise 1.
3. It's often possible to discern something about a class's design based on the messages that are getting passed to objects in client code. Consider the following client code snippet:

```
Student s;
Professor p;
boolean b;
String x = "Math";
```

```
s.setMajor(x);
if (!s.hasAdvisor()) {
  b = s.designateAdvisor(p);
}
```

What features—attributes, methods—are implied for the `Student` and `Professor` classes by virtue of how this client code is structured? Be as specific as possible with respect to

- The accessibility of each feature
- How each feature would be declared (e.g., the details, to the extent that you can "discover" them, of each method header)

4. [*Coding*] Expand the `Student` and `Professor` classes that you developed for Exercise 2 of Chapter 3 (and, optionally, the `Department` class that you developed for Exercise 3 of Chapter 3) as follows:
   - Include accessor methods for every attribute.
   - Reflect the appropriate accessibility on all features.
   - Include one or more constructors per class.
   - Write a method with the header `public void printAllAttributes()` that can be used to display the values of all attributes to the command prompt, for example:

```
Student Name: John Smith
Student ID: 123-45-6789
```

etc.

Then, modify the accompanying `MainClass`'s `main` method to take advantage of your new constructors to instantiate one of each of the object types.

5. What's wrong with the following code? Point out things that go against OO convention or best practices based on what you've learned in this chapter, regardless of whether or not the Java compiler would "complain" about them.

```
public class Building {
  private String address;
  public int numberOfFloors;

  void GetnumberOfFloors() {
    return numberOfFloors;
  }

  private void SetNoOfFloors(float n) {
    NumberOfFloors = n;
  }

  public void display() {
    System.out.println("Address:  " + address);
    System.out.println("No. of Floors:  " + numberOfFloors);
  }
```

CHAPTER 5

Relationships Between Objects

You learned in Chapter 4 that any two objects can have a "fleeting" relationship based on the fact that they exchange messages, in the same way that two strangers passing on the street might say "Hello!" to one another. We informally call such relationships between objects **behavioral relationships**, because they arise out of the behaviors, or actions, taken by one object X relative to another object Y.

With behavioral relationships, object X either is temporarily handed a reference to object Y as an argument in a method call or temporarily requests a handle on Y from another object Z. However, the emphasis is on ***temporary***: when X is finished communicating with Y, object X often discards the reference to Y.

In the same way that you have significant and more lasting relationships with some people (family members, friends, colleagues), there is also the notion of a more permanent relationship between objects. We informally refer to such relationships as **structural relationships** because, in order to keep track of such relationships, an object actually maintains long-term references to its related objects in the form of attributes, a technique that we discussed in Chapter 3.

In this chapter, you'll learn

- The various kinds of structural relationships that can be defined between classes, which in turn govern how individual objects may be linked together at run time
- How a powerful OOPL mechanism called **inheritance** enables us to derive new classes by describing only how they differ from existing classes

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_5

- The rules for what we can and can't do when deriving classes through inheritance
- How we must refine our understanding of (a) constructors and (b) accessibility of features when inheritance is at work

Associations and Links

The formal name for a structural relationship that exists between classes is an **association**. With respect to the Student Registration System, some sample associations might be as follows:

- A `Student` ***is enrolled in*** a `Course`.
- A `Professor` ***teaches*** a `Course`.
- A `DegreeProgram` ***requires*** a `Course`.

Whereas an association refers to a relationship between ***classes***, the term **link** is used to refer to a structural relationship that exists between two specific ***objects*** (***instances***). Given the association "a `Student` ***is enrolled in*** a `Course`," we might have the following links:

- Chloe Shylow (a particular `Student` object) is enrolled in Math 101 (a particular `Course` object).
- Fred Schnurd (a particular `Student` object) is enrolled in Basketweaving 972 (a particular `Course` object).
- Mary Smith (a particular `Student` object) is enrolled in Basketweaving 972 (a particular `Course` object—as it turns out, the ***same*** `Course` object that Fred Schnurd is linked to).

In the same way that an object is a specific instance of a class with its attribute values filled in, a link may be conceptually thought of as a specific instance of an association with its participating objects filled in, as illustrated in Figure 5-1.

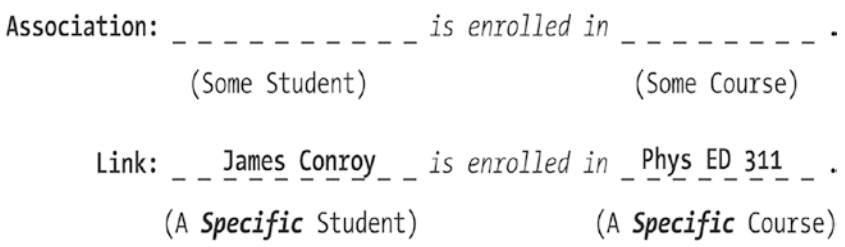


Figure 5-1. *An association is a template for creating links*

Yet another way to think of the difference between an association and a link is that

- An association is a ***potential*** relationship between objects of a certain type/class.
- A link is an ***actual*** relationship between objects of those particular types.

For example, given any `Student` object X and any `Course` object Y, there is the ***potential*** for a link of type *is enrolled in* to exist between those two objects ***precisely because*** there is an *is enrolled in* association defined between the two classes that those objects belong to. In other words, ***associations enable links***.

Most associations arise between two ***different*** classes; such associations are known as **binary associations.** The *is enrolled in* association, for example, is a binary association, because it interrelates two different classes—`Student` and `Course`. A **unary**, or **reflexive, association,** on the other hand, is between two instances of the ***same class***, for example:

- A `Course` ***is a prerequisite for*** (another) `Course`(s).
- A `Professor` ***supervises*** (other) `Professor`(s).

Even though the two classes specified at either end of a reflexive association are the same, the ***objects*** are typically different instances of that class:

- Math 101 (a `Course` object) is a prerequisite for Math 202 (a ***different*** `Course` object).
- Professor Smith (a `Professor` object) supervises Professors Jones and Green (other `Professor` objects).

Although somewhat rare, there can be situations in which the ***same*** object can serve in both roles of a reflexive relationship. For example, with regard to the association "a Professor ***is the chairman who represents*** other Professors," the actual Professor who is the chair of a given department would be their own representative.

Higher-order associations are also possible. A **ternary association** involves three classes—for example, "a `Student` takes a `Course` from a particular `Professor`," as illustrated in Figure 5-2.

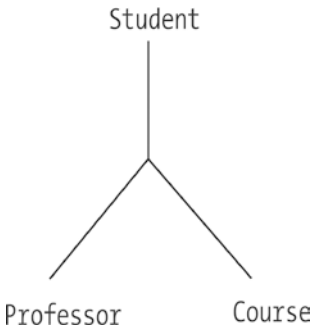


Figure 5-2. *A ternary association*

We typically decompose higher-order associations into the appropriate number of binary associations. We can, for example, represent the preceding three-way association as three binary associations instead (see Figure 5-3):

- A `Student` ***attends*** a `Course`.
- A `Professor` ***teaches*** a `Course`.
- A `Professor` ***instructs*** a `Student`.

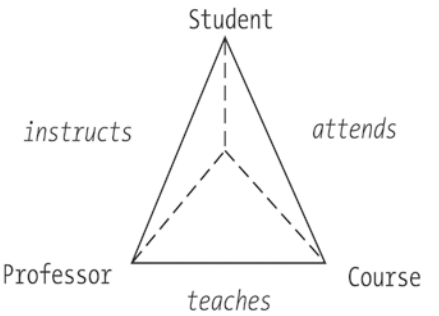


Figure 5-3. *An equivalent representation using three binary associations*

Within a given association, each participant class is said to have a **role**. In the ***advises*** association (a `Professor` ***advises*** a `Student`), the role of the `Professor` might be said to be "advisor," and the role of the `Student` might be said to be "advisee."

We only bother to assign names to the roles for the objects participating in an association if it helps clarify the abstraction. In the "is enrolled in" association (a `Student` ***is enrolled in*** a `Course`), there is no need to invent role names for the `Student` and `Course` ends of the association, because such role names wouldn't add significantly to the clarity of the abstraction.

Multiplicity

For a given association type X between classes A and B, the term **multiplicity** refers to the number of objects of type A that may be associated with a given instance of type B. For example, a `Student` attends ***multiple*** `Courses`, but a `Student` has only ***one*** `Professor` in the role of advisor.

There are three basic "flavors" of multiplicity: **one-to-one**, **one-to-many**, and **many-to-many**.

One-to-One (1:1)

With a one-to-one (1:1) association, exactly one instance of class A is related to exactly one instance of class B—no fewer, no more, and vice versa. For example:

- A `Student` has exactly one `Transcript`, and a `Transcript` belongs to exactly one `Student`.
- A `Professor` chairs exactly one `Department`, and a `Department` has exactly one `Professor` in the role of chairperson.

We can further constrain an association by stating whether the participation of the class at either end is optional or mandatory. For example, we can change the preceding association to read as follows:

- A `Professor` ***optionally*** chairs exactly one `Department`, but it is ***mandatory*** that a `Department` has exactly one `Professor` in the role of chairperson.

This revised version of the association is a more realistic portrayal of real-world circumstances than the previous version. While every department in a university typically does indeed have a chairperson, not every professor is a chairperson of a department—there aren't enough departments to go around! However, it's true that, ***if*** a professor happens to be a chairperson of a department, then that professor is the chairperson of only ***one*** department.

One-to-Many (1:m)

In a one-to-many (1:m) association, there can be ***many*** instances of class B related to a ***single*** instance of class A in a particular fashion; but, from the perspective of an instance of class B, there can only be ***one*** instance of class A that is so related. For example:

- A `Department` employs ***many*** `Professors`, but a `Professor` works for ***exactly one*** `Department`.
- A `Professor` advises ***many*** `Students`, but a given `Student` has ***exactly one*** `Professor` as an advisor.

Note that "many" in this case can be interpreted as either "zero or more (optional)" or as "one or more (mandatory)." To be a bit more specific, we can refine the previous one-to-many associations as follows:

- A `Department` employs ***one or more*** ("many," ***mandatory***) `Professors`, but a `Professor` works for exactly one `Department`.
- A `Professor` advises ***zero or more*** ("many," ***optional***) `Students`, but a given `Student` has exactly one `Professor` as an advisor.

In addition, as with one-to-one relationships, the "one" end of a one-to-many association may also be designated as mandatory or as optional. If we're modeling a university setting in which students aren't required to select an advisor, for example, we'd refine the previous association as follows:

- A `Professor` advises ***zero or more*** ("many," ***optional***) `Students`, but a given `Student` may ***optionally*** have ***at most one*** (i.e., ***zero or one***) `Professor` as an advisor.

Many-to-Many (m:m)

With a many-to-many (m:m) association, a given single instance of class A can have many instances of class B related to it and vice versa. For example:

- A `Student` enrolls in many `Courses`, and a `Course` has many `Students` enrolled in it.
- A given `Course` can have many prerequisite `Courses`, and a given `Course` can in turn ***be*** a prerequisite for many ***other*** `Courses`. (This is an example of a many-to-many ***reflexive*** association.)

As with one-to-many associations, "many" can be interpreted as ***zero*** or more (***optional***) or as ***one*** or more (***mandatory***) at either end of an (m:m) association, for example:

- A `Student` enrolls in ***zero or more*** ("many," ***optional***) `Courses`, and a `Course` has ***one or more*** ("many," ***mandatory***) `Students` enrolled in it.

Of course, the validity of a particular association—the classes that are involved, its multiplicity, and the optional or mandatory nature of participation in the association on the part of both participating classes—is wholly dependent on the real-world circumstances being modeled. If you were modeling a university in which departments could have more than one chairperson or where students could have more than one advisor, your choice of multiplicities would differ from those used in the preceding examples.

Multiplicity and Links

Note that the concept of multiplicity pertains to associations, but not to links. ***Links always exist in pairwise fashion between two objects*** (or, as mentioned earlier, in rare cases between an object and itself). Therefore, multiplicity in essence defines how many links of a certain association type can originate from a given object. This is best illustrated with an example.

Consider once again the many-to-many "is enrolled in" association:

- A `Student` enrolls in zero or more `Courses`, and a `Course` has one or more `Students` enrolled in it.

A ***specific*** `Student` object can have zero, one, or more links to `Course` objects, but any ***one*** of those links is between exactly ***two*** objects: a single `Student` object and a single `Course` object. In Figure 5-4, for example

- `Student` X has one link (to `Course` A).
- `Student` Y has four links (to `Courses` A, B, C, and D).
- `Student` Z has no links to any `Course` objects whatsoever. (Z is taking the semester off!)

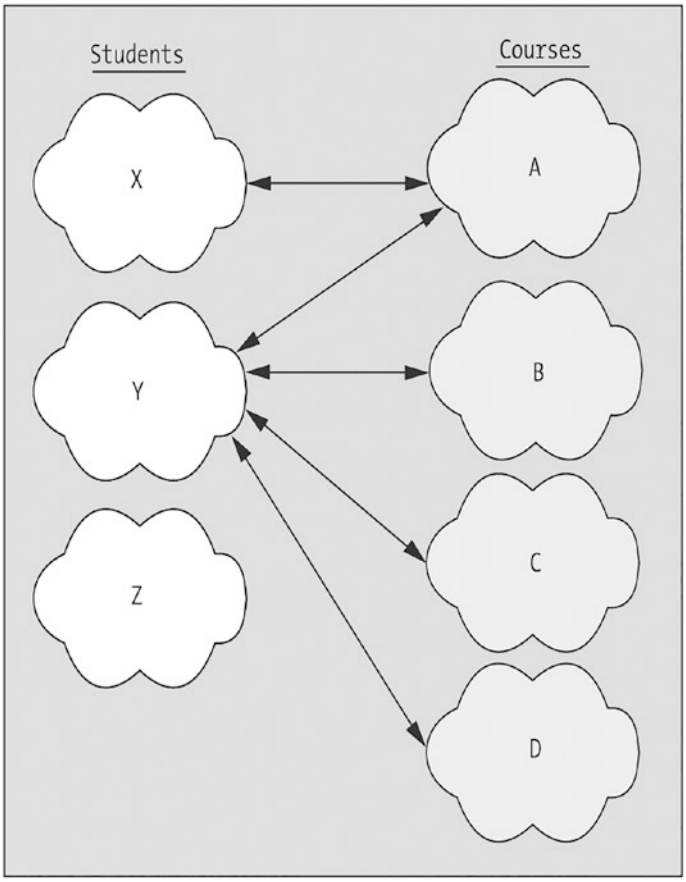


Figure 5-4. *Illustrating a many-to-many association between classes with pairwise links between objects*

Conversely, a ***specific*** `Course` object must have one or more links to `Student` objects to satisfy the mandatory nature and multiplicity of the "is enrolled in" association, but again, any ***one*** of those links is between exactly ***two*** objects: a single `Course` object and a single `Student` object. In Figure 5-4, for example

- `Course` A has two links (to `Students` X and Y).
- `Courses` B, C, and D each have one link (to the same `Student`, Y).

This example scenario does indeed uphold the many-to-many "is enrolled in" association between the `Student` and `Course` classes; it's but one of an infinite number of possible scenarios that may exist between the classes in question.

Just to make sure that this concept is clear, let's look at another example, this time using the one-to-one association:

- A `Professor` ***optionally*** chairs exactly one `Department`, and it is ***mandatory*** that a `Department` has exactly one `Professor` in the role of chairperson.

In Figure 5-5, we see that

- `Professor` objects 1 and 4 each have one link, to `Department` objects A and B, respectively.
- `Professor` objects 2 and 3 have no such links.

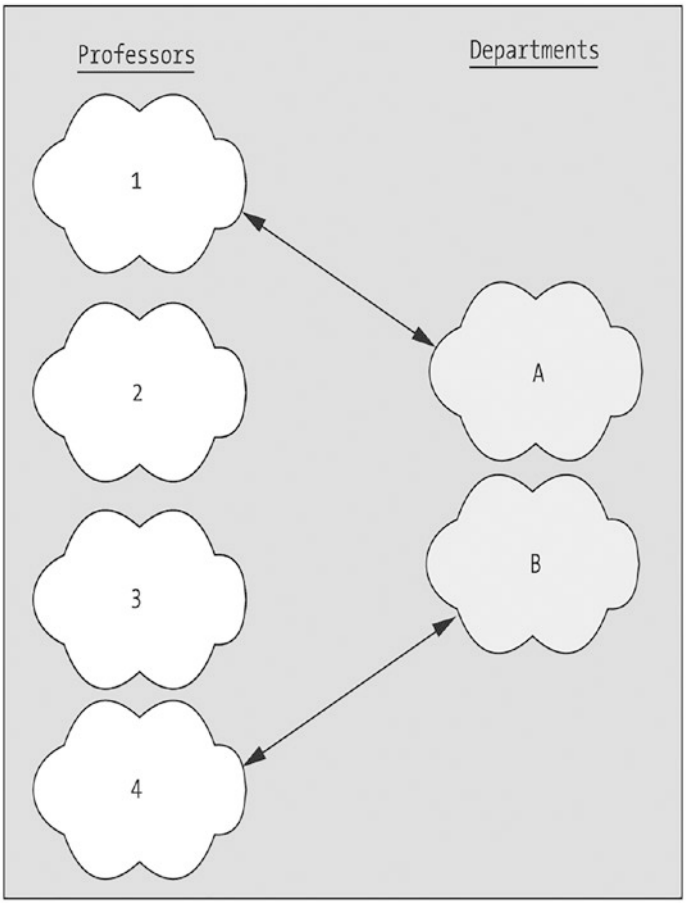


Figure 5-5. *Illustrating a one-to-one association between classes with* ***binary*** *links between objects*

Moreover, from the `Department` objects' perspective, each `Department` does indeed have exactly one link to a `Professor`. Therefore, this example upholds the one-to-one "chairs" association between `Professor` and `Department` while further illustrating the optional nature of the `Professor` class's participation in such links. Again, it's but one of an infinite number of possible scenarios that may exist between the classes in question.

Aggregation and Composition

Aggregation is a special form of association, alternatively referred to as the "consists of," "is composed of," or "has a" relationship. Like an association, an aggregation is used to represent a relationship between two classes, A and B. But, with an aggregation, we're

representing more than mere relationship: we're stating that an object belonging to class A, known as an **aggregate**, is composed of, or contains, **component objects** belonging to class B.

For example, a car is composed of an engine, a transmission, four wheels, etc., so if `Car`, `Engine`, `Transmission`, and `Wheel` were all classes, then we could form the following aggregations:

- A `Car` ***contains*** an `Engine`.
- A `Car` ***contains*** a `Transmission`.
- A `Car` ***contains*** many (in this case, four) `Wheels`.

Or, as an example related to the SRS, we can say that

- A `University` ***is composed of*** many `Schools` (the School of Engineering, the School of Law, etc.).
- A `School` ***is composed of*** many `Departments`.
- And so forth.

We wouldn't typically say, however, that a `Department` ***is composed of*** many `Professors`; instead, we'd probably state that a `Department` ***employs*** many `Professors`.

Note that these aggregation statements appear very similar to associations, where the name of the association just so happens to be "is composed of" or "contains." That's because an aggregation ***is*** an association in the broad sense of the term.

Why the fuss over trying to differentiate between aggregation and association? If an aggregation is really an association, must we even ***acknowledge*** aggregation as a distinct type of relationship between classes? Strictly speaking, the answer is no.

- There are indeed distinct representations in UML for the notions of aggregation vs. association, which we'll discuss in Chapter 10.
- However, as it turns out, both of these abstractions are ultimately rendered in code in precisely the same way.

Thus, it can be argued that it isn't really absolutely necessary to differentiate the notion of aggregation from association. Nonetheless, it behooves anyone who aspires to become proficient with object modeling and UML to be aware of this subtle distinction, if for no other reason than to be able to communicate effectively with other UML practitioners who are using such notation.

Composition is a strong form of aggregation, in which the "parts" cannot exist without the "whole." As an example, given the relationship "a `Book` is composed of many `Chapters`," we could argue that a chapter cannot exist if the book to which it belongs ceases to exist; whereas given the relationship "a `Car` ***is composed of*** many `Wheels`," we know that a wheel can be removed from a car and still serve a useful purpose. Thus, we'd categorize the `Book-Chapter` relationship as ***composition*** and the `Car-Wheel` relationship as ***aggregation***.

Inheritance

Whereas many of the OO techniques that you've learned for achieving a high degree of code flexibility and maintainability—for example, encapsulation and information hiding—are arguably achievable with non-OO languages in some form or another, the **inheritance** mechanism is what truly sets OO languages apart from their non-OO counterparts.

Before we dive into an in-depth discussion of how inheritance works, let's establish a compelling case for inheritance by looking at the problems that arise in its absence.

Responding to Shifting Requirements with a New Abstraction

Let's assume that we've accurately and thoroughly modeled all of the essential features of students via our `Student` class and that we've programmed the class in Java. A simplified version of the `Student` class is as follows:

```
public class Student {
  private String name;
  private String studentId;
  // etc.

  public String getName() {
    return name;
  }

  public void setName(String name) {
      this.name = n;
  }
```

```
  public String getStudentId() {
    return studentId;
  }

  public void setStudentId (String studentId) {
      this.studentId = studentId;
  }

  // etc.
}
```

Let's further assume that our `Student` class has been rigorously tested, has been found to be bug-free, and is actually being used in a number of applications: our Student Registration System, for example, as well as perhaps a Student Billing System and an Alumni Relations System for the same university.

A new requirement has just arisen for modeling graduate students as a special type of student. As it turns out, the only information about a graduate student that we need to track above and beyond that which we've already modeled for a "generic" student is

- What undergraduate degree the student previously received before entering their graduate program of study
- What institution the student received the undergraduate degree from

All of the other features necessary to describe a graduate student—the attributes `name`, `studentId`, and so forth, along with the corresponding accessor methods—are the same as those that we've already programmed for the `Student` class, because a graduate student ***is*** a student, after all.

How might we approach this new requirement for a `GraduateStudent` class? If we weren't well versed in object-oriented concepts, we might try one of the following approaches.

(Inappropriate) Approach #1: Modify the Student Class

We could add attributes to our existing `Student` class to reflect undergraduate degree information, along with "get"/"set" methods for these new attributes, as follows:

```
public class Student {
  private String name;
  private String studentId;
```

```
  // We've added two attributes to Student to handle the new
     requirements for
  // graduate students.
  private String undergraduateDegree;
  private String undergraduateInstitution;
  // etc.

  // We've also added four accessor methods.
  public String getName(...
  public void setName(...
  public String getStudentId(...
  public void setStudentId(...
  public String getUndergraduateDegree(...
  public void setUndergraduateDegree(...
  public String getUndergraduateInstitution(...
  public void setUndergraduateInstitution(...

  // etc.
}
```

Because these new features are not relevant to ***all*** students—only to graduate students—we'd perhaps simply allow these attributes to remain uninitialized for students who haven't yet received an undergraduate degree. However, to keep track of whether or not these attributes are supposed to contain values for a given `Student` object, we'd probably also want to add a `boolean` attribute to serve as a flag, along with accessor methods for this attribute:

```
public class Student {
  private String name;
  private String studentId;
  private String undergraduateDegree;
  private String undergraduateInstitution;

  // We'll set this next attribute to true if this is a
  // graduate student, false otherwise.
  private boolean graduateStudent;

  // etc.
```

```
  public String getName(...
  public void setName(...
  public String getStudentId(...
  public void setStudentId(...
  public String getUndergraduateDegree(...
  public void setUndergraduateDegree(...
  public String getUndergraduateInstitution(...
  public void setUndergraduateInstitution(...
  public boolean isGraduateStudent(...
  public void setGraduateStudent(...

  // etc.
}
```

Finally, in any methods that we've written for this class—or those that we write for this class in the future—we'd have to take the value of this `boolean` attribute into account:

```
  public void display() {
    System.out.println(getName());
    System.out.println(getStudentId());

    // If a particular student is NOT a graduate student, then the values
    // of the attributes "undergraduateDegree" and
       "undergraduateInstitution"
    // would be undefined/irrelevant, and so we only want to print them
    // if we are dealing with a GRADUATE student.
    if (this.isGraduateStudent()) {
       System.out.println(getUndergraduateDegree());
       System.out.println(getUndergraduateInstitution());
    }

    // etc.
  }
```

Having to sort out whether or not a given student is a graduate student in each and every `Student` method (the `display` method being but one) results in convoluted code that is difficult to debug and maintain. Where this ***really*** gets messy, however, is if we have to add a third or a fourth or a fifth type of "specialized" `Student` to the mix. For example,

consider how complicated the `display` method would become if we wanted to use it to represent a third type of student: namely, continuing education students, who don't seek a degree, but rather are just taking courses for continuing professional enrichment.

- Perhaps for such students, we'd like to track their current place of employment as an attribute.
- We'd most likely need to add yet another `boolean` flag as an attribute, as well, to keep track of whether or not a particular `Student` is a continuing education student.

We'd perhaps extend our `Student` class once more, as highlighted in **bold** in the following code, to reflect the newly added attributes and accessor methods:

```
public class Student {
  private String name;
  private String studentId;
  private String undergraduateDegree;
  private String undergraduateInstitution;
  private String placeOfEmployment;
  private boolean graduateStudent;
  private boolean continuingEdStudent;
  // etc.

  public String getName(...
  public void setName(...
  public String getStudentId(...
  public void setStudentId(...
  public String getUndergraduateDegree(...
  public void setUndergraduateDegree(...
  public String getUndergraduateInstitution(...
  public void setUndergraduateInstitution(...
  public boolean isGraduateStudent(...
  public void setGraduateStudent(...
  public boolean isContinuingEdStudent(...
  public void setContinuingEdStudent(...

  // etc.
}
```

We also now must take the value of the `boolean isContinuingEdStudent` attribute into account in all of the `Student` methods involving the notion of `placeOfEmployment`. Take a look at how this impacts the logic of the `display` method:

```
public void display() {
  System.out.println(getName());
  System.out.println(getStudentId());
  // etc.

  if (this.isGraduateStudent()) {
    System.out.println(getUndergraduateDegree());
    System.out.println(getUndergraduateInstitution());
  }

  if (this.isContinuingEdStudent()) {
    System.out.println(getPlaceOfEmployment());
  }

  // etc.
}
```

Now, imagine how much ***more*** "spaghetti-like" our code might become if we had ***dozens*** of different student types to accommodate. ***Approach #1 is clearly not the answer***! The underlying flaw with this approach is that we're trying too hard to force a ***single*** abstraction, `Student`, to represent ***multiple*** real-world object types. While graduate students, continuing education students, and "generic" students certainly have some features in common, they are nonetheless ***different*** types of object.

(Inappropriate) Approach #2: "Clone" the Student Class to Create a GraduateStudent Class

We could instead create a new `GraduateStudent` class by ***copying*** the code of `Student.java` to create `GraduateStudent.java`, renaming the latter class `GraduateStudent`, and then adding the extra features required of a graduate student to the ***copy.***

Here's the resultant `GraduateStudent` class:

```
// GraduateStudent.java

public class GraduateStudent {
    // Student attributes DUPLICATED!
    private String name;
    private String birthDate;
    // etc.

    // Add the two new attributes required of a GraduateStudent.
    private String undergraduateDegree;
    private String undergraduateInstitution;

    // Student methods DUPLICATED!
    public String getName(...
    public void setName(...
    public String getBirthDate(...
    public void setBirthDate(...
    // etc.

    // Add the new accessor methods required of a GraduateStudent.
    public String getUndergraduateDegree(...
    public void setUndergraduateDegree(...
    public String getUndergraduateInstitution(...
    public void setUndergraduateInstitution(...
}
```

This would be a very poor design, since we'd have much of the same code in two places: `Student.java` and `GraduateStudent.java`. If we wanted to change how a particular method worked or how an attribute was defined later on—say, a change of the type of the `birthDate` attribute from `String` to `Date`, with a corresponding change to the accessor methods for that attribute—then we'd have to make the same changes in ***both*** classes. Again, this problem quickly gets compounded if we've defined three or four or a ***dozen*** different types of `Student`, all created as "clones" of the original `Student` class; the code maintenance burden would quickly become excessive. ***Approach #2 is clearly not the answer, either***!

Strictly speaking, either of the preceding two approaches would work, but the inherent redundancy/complexity of the resultant code would make the application prohibitively difficult to maintain. Unfortunately, with non-OO languages, such convoluted approaches would typically be our ***only*** options for handling the requirement for a new type of object. It's no wonder that applications become so complicated and expensive to maintain as requirements inevitably evolve over time. Fortunately, we do have yet another very powerful approach that we can take specific to OO programming languages: we can take advantage of the mechanism of **inheritance**.

The Proper Approach (#3): Taking Advantage of Inheritance

With an object-oriented programming language, we can solve the problem of specializing the `Student` class by harnessing the power of **inheritance**, a mechanism for defining a new class by stating only the differences (in terms of features) between the new class and another class that we've already established.

Using inheritance, we can declare a new class named `GraduateStudent` that inherits all of the features of the `Student` class "as is." The `GraduateStudent` class would then only have to specify the two extra attributes associated with a graduate student—`undergraduateDegree` and `undergraduateInstitution`—plus their accessor methods, as shown in the following `GraduateStudent` class. Note that inheritance is triggered in a Java class declaration using the `extends` keyword: `public class` *`NewClass`* `extends` *`ExistingClass`* `{ ....`

```
public class GraduateStudent extends Student {
  // Declare two new attributes above and beyond
  // what the Student class has already declared ...

  private String undergraduateDegree;
  private String undergraduateInstitution;

  // ... and accessor methods for each of these new attributes.

  public String getUndergraduateDegree {
      return undergraduateDegree;
  }
```

```
  public void setUndergraduateDegree(String s) {
      undergraduateDegree = s;
  }

  public String getUndergraduateInstitution {
      return undergraduateInstitution;
  }

  public void setUndergraduateInstitution(String s) {
      undergraduateInstitution = s;
  }

  // That's the ENTIRE GraduateStudent class declaration!
  // Short and sweet!
}
```

That's all we need to declare in establishing our new `GraduateStudent` class: two attributes plus the associated four accessor methods. There is no need to duplicate any of the features of the `Student` class within the code of `GraduateStudent`, because we're automatically inheriting these. It's as if we had "plagiarized" the code for the attributes and methods of the `Student` class, copying this code from `Student` and pasting it into `GraduateStudent`, but without the fuss of actually having done so. The `GraduateStudent` class thus has ***n*** + 6 features: the six features that are explicitly declared within the `GraduateStudent.java` file plus ***n*** more that are inherited from `Student`.

When we take advantage of inheritance, the original class that we're starting from—`Student`, in this case—is called the (**direct**) **superclass.** The new class—`GraduateStudent`—is called a (**direct**) **subclass.** A subclass is said to **extend** its direct superclass.

The "is a" Nature of Inheritance

Inheritance is often referred to as the "is a" relationship between two classes, because if a class B (`GraduateStudent`) is derived from a class A (`Student`), then B truly ***is a*** special case of A. Anything that we can say about a superclass must therefore also be true about all of its subclasses; that is

- A `Student` attends classes, and so a `GraduateStudent` attends classes.
- A `Student` has an advisor, and so a `GraduateStudent` has an advisor.
- A `Student` pursues a degree, and so a `GraduateStudent` pursues a degree.

In fact, an "acid test" for legitimate use of inheritance is as follows: ***if there is something that can be said about a class A that can't be said about a proposed subclass B, then B isn't a valid subclass of A***.

Because subclasses are special cases of their superclasses, the term **specialization** is used to refer to the process of deriving one class from another. **Generalization**, on the other hand, is a term used to refer to the opposite process: namely, recognizing the common features of several existing classes and creating a new, common superclass for them all.

Let's say we now wish to declare a `Professor` class to complement our `Student` class. `Students` and `Professors` have some features in common: attributes `name`, `birthDate`, etc. and the methods that manipulate these attributes. Yet, they each have unique features, as well:

- The `Professor` class might require the attributes `title` (a `String`) and `worksFor` (a reference to a `Department`).
- Conversely, the `Student` class's `studentID`, `degreeSought`, and `majorField` attributes are irrelevant for a `Professor`.

Because each class has attributes that the other would find useless, neither class can be derived from the other. Nonetheless, to ***duplicate*** their common attribute declarations and method code in two places would be very inefficient. In such a circumstance, we'd want to invent a new ***superclass*** called `Person`, consolidate the features common to both `Students` and `Professors` in the `Person` class, and then have `Student` and `Professor` inherit these common features by `extending Person`. The resultant code in this situation follows.

First, we'll define the `Person` superclass in a file named `Person.java`:

```
// Person.java

public class Person {
  // Attributes common to Students and Professors.
  private String name;
```

```
  private String address;
  private String birthDate;

  // Common accessor methods.

  public String getName() {
      return name;
  }

  public void setName(String name) {
     this.name = name;
  }

  // etc. for the other two attributes

  // Other general-purpose Person methods, if any, would go here - details
     omitted.
}
```

Next, we'll streamline our Student class as previously presented to remove those features that it will now inherit from Person:

```
// Student.java

public class Student extends Person {
  // Attributes specific only to a Student; redundant attributes -
     i.e., those
  // that are shared with Professor, and hence are now declared by
     Person - have
  // been REMOVED from Student.
  private String studentId;
  private String majorField;
  private String degreeSought;

  // Student-specific accessor methods - redundant methods have been
     removed.

  public String getStudentId() {
      return studentId;
  }
```

```
  public void setStudentId(String studentId) {
      this.studentId = studentId;
  }

  // etc. for the other two explicitly declared Student attributes.

  // Other Student-specific methods go here, if any; details omitted.
}
```

Finally, we'll define the second new (sub)class, `Professor`. This class would go into a separate `Professor.java` file:

```
// Professor.java

public class Professor extends Person {
  // Attributes specific only to a Professor; redundant attributes -
     i.e., those
  // that are shared with Student, and hence are now declared by Person - are
  // not included here.
  private String title;
  private Department worksFor;

  // Professor-specific accessor methods go here.

  public String getTitle() {
      return title;
  }

  public void setTitle(String title) {
      this.title = title;
  }

 public Department getWorksFor() {
    return worksFor;
}

public void setWorksFor(Department worksFor) {
    this.worksFor = worksFor;
}

  // Other Professor-specific methods go here, if any; details omitted.
}
```

By generalizing the shared features of `Students` and `Professors` into a common superclass called `Person`, we'll easily be able to introduce a third type of `Person` or a fourth or a fifth, if needed in the future, and they'll all ***share in these same features*** through the mechanism of inheritance. Furthermore, if we wish to introduce new subtypes of these subclasses—perhaps `AdjunctProfessor` and `TenuredProfessor` as subclasses of the `Professor` class—they'll all derive a common set of features as a result of their shared `Person` "ancestry."

The Benefits of Inheritance

Inheritance is perhaps one of the most powerful and unique aspects of an OO programming language for the following reasons:

- ***We dramatically reduce code redundancy***, thus lessening the burden of code maintenance when requirements change or logic flaws are detected.
- ***Subclasses are much more succinct than they would be without inheritance***. A subclass contains only the essence of what differentiates it from its direct superclass. We know from looking at the `GraduateStudent` class definition, for example, that a graduate student is "a student who already holds an undergraduate degree from an educational institution." As a result, ***the total body of code for a given OO application is significantly reduced*** as compared with the traditional/non-OO version of the same application.
- ***Through inheritance, we can reuse and extend code that has already been thoroughly tested without modifying it***. As you saw, we were able to invent a new class—`GraduateStudent`—without disturbing the `Student` class code in any way. We can therefore rest assured that any client code that relies on instantiating generic `Student` objects and passing messages to them will be unaffected by the creation of subclass `GraduateStudent`, and thus we avoid having to retest huge portions of our existing application(s). (Had we used a non-OO approach of "tinkering" with the `Student` class code to try to accommodate graduate student requirements, on the other hand, we would have had to retest our entire existing application to make sure that nothing had "broken"!)

- ***Best of all, we can derive a new class from an existing class even if we don't own the source code for the latter***! As long as we have the ***compiled bytecode version*** of a class, the inheritance mechanism works just fine; we don't need the original source code of a class in order to extend it. ***This is one of the most dramatic ways to achieve productivity with an object-oriented language***: find a class (either one written by someone else or one that is built into the language) that does much of what you need, and create a subclass of that class, adding just those features that you need for your own purposes.

We'll look at a specific example of extending one of the predefined Java collection classes in Chapter 6.

- Finally, as we discussed in Chapter 1, ***classification is the natural way that humans organize information***; so it only makes sense that we'd organize software along the same lines, making it much more intuitive and hence easier to develop, maintain, extend, and communicate with users about.

Class Hierarchies

Over time, we build up an inverted tree of classes that are interrelated through inheritance; such a tree is called a **class hierarchy**. One such class hierarchy example is shown in Figure 5-6. Note that arrows are used to point ***upward*** from each subclass to its direct superclass.

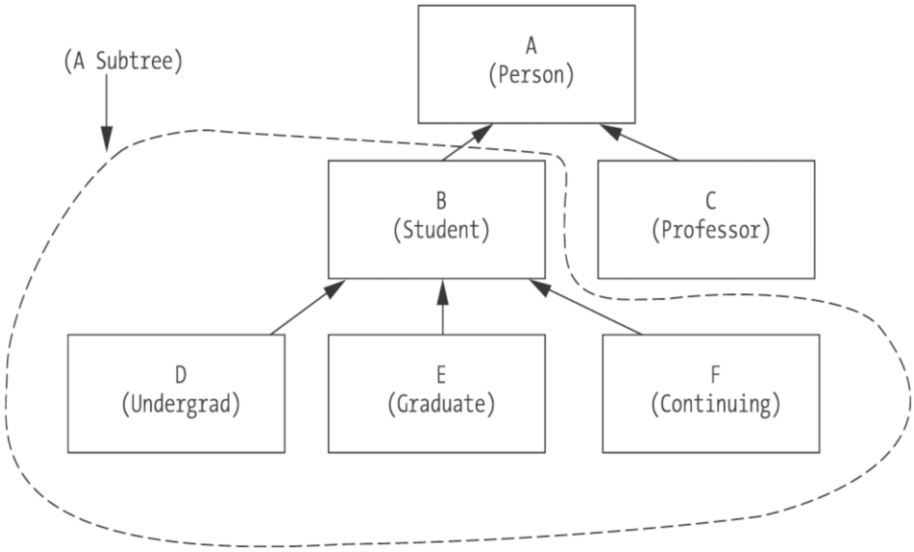


Figure 5-6. *A sample class hierarchy*

A bit of nomenclature follows:

- We may refer to each class as a **node** in the hierarchy.
- Any given node in the hierarchy is said to be (directly or indirectly) ***derived from*** all of the nodes above it in the hierarchy, known collectively as its **ancestors**.
- The ancestor that is ***immediately*** above a given node in the hierarchy is considered to be that node's direct superclass.
- Conversely, all nodes below a given node in the hierarchy are said to be its **descendants**.
- The node that sits at the top of the hierarchy is referred to as the **root node**.
- A **terminal**, or **leaf**, **node** is one that has no descendants.
- Two nodes that are derived from the same direct superclass are known as **siblings**.

Applying this terminology to the example hierarchy in Figure 5-6

- Class `A` (`Person`) is the root node of the entire hierarchy.
- Classes `B`, `C`, `D`, `E`, and `F` are all said to be derived from class `A` and thus are all descendants of `A`.

- Classes `D`, `E`, and `F` can be said to be derived from class `B`.
- Classes `D`, `E`, and `F` are siblings; so are classes `B` and `C`.
- Class `D` has two ancestors, `B` (its direct superclass) and `A`.
- Classes `C`, `D`, `E`, and `F` are terminal nodes in that they don't have any classes derived from them (as of yet, at any rate).

As with any hierarchy, this one may evolve over time:

- It may ***widen*** with the addition of new siblings/branches in the tree.
- It may ***expand downward*** as a result of future specialization.
- It may ***expand upward*** as a result of future generalization.

Such changes to the hierarchy are made as new requirements emerge or as our understanding of the existing requirements improves. For example, we may determine the need for `MastersStudent` and `PhDStudent` classes as specializations of `GraduateStudent` or of an `Administrator` class as a sibling to `Student` and `Professor`. This would yield the revised hierarchy shown in Figure 5-7.

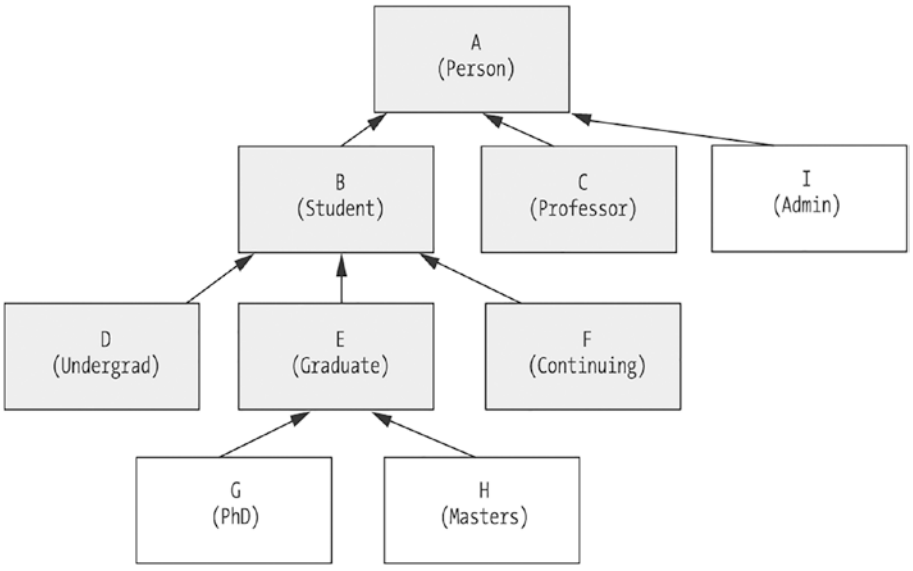


Figure 5-7. *Class hierarchies inevitably expand over time*

The Object Class

In the Java language, the built-in `Object` class serves as the ultimate superclass for all other reference types, both user-defined and those built into the language. Even when a class is not explicitly declared to extend `Object`, such extension is implied. That is, when we declare a class as follows

```
public class Person { ... }
```

it is as if we've written

```
public class Person extends Object { ... }
```

without having to explicitly do so. And, when we write

```
public class Student extends Person { ... }
```

then, because the `Person` class is derived from `Object`, `Student` is derived from `Object` as well. Thus, the ***true*** root of the hierarchy illustrated in Figure 5-7—and of ***all*** (Java) class hierarchies—is the `Object` class.

We'll talk in depth about the significance of the `Object` class, and the fact that all Java objects are ultimately descended from `Object`, in Chapter 13.

Is Inheritance Really a Relationship?

Association, aggregation, and inheritance are all said to be relationships between ***classes***. Where inheritance differs from association and aggregation is at the ***object*** level.

As you saw earlier in this chapter, association (and aggregation, as a special form of association) can be said to relate individual objects, in the sense that two different objects are linked to one another by virtue of the existence of an association between their respective classes. Inheritance, on the other hand, does ***not*** involve linking distinct objects; rather, inheritance is a way of describing the collective features of a ***single object***. With inheritance, an object is ***simultaneously*** an instance of a subclass and all of its superclasses: a `GraduateStudent` is a `Student` that is a `Person` that is an `Object`, all wrapped into one!

So, in looking once again at the hierarchy of Figure 5-7, we see that

- ***All*** classes in the hierarchy—class A (`Person`) as well as all of its descendants B through I—may be thought of as yielding `Person` objects.
- Class B (`Student`), along with its descendants D through H, may all be thought of as yielding `Student` objects.

This notion of an object having "multiple identities" is a significant one that we'll revisit several times throughout the book.

So, getting back to the question posed as the title of this section, inheritance is indeed a relationship between ***classes***, but ***not*** between distinct ***objects***.

Avoiding "Ripple Effects" in a Class Hierarchy

Once a class hierarchy is established and an application has been coded, changes to ***non***-leaf classes (i.e., those classes that have descendants) have the potential to introduce undesired ripple effects further down the hierarchy. For example, if after we've established the `GraduateStudent` class, we go back and add a `minorField` attribute to the `Student` class, then the `GraduateStudent` class will automatically inherit this new attribute. Perhaps this is what we want; on the other hand, we might not have anticipated the derivation of a `GraduateStudent` class when we first conceived of `Student`, and so this may ***not*** be what we want!

As the developers of the `Student` superclass, it would be ***ideal*** if we could speak with the developers of all derived classes—`GraduateStudent`, `MastersStudent`, and `PhDStudent`—to obtain their approval for any proposed changes to `Student`. But this is typically not practical; in fact, we often don't even ***know*** that our class has been extended if, for example, our code is being distributed and reused on other projects. This evokes a general rule of thumb:

Whenever possible, avoid adding features to non-leaf classes once they have been deployed in code form in an application, to avoid ripple effects throughout an inheritance hierarchy.

This is easier said than done! However, it reinforces the importance of spending as much time as possible on the requirements analysis and object modeling stages of an OO application development project before diving into the coding stage. This won't prevent new requirements from emerging over time, but we should at least do everything possible to avoid oversights regarding the ***current*** requirements.

Rules for Deriving Classes: The "Do's"

When deriving a new class, we can do several things to specialize the superclass that we are starting out with:

- We may ***extend*** the superclass by ***adding features***. In our `GraduateStudent` example, we added six features: two attributes—`undergraduateDegree` and `undergraduateInstitution`—and four accessor methods, `getUndergraduateDegree`/`setUndergraduateDegree` and `getUndergraduateInstitution`/`setUndergraduateInstitution`.
- We also may ***specialize*** the way that a subclass performs one or more of the ***services*** inherited from its superclass.

For example, when a "generic" student enrolls for a course, the business rules for the SRS may require us to ensure that

- The student has taken the necessary prerequisite courses.
- The course is required for the degree that the student is seeking.

When a ***graduate student*** enrolls for a course, on the other hand, the business rules may involve doing both of these things as well as ensuring that the student's graduate committee feels that the course is appropriate.

Specializing the way that a subclass performs a service—that is, how it responds to a given message as compared with the way that its superclass would have responded to the same message—is accomplished via a technique known as **overriding**.

Overriding

Overriding involves "rewiring" how a method works internally, without changing the client code interface to/signature of that method. For example, let's say that we've defined a `print` method for the `Student` class to print out the values of all of a `Student`'s attributes:

```
public class Student {
  // Attributes.
  private String name;
  private String studentId;
```

```
  private String majorField;
  private double gpa;
  // etc.

  // Accessor methods for each attribute would also be provided; details
     omitted.

  public void print() {
    // Print the values of all of the attributes that the Student class
    // knows about.  (Remember: "\n" is a newline.)
    System.out.println("Student Name:  " + getName() + "\n" +
                       "Student No.:  " + getStudentId() + "\n" +
                       "Major Field:  " + getMajorField() + "\n" +
                       "GPA:  " + getGpa());
  }
}
```

By virtue of inheritance, all of the subclasses of `Student` will inherit this method.

We go on to derive the `GraduateStudent` subclass from `Student`, adding two attributes to `GraduateStudent`—`undergraduateDegree` and `undergraduateInstitution`. If we take the "lazy" approach of just letting `GraduateStudent` inherit the `print` method of `Student` as is, then whenever we invoke the `print` method for a `GraduateStudent`, all that will be printed are the values of the four attributes inherited from `Student`—`name`, `studentId`, `major`, and `gpa`—because these are the only attributes that the `print` method has been explicitly programmed to print the values of. Ideally, we would like the `print` method, when invoked for a `GraduateStudent`, to print these same four attributes ***plus*** the two additional attributes of `undergraduateDegree` and `undergraduateInstitution`.

With an object-oriented language, we are able to ***override***, or supersede, the superclass's version of a method with a subclass-specific version. To override a superclass's method in Java rather than merely inheriting the method as is, the header of the method as declared in the superclass must be repeated in the subclass; we are then free to reprogram the ***body*** of that method in the subclass to specialize its behavior.

Let's look at how the `GraduateStudent` class would go about overriding the `print` method of the `Student` class. For your convenience, I've repeated the code of the `Student` class here:

```
public class Student {
  // Attributes.
  private String name;
  private String studentId;
  private String majorField;
  private double gpa;
  // etc.

  // Accessor methods for each attribute would also be provided; details
     omitted.

  public void print() {
    // Print the values of all the attributes that the Student class
    // knows about; again, note the use of accessor methods.
    System.out.println("Student Name:  " + getName() + "\n" +
                       "Student No.:  " + getStudentId() + "\n" +
                       "Major Field:  " + getMajorField() + "\n" +
                       "GPA:  " + getGpa());
  }
}

//-------------------------------------------

public class GraduateStudent extends Student {
  private String undergraduateDegree;
  private String undergraduateInstitution;

  // Accessor methods for each newly added attribute would also be
     provided;
  // details omitted.

  // We are overriding the Student class's print method; note that
  // we've repeated the print method header verbatim from the
  // Student class, which triggers overriding.
  public void print() {
    // We print the values of all of the attributes that the
    // GraduateStudent class knows about:  namely, those that it
    // inherited from Student plus those that it explicitly declares above.
```

```
    System.out.println("Student Name:  " + this.getName() + "\n" +
                       "Student No.:  " + this.getStudentId() + "\n" +
                       "Major Field:  " + this.getMajorField() + "\n" +
                       "GPA:  " + this.getGpa() + "\n" +
                       "Undergrad. Deg.:  " + this.
                       getUndergraduateDegree() +
                        "\n" + "Undergrad. Inst.:  " +
                        this.getUndergraduateInstitution());
  }
}
```

The `GraduateStudent` class's version of `print` thus overrides, or supersedes, the version that would otherwise have been inherited from the `Student` class.

In a complex inheritance hierarchy, we often have occasion to override a given method multiple times. In the hierarchy shown in Figure 5-8

- Root class A (`Person`) declares a method with the header `public void print()` that prints out all of the attributes declared for the `Person` class.
- Subclass B (`Student`) overrides this method, changing the internal logic of the method body to print not only the attributes inherited from `Person` but also those that were added by the `Student` class itself.
- Subclass E (`GraduateStudent`) overrides this method again, to print not only the attributes inherited from `Student` (which include those inherited from `Person`) but also those that were added by the `GraduateStudent` class itself.

Note that, in all cases, the method signature ***must*** remain the same—`print()`—for overriding to take place.

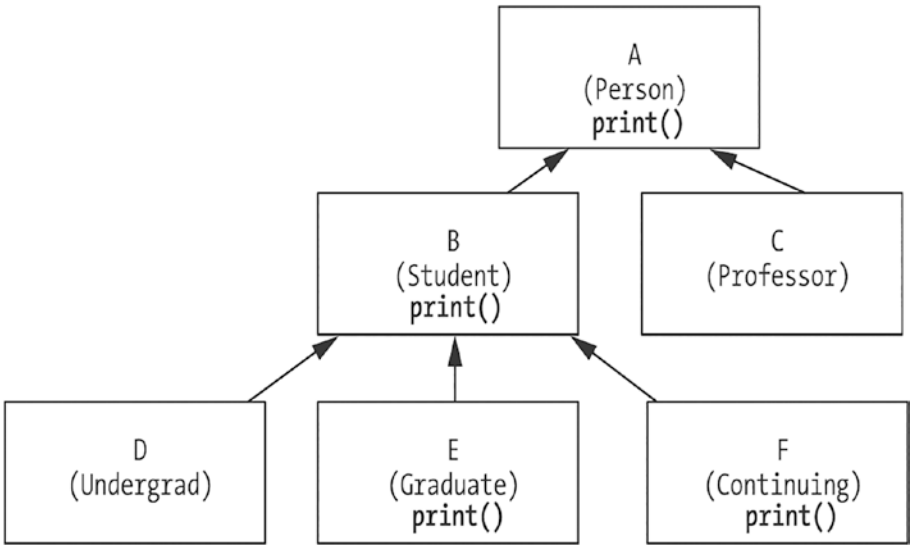


Figure 5-8. *A method may be overridden multiple times within a class hierarchy*

Under such circumstances, any class not specifically overriding a given method itself will inherit the definition of that method used by its most immediate ancestor. Thus,

- Classes C and D in Figure 5-8 inherit the versions of `print()` that are defined by A and B, respectively.
- B, E, and F are all overriding the `print()` method of their parent classes.

Reusing Superclass Behaviors: The "super" Keyword

The preceding example of overriding is less than ideal because the first four lines of the `print` method of `GraduateStudent` duplicated the code from the `Student` class's version of `print`. Here's the `Student` version of the method once again:

```
public void print() {
    // Print the values of all the attributes that the Student class
    // knows about; again, note the use of accessor methods.
    System.out.println("Student Name:  " + getName() + "\n" +
                       "Student No.:  " + getStudentId() + "\n" +
                       "Major Field:  " + getMajorField() + "\n" +
                       "GPA:  " + getGpa());
  }
```

And here's the GraduateStudent version:

```
public void print() {
  // This code is repeated from the Student version!
  System.out.println("Student Name:  " + getName() + "\n" +
                     "Student No.:  " + getStudentId() + "\n" +
                     "Major Field:  " + getMajorField() + "\n" +
                     "GPA:  " + getGpa() + "\n" +
                     "Undergrad. Deg.:  " + getUndergraduateDegree()
                     + "\n" +
                     "Undergrad. Inst.:  " +
                     getUndergraduateInstitution());
}
```

Redundancy in an application is to be avoided whenever possible, because redundant code represents a maintenance headache. When we have to change code in one place in an application, we don't want to have to remember to change it in countless other places or, worse yet, forget to do so and wind up with inconsistency in our logic.

Fortunately, Java provides a way for us to have our cake and eat it, too—that is, a way for us to ***override*** the print method while ***simultaneously reusing its code***. We'd code the print method for the GraduateStudent class as follows:

```
public class GraduateStudent extends Student {
  // Details omitted.

  public void print() {
    // Reuse code by calling the print method as defined by the Student
    // superclass ...
    super.print();

    // ... and then go on to do something extra - namely, print
    this derived
    // class's specific attributes.
    System.out.println("Undergrad. Deg.:  " + getUndergraduateDegree()
    + "\n" +
                       "Undergrad. Inst.:  " + getUndergraduateInstitution());
  }
}
```

We use the Java keyword `super` as the qualifier for a method call

```
super.methodName(arguments);
```

whenever we wish to invoke the version of method *methodName* that was defined by our superclass. That is, in the preceding example, we're essentially saying to the compiler, "First, execute the `print` method the way that the superclass, `Student`, would have executed it, and then do something extra—namely, print out the values of the new `GraduateStudent` attributes."

Note that the syntax

```
super.methodName(arguments);
```

involves ***invoking*** one method from within another. Let's look at a slightly more involved example to emphasize this syntax.

We'll start with this superclass declaration

```
public class Superclass {
    public void foo(int x, int y) { ... }
}
```

and derive this subclass from it:

```
public class Subclass extends Superclass {
    // We're overriding the foo method.
    // (Note that we're using a and b as parameter names here to override
    // parameters x and y in the superclass; this is perfectly fine as long
    // as their types are identical.)
    public void foo(int a, int b) {
        // Details to follow ...
    }
}
```

We have numerous options as to how we might use the `super` keyword within the overridden `foo` method of a subclass, as illustrated by the **bolded** passages (and corresponding comments) in the examples that follow:

```
public class Subclass extends Superclass {
    // We're overriding the foo method.
    public void foo(int a, int b) {
```

```
        // We can pass the argument values a and b through to our
        superclass's
        // version of foo ...
        super.foo(a, b);
    }
}
```

or

```
public class Subclass extends Superclass {
    // We're overriding the foo method.
    public void foo(int a, int b) {
        int x = 2;  // a local variable

        // We can pass selected argument values through to our superclass's
        // version of foo ...
        super.foo(a, x);
    }
}
```

or even

```
public class Subclass extends Superclass {
    // We're overriding the foo method.
    public void foo(int a, int b) {
        int x = 2;  // a local variable

        // Here, we're using neither a nor b as arguments.
        super.foo(x, 3);
    }
}
```

Note that our invocation of `super.foo(...)` can occur anywhere within the method:

```
public class Subclass extends Superclass {
    // We're overriding the foo method.
    public void foo(int a, int b) {
        // Pseudocode.
        do some stuff;

        super.foo(a, b);
```

```
        // Pseudocode.
        do more stuff;
    }
}
```

And, if `foo` were declared to have a non-`void` return type in `Superclass`—say, `int`—we could even return the result of calling `super.foo(...)`:

```
public class Subclass extends Superclass {
    // We're overriding the foo method (here, we
    // assume that foo was declared with an int return
    // type in the superclass).
    public int foo(int a, int b) {
        int x = 3 * a;
        int y = 17 * b;
        return super.foo(x, y);
    }
}
```

The bottom line is we can use `super.`*`methodName`*`(...)` in whatever way makes sense in carrying out a subclass's version of a method that is being overridden.

Another important use of the `super` keyword has to do with reusing constructor code; we'll learn about this alternative use of `super` later in this chapter.

Rules for Deriving Classes: The "Don'ts"

When deriving a new class, there are some things that we should ***not*** attempt to do. (And, as it turns out, OO language compilers will actually prevent us from successfully compiling programs that attempt to do most of these things.)

We shouldn't change the semantics—that is, the intention or meaning—of a feature. For example:

- If the `print` method of a superclass such as `Student` is intended to display the values of all of an object's attributes in the command window, then the `print` method of a subclass such as `GraduateStudent` shouldn't, for example, be overridden so that it directs all of its output to a file instead.

- If the `name` attribute of a superclass such as `Person` is intended to store a person's name in "last name, first name" order, then the `name` attribute of a subclass such as `Student` should be used in the same fashion.

We can't physically eliminate features, nor should we effectively eliminate them by ignoring them. To attempt to do so would break the spirit of the "is a" hierarchy. By definition, inheritance requires that ***all*** features of ***all*** ancestors of a class A must also apply to class A itself in order for A to truly be a proper subclass. If a `GraduateStudent` could eliminate the `degreeSought` attribute that it inherits from `Student`, for example, is a `GraduateStudent` ***really*** a `Student` after all? Strictly speaking, the answer is no.

Furthermore, from a practical standpoint, if we effectively disable a method by overriding it with a "do nothing" version, as illustrated in the following `BadStudent` example

```
public class Student {
    // Details omitted.
    public void printStudentInfo() {
        // Pseudocode.
        print all attribute values ...
    }
}

public class BadStudent extends Student {
    // Details omitted.
    // We're overriding the printStudentInfo method of Student by
    // "stubbing it out" - that is, by providing it with an EMPTY method
    // body, so that it effectively does NOTHING.
    // (Note that this WILL compile!)
    public void printStudentInfo() { }
}
```

someone else might wish to derive a subclass from ***our*** subclass later on

```
public class NaiveStudent extends BadStudent { ...
```

assuming that they'll inherit a ***meaningful*** version of `printStudentInfo` from our `BadStudent` superclass. This is a reasonable thing for them to assume, given the "all or nothing" nature of inheritance, especially if this other developer doesn't have access

to the ***source code*** of `BadStudent` to look at. Unfortunately, because we've broken the spirit of the "is a" relationship in the way that we've compromised the `printStudentInfo` method, we've burdened them—and anyone else who might choose to derive a class from `BadStudent`—with a "defective" method. ***The bottom line is never do this***!

Finally, ***we shouldn't attempt to change a method's signature when we override it***. For example, if the `print` method inherited by the `Student` class from `Person` has the signature `print()`, then the `Student` class can't change this method's header to accept an argument, say, `print(int noOfCopies)`. To do so is to create a different method entirely, due to another language feature known as **overloading**, a concept that we discussed in Chapter 4. That is, in the following example

```
public class Person {
    // Details omitted.
    public void print() { ... }
}

public class Student extends Person {
    // Details omitted.
    // We're naively trying to modify the print method signature here.
    public void print(int noOfCopies) { ... }
}
```

the `Student` class will wind up with ***two*** overloaded versions of the `print` method: the version that it has explicitly declared to take one `int` argument plus the parameterless version that it has inherited from the `Person` class.

Private Features and Inheritance

As mentioned earlier, inheritance is an "all or nothing" proposition. That is, if class Y is declared to be a subclass of class X

```
public class Y extends X { ... }
```

then Y cannot pick and choose which features of X it will inherit. In particular, while all of the attributes declared by X will become an inherent part of the "bone structure" of all objects of type Y, some of the attributes of superclass X may not be ***directly referenceable*** by name within subclass Y, depending on what accessibility the attributes were assigned in the superclass.

Consider the following code:

```
public class Person {
  accessibility modifier int age;
  // Other details omitted.
}
```

You learned about two types of accessibility in Chapter 4: `public` and `private`. As it turns out, there are actually ***three*** different explicit accessibility modifier keywords in Java. That is, *<accessibility modifier>* can be one of the following:

- `private`
- `public`
- `protected` (an accessibility modifier that is only relevant within a superclass/subclass relationship, as you'll see shortly)

If the accessibility modifier is omitted entirely, a feature has what is known as **package visibility** by default. We'll discuss this notion in Chapter 13.

Suppose that we were to derive the `Student` class from `Person` as follows:

```
public class Student extends Person {
  // Details omitted.

  // We declare a method that manipulates the age attribute.
  public boolean isOver65( ) {
    if (age > 65) {  // age was declared as an attribute in Person and is
    inherited by Student
        return true;
    } else {
        return false;
    }

  // Other details omitted.
}
```

What will happen when we try to compile this Student class? The answer to this question depends on what accessibility was granted to the age attribute when it was declared in the Person class.

If age is declared to be private in Person, as most attributes typically are

```
public class Person {
  private int age;
  // etc.
}
```

then we'll encounter a compilation error on the line of code **highlighted** in the following code for the Student class:

```
public class Student extends Person {
  // Details omitted.

  public boolean isOver65( ) {
    if (age > 65) {  // this won't compile!
        return true;
    }  else {
        return false;
    }

  // Other details omitted.
}
```

The error message will be

```
cannot find symbol
symbol:    variable age
location:  class Student
if (age > 65) {
    ^
```

Why is this? Since the age attribute is declared to be private in Person, the symbol age is not inherited, so it is not in scope within the Student class. Yet, the memory allocated for a Student object when it is instantiated does indeed allow for storage of a student's age, because as mentioned earlier, it is part of the "bone structure" of a Person, and a Student ***is a*** Person by virtue of inheritance.

What can we do to get around this roadblock? It turns out that we have three choices in Java.

Option #1: We can change the accessibility of age to be `public` in `Person`

```
public class Person {
  public int age;
  // etc.
}
```

thus making it inherited and directly accessible by name in the `Student` subclass. The line of code that previously generated a compiler error in our `Student` subclass, namely

```
if (age > 65) {
```

would now compile without error. The downside of this approach, however, is that by making the age attribute `public` in `Person`, we'd thus be allowing ***client code*** to freely access the age attribute, as well:

```
public class Example {
    public static void main(String[] args) {
        Student s = new Student();
        // Details omitted.
        s.age = 23; // This would compile, but is undesirable.
    }
}
```

This is, generally speaking, a bad practice, for the reasons discussed at length in Chapter 4.

Option #2: We could modify the accessibility of age to be `protected` in `Person`:

```
public class Person {
  protected int age;
  // etc.
}
```

`protected` accessibility is a sort of "middle ground" between `private` and `public` accessibility in that `protected` features are inherited by/***in scope as symbols*** within subclasses; that is, age would now be recognized as a symbol in the `Student` class, such that

```
if (age > 65) {
```

would compile in Student. However, protected features are ***not*** accessible by classes that ***aren't*** derived from the superclass. For example, the following would not compile:

```
public class Example {
    public static void main(String[] args) {
        Student s = new Student();
        // Details omitted.
        s.age = 23; // This would NOT compile if age were declared to be
                    // protected in Person.
    }
}
```

This would be a step in the right direction, but unfortunately requires us to modify the source code of the Person class, which we'd like to avoid if at all possible. Furthermore, we may not even ***have*** the Person source code at our disposal.

Option #3: The ***best*** approach is to allow age to ***remain*** a private attribute of Person, but to use the ***publicly accessible*** getAge/setAge methods that we inherit from the Person class to manipulate the value of a Student's age:

```
public class Person {
  // Let's allow age to REMAIN private!
  private int age;
  // etc.

  // We assume that Person declares public
  // get/set methods for age ... details omitted.
}

public class Student extends Person {
  public boolean isOver65( ) {
    // All is well!  We're using our publicly inherited getAge
    // method to access our Student's age.
    if (getAge() > 65) {
        return true;
    } else {
       return false;
    }
}
```

As we first learned in Chapter 4, it is considered a best practice to always use a class's own "get" and "set" methods to access its attribute values from within its own methods. By doing so, we take advantage of any special processing that the "get"/"set" method might provide relative to that attribute. You've just learned yet another reason doing so is a best practice, when inheritance is involved.

And, if we ***don't*** inherit a `public` "get"/"set" method with which to access a `private` attribute declared by a superclass, then we can argue that we ought not to be "tinkering" with such an attribute in the first place!

Inheritance and Constructors

You learned about constructors as special procedures used to instantiate objects in Chapter 4. Now that you've learned about inheritance, there are a number of complexities with regard to constructors in the context of inheritance hierarchies that I'd like to alert you to.

Constructors Are Not Inherited

Constructors are not inherited. This raises some interesting challenges that are best illustrated via an example.

Let's start by declaring a constructor for the `Person` class that takes two arguments:

```
public class Person {
  private String name;
  private String ssn;

  // Other details omitted.

  public Person(String n, String s) {
    // Initialize our attributes.
    setName(n);
    setSsn(s);

    // Pseudocode.
    do other complex things related to instantiating a Person
  }
}
```

We know from our discussion of constructors in Chapter 4 that the `Person` class now only recognizes one constructor signature—that which takes two arguments—because the default parameterless constructor has been eliminated. (We'll return to discuss the implications of this with regard to inheritance in a moment.)

Next, let's derive the `Student` class from `Person`, declaring ***two*** constructors—one that takes two arguments and one that takes three arguments:

```
public class Student extends Person {
  private String major;

  // Other details omitted.

  // Constructor with two arguments.
  public Student(String n, String s) {
    // Note the redundancy of logic between this constructor and
    // the Person constructor - we'll come back and fix this in a
    // moment.

    // Initialize our attributes.
    setName(n);
    setSsn(s);
    setMajor("UNDECLARED");

    // Pseudocode.
    do other complex things related to instantiating a Person ...
    ... and still more complex things related to instantiating a Student
    specifically.

  }

  // Constructor with three arguments.
  public Student(String n, String s, String m) {
    // More redundancy!

    // Initialize our attributes.
    setName(n);
    setSsn(s);
    setMajor(m);
```

```
    // Pseudocode.
    do other complex things related to instantiating a Person ...
    ... and still more complex things related to instantiating a Student
    specifically.
  }
}
```

As a result of having declared explicit constructors, the `Student` class has also lost its default parameterless constructor.

The first thing that we notice is that we've duplicated code that was provided by the `Person` constructor in ***both*** of the constructors for the `Student` class:

```
    // Initialize our attributes.
    setName(n);
    setSsn(s);

    // Pseudocode.
    do other complex things related to instantiating a Person ...
```

As I've said numerous times before, code redundancy is to be avoided in an application whenever possible; fortunately, Java provides us with a mechanism for reusing a superclass's constructor code from within a subclass's constructor.

super(...) for Constructor Reuse

We accomplish code reuse of a superclass constructor via the same `super` keyword we discussed earlier in the chapter for the reuse of standard methods of a superclass. However, the syntax for reusing constructor code is a bit different. If we wish to explicitly reuse a particular parent class's constructor, we refer to it as follows in the subclass constructor body:

```
super(arguments);  // note that there is no "dot" involved
                   // when reusing CONSTRUCTOR code
```

Using `super(arguments);` to invoke a superclass constructor is similar to using `this(arguments);` to invoke one constructor from within another in the ***same*** class, a technique that you learned about in Chapter 4.

We select whichever of a superclass's constructors we wish to reuse, if more than one exists, by virtue of the arguments that we pass into `super(...)`; because constructors, if overloaded for a given class, all have unique argument signatures, the compiler has no difficulty in sorting out which superclass constructor we're invoking. This is illustrated in the following revised version of the `Student` class (note the **bolded** code):

```
public class Student extends Person {
    // name and ssn are inherited from Person ...
    private String major;

  // Constructor with two arguments.
  public Student(String n, String s) {
    // We're explicitly invoking the Person constructor that accepts two
    // String arguments by passing in two String arguments - namely, the
    // values of n and s.
    super(n, s);

    // Then, go on to do only those things that need to be done uniquely
    // for a Student.
    setMajor("UNDECLARED");
    // Pseudocode.
    do complex things related to instantiating a Student specifically.
  }

  // Constructor with three arguments.
  public Student(String n, String s, String m) {
    // See comments above.
    super(n, s);
    setMajor(m);
    // Pseudocode.
    do complex things related to instantiating a Student specifically.
  }
}
```

One important thing to note is that if we explicitly call a superclass constructor from a subclass constructor using the `super(...)` syntax, the call ***must*** be the ***first*** statement in the subclass constructor—that is, the following constructor would fail to compile:

```
public Student(String n, String s, String m) {
  setMajor(m);

  // This won't compile, because the call to the superclass's
  // constructor must come first in the subclass's constructor.
  super(n, s);
}
```

The following error message would arise:

```
call to super(n, s) must be first statement in constructor
```

The requirement to put a call to `super(...)` as the first line of code in a constructor arises by virtue of the "is a" nature of inheritance. When we create a `Student` object, we are in reality simultaneously creating an `Object`, a `Person`, and a `Student`, all rolled into one. So, whether we ***explicitly*** call a superclass constructor from a subclass constructor using `super(...)` or not, the fact is that Java will ***always*** attempt to execute constructors for all of the ancestor classes for a given class, from most general to most specific in the class hierarchy, before launching into that given class's constructor code. For example, if we are instantiating a `Student`

```
Student s = new Student("Fred", "123-45-6789");
```

then, behind the scenes, an `Object` constructor will automatically be executed first, followed by a `Person` constructor, followed by whichever `Student` constructor we've explicitly invoked—in this example, the one that takes two `String` arguments. The question is, ***which superclass constructors get called if there's more than one defined for a given superclass***? Unless we explicitly invoke a particular constructor as we did in our `Student` constructors, for example

```
public Student(String n, String s) {
  super(n, s);
  // etc.
```

then the parameterless constructor for the superclass is called automatically. That is, if we write a constructor without an ***explicit*** call to `super(`*`args`*`)`, as follows

```
  public Student(String n, String s) {
    // NO EXPLICIT CALL TO super(...)
    setName(n);
    setSsn(s);
    setMajor("UNDECLARED");
    // etc.
  }
```

it is as if we've written

```
  public Student(String n, String s) {
    super(); // implied
    setName(n);
    setSsn(s);
    setMajor("UNDECLARED");
    // etc.
  }
```

instead. ***Herein arises a potential problem***, which is described in the next section.

Replacing the Default Parameterless Constructor

If we don't bother to define any explicit constructors for a particular class, then as discussed in Chapter 4, Java will attempt to provide us with a default parameterless constructor for that class. What we've just seen is that when we invoke the default parameterless constructor for a ***derived*** class such as `Student`, the compiler will automatically try to invoke a parameterless constructor for each of the ***ancestor*** classes in the inheritance hierarchy in top-down fashion. So in writing code as follows

```
// Person.java

public class Person {
    // Attributes ... details omitted.

    // NO EXPLICIT CONSTRUCTORS PROVIDED!!!
    // We're going to be "lazy," and let Java provide us with
    // a default parameterless constructor for the Person class.

    // Methods ... details omitted.
}
```

```
//---------------------------------------

// Student.java

public class Student extends Person {
    // Attributes ... details omitted.

    // NO EXPLICIT CONSTRUCTORS PROVIDED!!!
    // We're going to be "lazy," and let Java provide us with
    // a default parameterless constructor for the Student class.

    // Methods ... details omitted.
}
```

it is as if we've designed our classes as follows:

```
// Person.java

public class Person {
    // Attributes ... details omitted.

    // The default parameterless Person constructor essentially would
    // look like this if we were to code it explicitly:
    public Person() {
        // Calling the default constructor for the Object class.
        super();
    }

    // Methods ... details omitted.
}

//-----------------------------------------

// Student.java

public class Student extends Person {
    // Attributes ... details omitted.

    // The default parameterless Student constructor essentially would
    // look like this if we were to code it explicitly:
```

```
    public Student() {
        // Calling the default constructor for the Person class.
        super();
    }

    // Methods ... details omitted.
}
```

The implication is that if we derive a class B from class A and write no explicit constructors for B, then ***the (default) parameterless constructor of B will automatically look for a parameterless constructor of A***. Thus, code such as the following example ***won't compile***:

```
// Person.java

public class Person {
    private String name;

    // We've written an explicit constructor with one argument for
    // this (super)class; by having done so, we've LOST the
    // Person class's default parameterless constructor.
    public Person(String n) {
        name = n;
    }

    // Note that we haven't bothered to REPLACE the parameterless
    // constructor with one of our own design. This is going to
    // cause us problems, as we'll see in a moment.

    // Methods ... details omitted.
}

//-----------------------------------

// Student.java

public class Student extends Person {
    // Attributes ... details omitted.

    // NO EXPLICIT CONSTRUCTORS PROVIDED!!!
    // We're going to be "lazy," and let Java provide us with
```

```
    // a default parameterless constructor for the Student class.

    // Methods ... details omitted.
}
```

When we try to compile this code, we'll get the seemingly ***very cryptic*** compiler error message regarding the following `Student` class:

```
Student.java: cannot find symbol
symbol:    constructor Person()
location:  class Person
public class Student extends Person {
       ^
```

This is because the Java compiler is trying to create a default parameterless constructor with no arguments for the `Student` class. In order to do so, the compiler knows that it is going to need to be able to invoke a parameterless constructor for `Person` from within the `Student` default constructor—however, no such constructor for `Person` exists!

The best way to avoid such a dilemma is to remember to ***always explicitly program a parameterless constructor for a class X any time you program any explicit constructor for class X, to replace the "lost" default constructor***.

A Few Words About Multiple Inheritance

All of the inheritance hierarchies that we've looked at in this chapter are known informally as **single-inheritance** hierarchies, because any particular class in the hierarchy may only have a ***single*** direct superclass/immediate ancestor. In the hierarchy shown in Figure 5-9, for example

- Classes B, C, and I all have the single direct superclass A.
- Classes D, E, and F have the single direct superclass B.
- Classes G and H have the single direct superclass E.

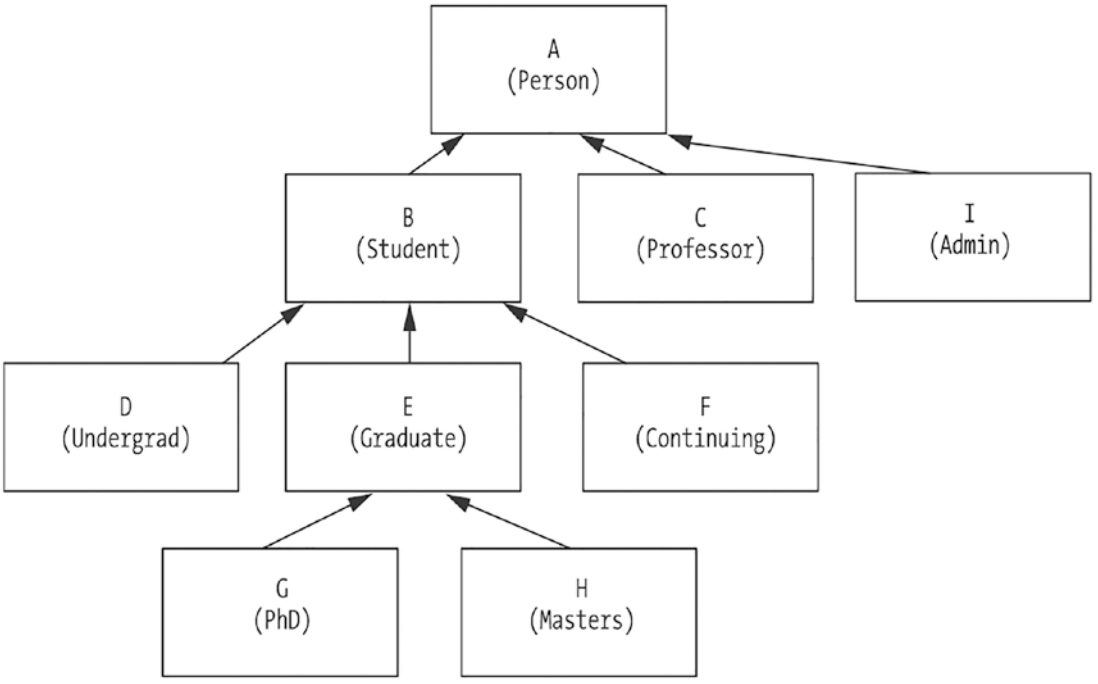


Figure 5-9. *A sample single-inheritance hierarchy*

If we for some reason find ourselves needing to meld together the characteristics of two different superclasses to create a hybrid third class, **multiple inheritance** may seem to be the answer. With multiple (as opposed to single) inheritance, any given class in a class hierarchy is permitted to have ***two or more*** classes as immediate ancestors. For example, we have a `Professor` class representing people who teach classes and a `Student` class representing people who take classes. What might we do if we have a professor who wants to enroll in a class via the SRS? Or a student—most likely a graduate student—who has been asked to teach an undergraduate-level course? In order to accurately represent either of these two people as objects, we would need to be able to combine the features of the `Professor` class with those of the `Student` class—a hybrid `ProfessorStudent`. This might be portrayed in our class hierarchy as shown in Figure 5-10.

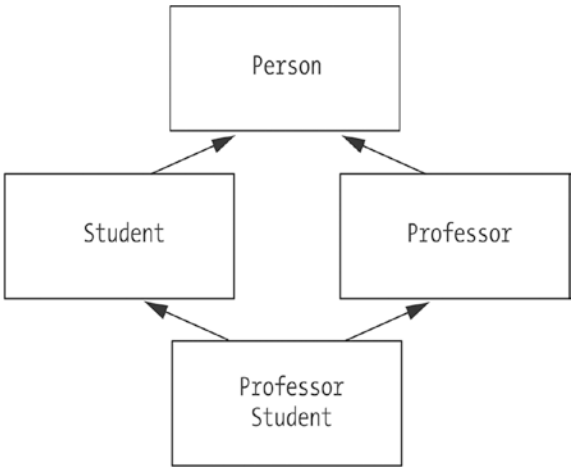


Figure 5-10. *Multiple inheritance permits a subclass to have multiple direct superclasses*

On the surface, this seems quite handy. However, there are many complications inherent in multiple inheritance—so many, in fact, that the Java language designers chose not to support multiple inheritance. Instead, they've provided an alternative mechanism for handling the requirement of creating an object with a "split personality": that is, one that can behave like two or more different real-world entities. This mechanism involves the notion of **interfaces** and will be explored in detail in Chapter 7. Therefore, if you're primarily interested in object concepts only as they pertain to the Java language, you may wish to skip the rest of this section. If, on the other hand, you're curious as to why multiple inheritance is so tricky, then please read on.

Here's the problem with what we've done in the preceding example. We discussed previously that, with inheritance, a subclass automatically inherits the attributes and methods of its superclass. What about when we have two or more direct superclasses? If these superclasses have no overlaps in terms of their features, then we are fine. But what if the direct superclasses in question were, as an example, to have ***conflicting*** features—perhaps `public` methods with the ***same*** signature, but with ***different*** code body implementations, as illustrated in the following simple example?

We'll start with a trivially simple `Person` class that declares one attribute and one method:

```
public class Person {
    private String name;

    // Accessor method details omitted.

    public String printDescription() {
        System.out.println(getName());
        // e.g., "John Doe"
    }
}
```

Later on, we decide to specialize `Person` by creating two subclasses—`Professor` and `Student`—which each add a few attributes along with overriding the `printDescription` method to take advantage of their newly added attributes, as follows:

```
public class Student extends Person {
    // We add two attributes.
    private String major;
    private String studentId;

    // Accessor method details omitted.

    // Override this method as inherited from Person.
    public String printDescription() {
        return getName() + " [" + getMajor() + "; " +
            getStudentId() + "]";
        // e.g., "Mary Smith [Math; 10273]"
    }
}

//-----------------------------------

public class Professor extends Person {
    // We add two attributes.
    private String title;
    private String employeeId;

    // Accessor method details omitted.

    // Override this method as inherited from Person.
```

```
    public String printDescription() {
        return getName() + " [" + getTitle() + "; "
            + getEmployeeId() + "]";
        // e.g., "Harry Henderson [Chairman; A723]"
    }
}
```

Note that both subclasses have overridden the `printDescription` method differently, to take advantage of each class's own unique attributes.

At some future point in the evolution of this system, we determine the need to represent a single object as both a `Student` and a `Professor` simultaneously, and so we create the hybrid class `StudentProfessor` as a subclass of both `Student` and `Professor`. We don't particularly want to add any attributes or methods—we simply want to meld together the characteristics of both superclasses—and so we'd ideally like to declare `StudentProfessor` as follows:

```
// * * * Important Note:  this is not permitted in Java!!! * * *
public class StudentProfessor extends Professor and Student { }
  // It's OK to leave a class body empty; the class itself is not
  // REALLY "empty," because it inherits the features of all of its
  // ancestors.However, we encounter a roadblock to doing so:
```

- `StudentProfessor` cannot inherit both the `Professor` and `Student` versions of the `printDescription` method, because we'd then wind up with two (overloaded) methods with identical signatures in `ProfessorStudent`, which is not permitted by the Java compiler.
- Chances are that we'll want to inherit ***neither***, because neither one takes full advantage of the other superclass's attributes. That is, the `Professor` version of `printDescription` knows nothing about the `getMajor` and `getStudentId` methods inherited from `Student`, nor does the `Student` version of `printDescription` know about the `getTitle` or `getEmployeeId` method inherited from `Professor`.
- If we did wish to use one of the superclass's versions of the method vs. the other, we'd have to invent some way of informing the compiler of which one we wanted to inherit.

This is just a simple example, but it nonetheless illustrates why multiple inheritance can be so problematic. (In a later chapter, we'll introduce the notion of **interfaces** to illustrate how we can work around the lack of multiple inheritance in Java.)

Three Distinguishing Features of an OOPL, Revisited

In Chapter 3, we called out three key features that are required of a programming language in order to be considered truly object-oriented. We've now discussed the first two of these three features at length:

- (Programmer creation of) ***User-defined types***, as discussed in Chapter 3
- ***Inheritance***, as discussed in this chapter
- Polymorphism

All that remains is to discuss **polymorphism**, one of the subjects of an upcoming chapter (Chapter 7, to be precise). We're going to take a bit of a detour first, however, to discuss what we can do to gather up and organize groups of objects as we create them through the use of a special type of object called a **collection**.

Summary

In this chapter, you've learned

- That an ***association*** describes a relationship between classes—that is, a potential relationship between objects of two particular types/classes—whereas a ***link*** describes an actual relationship between two objects belonging to these classes.
- That we define the ***multiplicity*** of an association between classes X and Y in terms of how many objects of type X can be linked to a given object of type Y and vice versa. Possible multiplicities are one-to-one (1:1), one-to-many (1:m), and many-to-many (m:m). In all of these cases, the involvement of the objects at either end of the relationship may be optional or mandatory.

- That an ***aggregation*** is a special type of association that implies containment.
- How to derive new classes based on existing classes through ***inheritance*** and what the do's and don'ts are when deriving these new classes–specifically, how we can ***extend*** a superclass and ***specialize*** it by ***adding features*** or ***overriding methods***.
- How class hierarchies develop over time and what we can do to try to avoid ripple effects to our application as the class hierarchy changes with evolving requirements.
- Some of the complexities of constructors with respect to inheritance.
- Why multiple inheritance can be so troublesome to implement in an OO language.

EXERCISES

1. Given the following pairs of classes, what associations might exist between them from the perspective of the PTS case study described in the Appendix? Be sure to specify multiplicity as well as option/mandatory qualities.
   - `Pharmacist-Prescription`
   - `Prescription-Medication`
   - `Patient-Prescription`
   - `Patient-Medication`
2. Go back to your solution for Exercise 3 at the end of Chapter 4 For all of the classes you suggested, list the pairwise associations that you might envision occurring between them.
3. If the class `FeatureFilm` were defined to have the following methods

   ```
   public void update(Actor a, String title)
   public void update(Actor a, Actor b, String title)
   public void update(String topic, String title)
   ```

   which of the following additional method headers would be allowed by the compiler?

```
public boolean update(String category, String theater)
public boolean update(String title, Actor a)
public void update(Actor a, Actor b, String title)
public void update(Actor a, Actor b)
```

4. [*Coding*] Try coding the `FeatureFilm` class discussed in Exercise 3 to verify your answer for Exercise 3. (Recall that you can "stub out" the `Actor` class by creating a file named `Actor.java` that contains the single line

   ```
   public class Actor { }
   ```

   This satisfies the compiler that `Actor` is a legitimate type.)

5. Given the following simplistic code, which illustrates overloading, overriding, and straight inheritance of methods across four classes—`Vehicle`, `Automobile`, `Truck`, and `SportsCar`

   ```
   public class Vehicle {
     String name;

     public void fuel(String fuelType) {
      // details omitted ...
     }

     public boolean fuel(String fuelType, int amount) {
      // details omitted ...
     }
   }

   public class Automobile extends Vehicle {
     public void fuel(String fuelType, String timeFueled) {
      // details omitted ...
     }

     public boolean fuel(String fuelType, int amount) {
      // ...
     }
   }

   public class Truck extends Vehicle {
   ```

```
  public void fuel(String fuelType) {
   // ...
  }
}

public class SportsCar extends Automobile {
  public void fuel(String fuelType) {
   // ...
  }

  public void fuel(String fuelType, String timeFueled) {
   // ...
  }
}
```

how many different `Fuel` argument signatures would each of the four classes recognize? List these.

6. Reflecting on all that you've learned about Java and OOPLs in general thus far, recount all of the language mechanisms that (a) facilitate code reuse and (b) minimize ripple effects due to requirements changes after code has been deployed.

7. Given the following simplistic classes, `FarmAnimal`, `Horse`, and `Cow`

```
public class FarmAnimal {
  private String name;

  public String getName() {
      return name;
  }

  public void setName(String n) {
      name = n;
  }

  public void makeSound() {
    System.out.println(getName() + " makes a sound ...");
  }
}

public class Cow extends FarmAnimal {
```

```
  public void makeSound() {
    System.out.println(getName() + " goes Moooooo ...");
  }
}

public class Horse extends FarmAnimal {
  public void setName(String n) {
      super.setName(n + " [a Horse]");
  }
}
```

what would be printed by the following client code?

```
Cow c = new Cow();
Horse h = new Horse();
c.setName("Elsie");
h.setName("Mr. Ed");
c.makeSound();
h.makeSound();
```

CHAPTER 6

Collections of Objects

You learned about the process of creating objects based on class definitions, a process known as **instantiation**, in Chapter 3. When we're only creating a few objects, we can afford to declare individualized reference variables for these objects: `Students` `s1`, `s2`, and `s3`, perhaps, or `Professors` `profA`, `profB`, and `profC`. But, at other times, individualized reference variables are impractical:

- Sometimes, there will be too many objects, as when creating `Course` objects to represent the hundreds of courses in a university's course catalog.
- Worse yet, we may not even ***know*** how many objects of a particular type we'll need to instantiate at ***run time*** and so cannot declare a predefined number of reference variables at ***compile time***.

Fortunately, OOPLs solve this problem by providing a special category of object called a **collection** that is used to hold and organize references to other objects.

In this chapter, you'll learn about

- The properties of three generic collection types: **ordered lists**, **sets**, and **dictionaries**
- The specifics of several different predefined Java collection types/classes, along with how we represent and manipulate classic arrays in Java
- How logically related classes like collections are bundled together in Java into **packages** and how we must **import** packages if we wish to make use of the classes that they contain
- How collections enable us to model very sophisticated real-world concepts or situations
- Design techniques for inventing our own collection types

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_6

What Are Collections?

We'd like a way to gather up objects as they are created so that we can manage them as a group and operate on them collectively, along with referring to them individually when necessary, for example:

- A professor may wish to step through all `Student` objects registered for a particular course that the professor is teaching in order to assign their final semester grades.
- The Student Registration System (SRS) application as a whole may need to iterate through all of the `Course` objects in the current schedule of classes to determine if any of them should be canceled due to insufficient enrollment.

We use a special type of object called a **collection** to organize other objects. Think of a collection like an egg carton and the objects it holds like the eggs: both the egg carton and the eggs are objects, but with decidedly different properties.

Collections Are Defined by Classes and Must Be Instantiated

The Java language predefines a number of different collection classes. As with any class, a collection object must be instantiated before it can be put to work. That is, if we merely declare a reference variable to be of a collection type

```
CollectionType<elementType> x;
```

for example

```
ArrayList<Student> x;  // ArrayList is one of Java's predefined
collection types.
```

then until we "hand" `x` a ***specific*** collection object to ***refer*** to, `x` is said to be undefined.

Note the use of less-than greater-than symbols to surround/specify the type of object the collection is to manage: ArrayList<Student>. We will discuss the concept of generics a bit later in this chapter.

We must take the distinct step of using the `new` operator to invoke a specific constructor for the type of collection that we wish to create:

```
x = new CollectionType<elementType>();
```

For example:

```
x = new ArrayList<Student>();
```

Think of the newly created collection object as an ***empty*** egg carton and the reference variable referring to the collection as the handle that allows us to locate and access—***reference***—this "egg carton" in the JVM's memory whenever we'd like.

If we leave the element type off the instantiation statement, as in

```
x = new ArrayList<>();
```

then the element type is implied by the declaration of x to be `Student`. I prefer to explicitly list the element type.

Collections Organize References to Other Objects

Actually, the "collection-as-egg-carton" analogy is a bit of an oversimplification, because rather than physically storing objects ("eggs") in a collection ("egg carton"), we store ***references*** to such objects in the collection. That is, the objects being organized by a collection live physically ***outside*** of the collection in the JVM's memory; only their ***handles*** reside inside of the collection. This notion is illustrated in Figure 6-1.

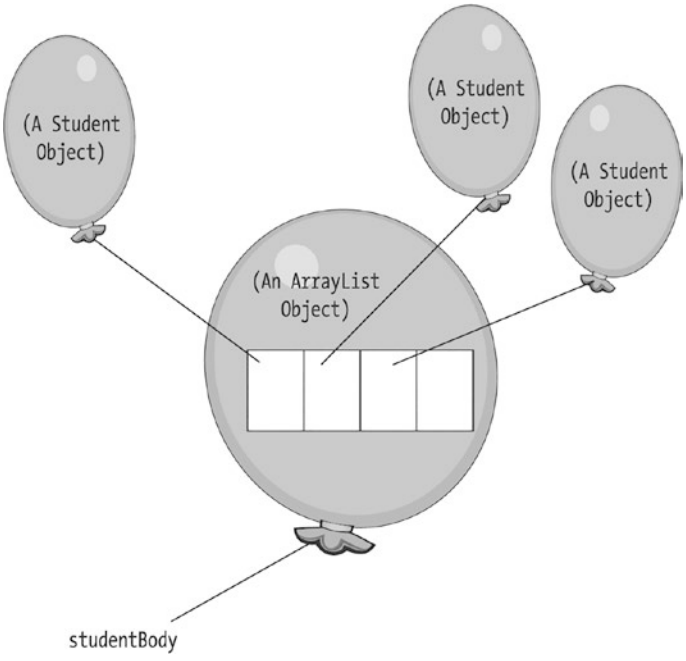


Figure 6-1. *A collection organizes references to objects that live in memory outside of the collection*

Thus, perhaps a better analogy than that of "collection-as-egg-carton" would be that of a collection as an ***address book***: we record an entry in an address book (collection) for each of the people (objects) whom we wish to be able to contact, but the people themselves are physically remote (see Figure 6-2).

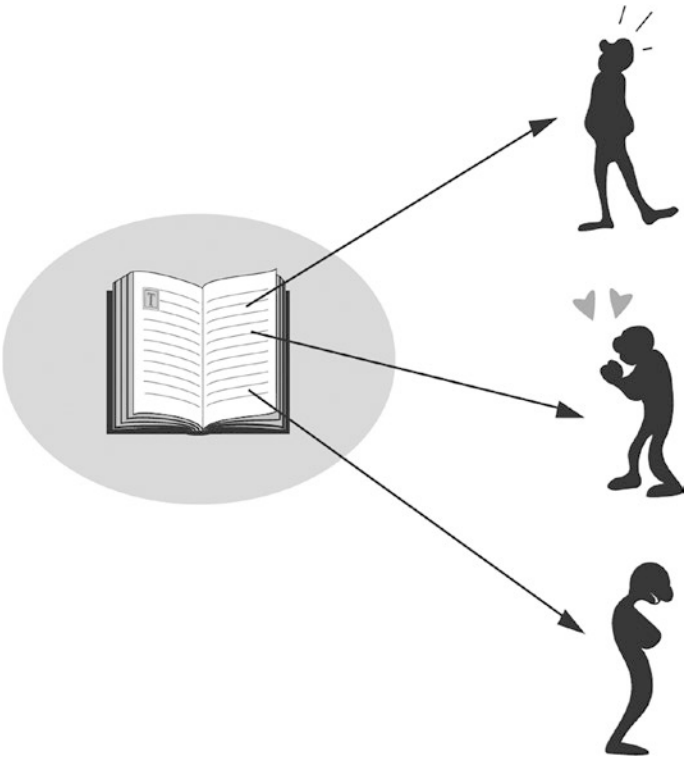


Figure 6-2. *A collection is analogous to an address book, with the people it references as the objects*

Collections Are Encapsulated

We don't need to know the private details of how object references are stored internally to a specific type of collection in order to use the collection properly; we only need to know a collection's ***public features***—in particular, its public method headers—in order to choose an appropriate collection type for a particular situation and to use it effectively.

Virtually all collections, regardless of type and regardless of the programming language in which they are implemented, provide, at a minimum, methods for

- Adding objects
- Removing objects
- Retrieving specific individual objects
- Iterating through the objects in some predetermined order

- Getting a count of the number of objects presently referenced by the collection
- Answering a true/false question as to whether a particular object's reference is in the collection or not

---

Throughout this chapter, we'll talk casually about manipulating ***objects*** in collections, but please remember that, with Java, what we really mean is that we're manipulating object ***references***.

Three Generic Types of Collection

Before diving into the specifics of some of Java's predefined collection classes, let's talk first about the general properties of three basic collection types implemented by most OO languages:

- Ordered lists
- Dictionaries
- Sets

Ordered Lists

An **ordered list** is a type of collection that allows us to insert items in a particular order and later retrieve them in that same order. Specific objects can also be retrieved based on their position in the list (e.g., we can retrieve the first or last or *n*th item).

The vast majority of collection types—ordered lists included—needn't be assigned an explicit capacity (in terms of "egg-carton compartments") at the time that they are instantiated; collections automatically expand as new items are added. Conversely, when an item is removed from most collection types—including ordered lists—the "hole" that would have been left behind is automatically closed up, as shown in Figure 6-3.

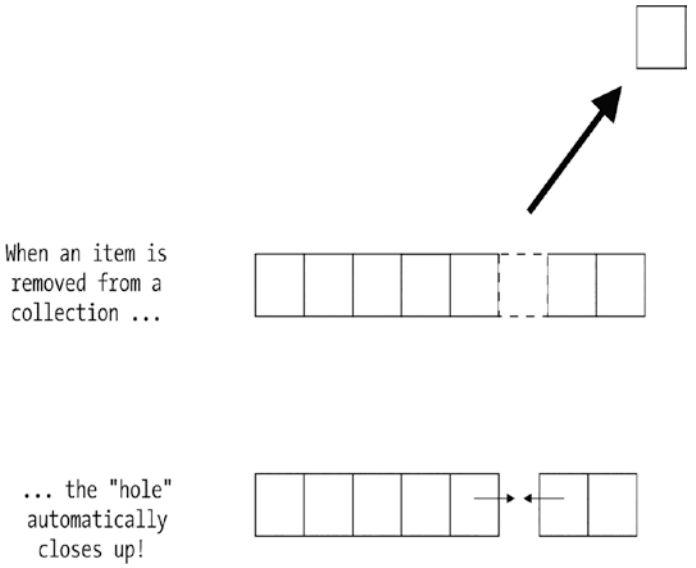


Figure 6-3. *Most collections automatically shrink in size as items are removed*

When we talk about classic arrays as a ***particular*** type of ordered list later in this chapter, we'll see that they alone have some limitations in this regard.

By default, items are added at the end of an ordered list, unless explicit instructions are given to insert an item at a different position.

An example of where we might use an ordered list in building our SRS would be to manage a waitlist for a course that has become full. Because the order with which `Student` objects are added to such a list is preserved, we can be fair about selecting students from the waitlist in first-come, first-served fashion should seats later become available in the course.

Several predefined Java classes implement the notion of ordered list collections: `ArrayList`, `LinkedList`, `Stack`, `Vector`, etc. We'll use the `ArrayList` class in building the SRS, and so we'll discuss the details of working with `ArrayLists` a bit later in this chapter.

Dictionaries

A **dictionary**—also known as a **map**—provides a means for storing each object reference along with a unique **lookup key** that can later be used to quickly retrieve the object (see Figure 6-4).

The key is often contrived based on a unique combination of one or more of the object's attribute values. For example, a `Student` object's student ID number would make an excellent key, because its value is inherently unique for each `Student`.

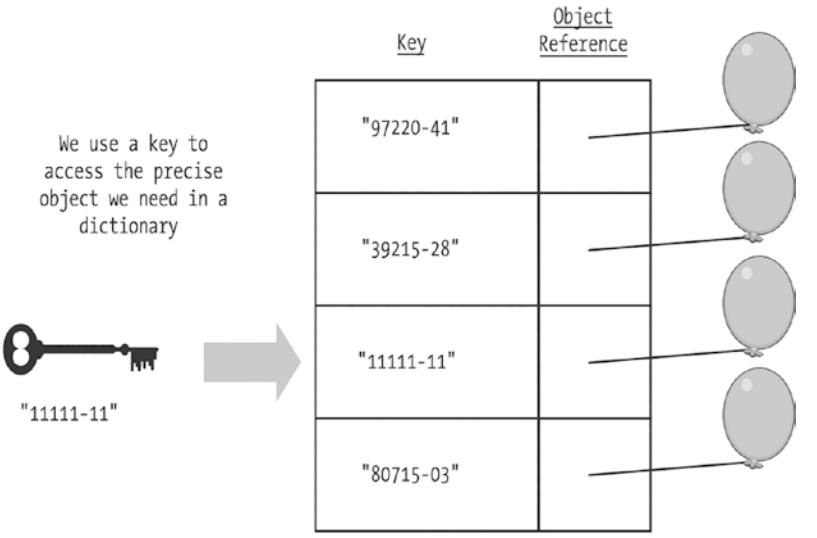


Figure 6-4. *Dictionary collections accommodate direct access by key*

Items in a dictionary can typically also be iterated through one by one, typically in ascending key (or some other predetermined) order.

The SRS might use a dictionary, indexed on course number, to manage its course catalog. With so many course offerings to keep track of, being able to "pluck" the appropriate `Course` object from a collection directly (instead of having to step through an ordered list one by one to find it) adds greatly to the efficiency of the application.

Several examples of predefined Java classes that implement the notion of a dictionary are `HashMap`, `Hashtable`, and `TreeMap`. We'll discuss the details of working with these specific collection types a bit later in this chapter.

Sets

A **set** is an ***unordered*** collection, which means that there is no way to ask for a particular item by number/position once it has been inserted into the set. Using a set is analogous to tossing an assortment of differently colored marbles into a sack (see Figure 6-5): we can reach into the sack (set) to pull out the marbles (objects) one by one, but there is no predictability as to the order with which we'll pull them out as compared with the order in which we put them in.

Figure 6-5. *A set is an unordered collection*

We can also perform tests on a set to determine whether or not a given specific object has been previously added to the set, just as we can answer the question "Is the blue marble in the bag?"

Note that duplicate entries aren't allowed in a set. That is, if we were to create a set of `Student` object references and a reference to a particular `Student` object had already been placed in that set, then a second reference to the ***same*** `Student` object couldn't be subsequently added to the same set; the set would simply ignore our request. This ***isn't*** true of collections in general. If warranted by the requirements of our application, we can add several references to the ***same*** `Student` object to a given ordered list or dictionary instance, as illustrated in Figure 6-6.

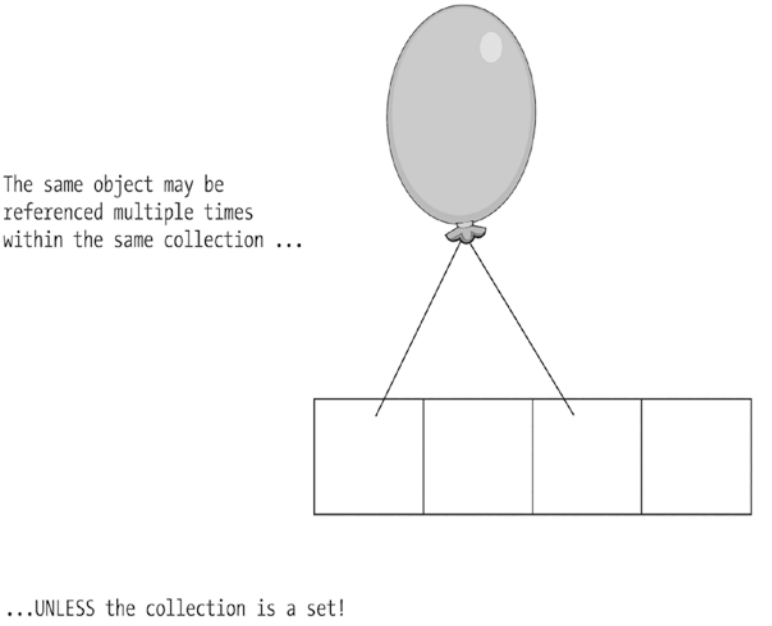


Figure 6-6. *Collections other than sets accommodate multiple references to the same object*

An example of where we might use sets in building our SRS would be to group students according to the academic departments that they are majoring in. Then, if a particular course—say, Biology 216—requires that a student be a biology major in order to register, it would be a trivial matter to determine whether or not a particular `Student` is a member of the "Biology Department Majors" set.

Two predefined Java classes that implement the notion of a set are `HashSet` and `TreeSet`; we do not use sets in building the SRS and so will not be discussing them further.

Note Declaring a TreeSet that will contain custom object types like Students requires that we declare the class as ***implementing the Comparable interface***. We will be discussing interfaces in Chapter 7, but will not be using sets in building the SRS.

Arrays As Simple Collections

One simple type of collection that you may already be familiar with from your work with other programming languages—OO or otherwise—is an **array**.

As mentioned in passing earlier in the chapter, an array is a simple type of ordered list. We can think of an array as a series of compartments, with each compartment sized appropriately for whatever type of data the array as a whole is intended to hold. Arrays typically hold items of like type—for example, `int`(eger)s or `char`(acter)s or, in an OO language, object references (references to `Student` objects, `Course` objects, `Professor` objects, etc.).

Declaring and Instantiating Arrays

Because many newcomers to the Java language are used to programming with arrays in non-OO languages like C, the Java language supports syntax for declaring and manipulating arrays that is borrowed from C and hence is decidedly "un-objectlike"!

The official Java syntax for declaring that a variable `x` will serve as a reference to an array containing items of a particular data type is as follows:

```
datatype[] x;
```

For example:

```
int[] x;
```

which is to be read "`int`(eger) array `x`" (or, alternatively, "`x` refers to an array of `int`s").

Because Java arrays are objects, they must be instantiated using the `new` operator. However, unlike any of the Java predefined collection classes we'll be talking about later in this chapter, we must specify how many items an array is capable of holding (i.e., its **capacity** in terms of its number of elements) when we first instantiate the array; the size is then fixed and cannot be changed.

Here is a code snippet that illustrates the somewhat unusual syntax for constructing an array. In this particular example, we're declaring and instantiating an array designed to hold `Student` object references, as depicted in Figure 6-7:

```
// We declare variable x as a reference to an array object
// that will be used to store 20 Student object references.
Student[] x = new Student[20];
```

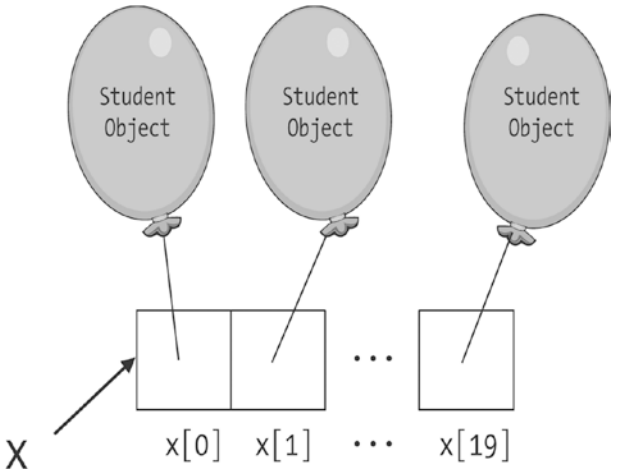


Figure 6-7. *Array* `x` *is designed to hold up to 20* `Student` *references*

This application of the `new` operator with arrays is unusual in that we don't see a typical constructor call ending in parentheses `(...)` following the `new` keyword, the way we do when we're instantiating other types of objects. Instead, we use ***square brackets*** `[...]` to enclose the desired capacity of the array. Despite its unconventional appearance, however, the line of code

```
Student[] x = new Student[20];
```

is indeed instantiating an array object behind the scenes.

It turns out that there is another way to create an array in Java that looks more "objectlike," but the code isn't "pretty":

```
// Declare an array "x" of 20 Student references.

Object x  = Array.newInstance(Class.forName("Student"), 20);
```

To fully appreciate what this code is doing is beyond the scope of what you've learned about objects thus far; suffice it to say that virtually all Java programmers use the shorthand form instead:

```
Student[] x = new Student[20];
```

Accessing Individual Array Elements

Individual array elements are accessed by appending square brackets to the end of the array name, enclosing an `int`(eger) expression representing the **index**, or position relative to the beginning of the array, of the element to be accessed (e.g., `x[3]`). This syntax is known as an **array access expression** and takes the place of using classic "get"/"set" methods to access array contents.

Note that when we refer to individual items in an array based on their index, we start counting at 0. As it turns out, the vast majority of collection types in Java, as well as in other languages, are **zero-based**. So the items stored in `Student[] x` in our previous example would be referenced as `x[0]`, `x[1]`, … , `x[19]`, as was illustrated in Figure 6-7. In the following code example, we declare and instantiate a `double` array of size 3. We then assign the `double` value `4.7` to the "***zeroeth***" (***first***) element of the array. Finally, we retrieve the value of the ***last*** element of the array, which is referred to as `data[2]`, because the size of the array is 3:

```
// Declare an array capable of holding three double values.
double[] data = new double[3];

// Set the FIRST (zeroeth) element to 4.7.
data[0] = 4.7;

// Details omitted ...

// Access the LAST element's value.
double temp = data[2];
```

In the next code example, we populate an array named `squareRoot` of type `double` to serve as a lookup table of square root values, where the value stored in cell `squareRoot[n]` represents the square root of `n`. We declare the array to be one element larger than we need it to be so that we may ignore the zeroeth cell—that is, for ease of lookup, we want the square root of 1 to be contained in cell 1 of the array, not in cell 0:

```
double[] squareRoot = new double[11];  // we'll ignore cell 0

// Note that we're skipping cell 0.
for (int n = 1; n <= 10; n++) {
  squareRoot[n] = Math.sqrt(n);
```

```
  System.out.println("The square root of " + n + " = " +
       squareRoot[n]);
}
```

Here's the output:

```
The square root of 1 = 1.0
The square root of 2 = 1.4142135623730951
The square root of 3 = 1.7320508075688772
The square root of 4 = 2.0
The square root of 5 = 2.23606797749979
The square root of 6 = 2.449489742783178
The square root of 7 = 2.6457513110645907
The square root of 8 = 2.8284271247461903
The square root of 9 = 3.0
The square root of 10 = 3.1622776601683795
```

The `Math.sqrt()` method computes the square root of a `double` argument passed to the method. We're passing in an `int` in the preceding example, which automatically gets cast to a `double`. We'll revisit the `Math` class again in Chapter 7.

Initializing Array Contents

Values can be assigned to individual elements of an array using indexes as shown earlier; alternatively, we can initialize an array with a complete set of values with a single Java statement when the array is first instantiated. In the latter case, initial values are provided as a comma-separated list enclosed in braces. For example, the following code instantiates and initializes a five-element `String` array:

```
String[] names = { "Steve", "Jacquie", "Chloe", "Shylow", "Baby Grode" };
```

Java automatically counts the number of initial values that we're providing and sizes the array appropriately. The preceding approach is much more concise than the equivalent alternative shown here:

```
String[] names = new String[5];
names[0] = "Steve";
names[1] = "Jacquie";
names[2] = "Chloe";
names[3] = "Shylow";
names[4] = "Baby Grode";
```

However, the result in both cases is the same: an array object of capacity 5 is instantiated, the zeroeth (first) element of the array will reference the `String "Steve"`, the next element will reference `"Jacquie"`, and so on.

Note that it isn't possible to bulk load an array in this fashion ***after*** the array has been instantiated, as a separate line of code. That is, the following won't compile:

```
String[] names = new String[4];

// This next line won't compile.
names = {"Mike", "Cheryl", "Mickey", "Will" };
```

If a set of comma-separated initial values aren't provided when an array is first instantiated, the elements of the array are automatically initialized to their zero-equivalent values:

- An `int` array would be initialized to contain integer zeroes (`0`s).
- A `double` array would be initialized to contain floating-point zeroes (`0.0`s).
- A `boolean` array would be initialized to contain the value `false` in each cell.
- And so on.

And, if we declare and instantiate an array intended to hold references to objects, as in

```
Student[] studentBody = new Student[100];
```

then we'd wind up with an array object filled with `null` values.

Manipulating Arrays of Objects

To fill our `Student` array with values other than `null`, we'd have to individually store `Student` object references in each cell of the array. If we wanted to create ***brand-new*** `Student` objects to store in our array, we could write code as follows:

```
studentBody[0] = new Student("Fred");
studentBody[1] = new Student("Mary");
// etc.
```

or

```
Student s = new Student("Fred");
studentBody[0] = s;

// Reuse s!
s = new Student("Mary");
studentBody[1] = s;

// etc.
```

In the latter example, note that we're "recycling" the ***same*** reference variable, `s`, to create many ***different*** `Student` objects. This works because, after each instantiation, we store a ***second*** handle on each newly created object in an array element, thus allowing `s` to let go of ***its*** handle on that same object, as depicted in Figure 6-8. This technique is used frequently, with ***all*** collection types, in virtually all OO programming languages.

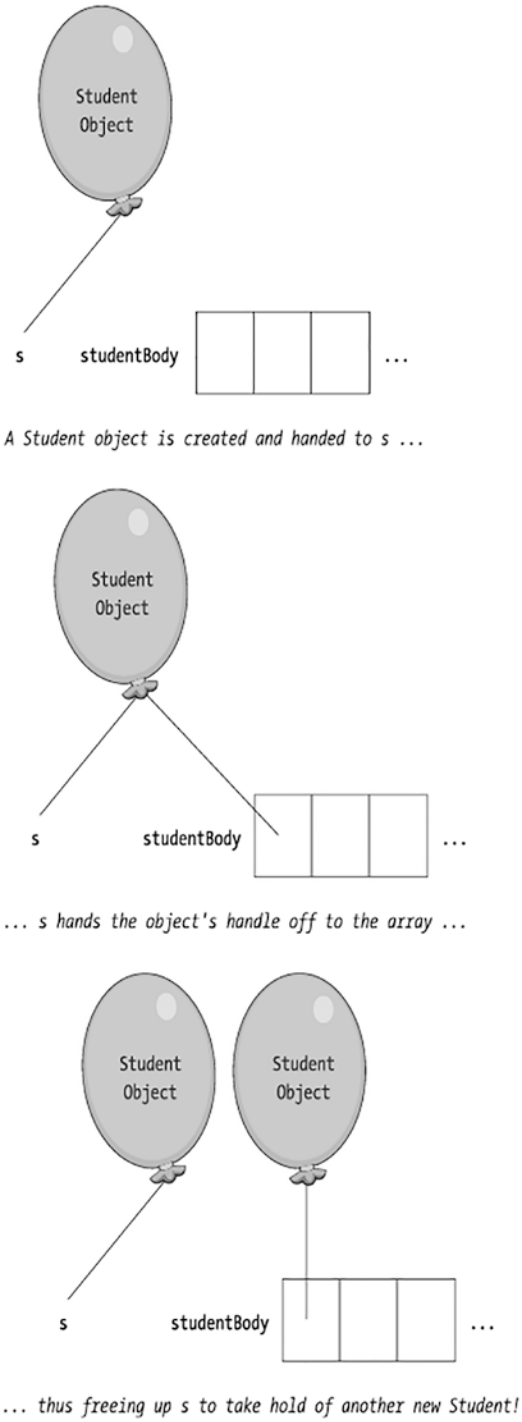


Figure 6-8. *Handing new objects one by one into a collection, using a single reference variable as a temporary handle*

If we're simply using the default (parameterless) constructor to instantiate "bare-bones" `Student` objects, however, we'd probably populate our array using a looping construct and eliminate the need for reference variable `s` entirely:

```
for (int i = 0; i < 20; i++) {
    // We're using the default constructor.
    studentBody[i] = new Student();
}
```

Assuming that we've fully populated all elements of the `studentBody` array, an indexed reference to any particular populated element in the array—for example, `studentBody[i]`—represents a `Student` object and can be used accordingly. In the following line of code, for example, we're invoking the `getName` method on such a `Student`:

```
studentBody[i].getName();  // We're using dot notation to call a method on
                           // studentBody[i], the ith Student object
                              referenced by
                           // the array.
```

Being able to reference individual objects within a collection in this fashion enables us to step through a collection using a looping construct to process its objects one by one. As an example, let's use a `for` loop to iterate through all of the `Students` in our `studentBody` array to print their names—in essence, we're printing a student roster:

```
// Step through all elements of the array.
for (int i = 0; i < studentBody.length; i++) {
    System.out.println(studentBody[i].getName());
}
```

Note the stopping condition on the `for` loop:

```
i < studentBody.length
```

It turns out that arrays have a ***public attribute*** named `length` whose value represents the capacity of the array in terms of the number of elements that it may accommodate. Since we count starting with 0 for the first element, we must stop one short of the value of `length` to avoid stepping past the end of the array.

If ever we accidentally try to step beyond the end of an array at run time, as in the following example

```
// Array x contains 3 elements, indexed as 0, 1, and 2.
int[] x = new int[3];

// However, rather than stopping at element 2, we accidentally try to step
// to (nonexistent) element 3 in our loop:
for (int i = 0; i <= 3; i++) {
    System.out.println(x[i]);
}
```

we'll get the following ***run-time*** error message:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
```

We'll discuss in Chapter 13 how these types of run-time failures—known as **exceptions**—can be programmatically anticipated and handled through a mechanism known as **exception handling**.

Note that the `length` attribute is a ***read-only*** attribute—that is, we cannot assign a value to it explicitly, perhaps in an attempt to enlarge an array at run time (recall that this isn't possible: an array, once sized, cannot be expanded). If we try to do so, as in the following snippet

```
int[] x = new int[3];

// Let's naively try to enlarge array x!
x.length = 10;
```

we'd get the following compilation error:

```
cannot assign a value to final variable length
```

We'll revisit **final** variables in Chapter 7.

We also have to take care when stepping through an array of object references to avoid "land mines" due to empty/`null` elements. That is, if we're iterating through an array, but the array isn't completely filled with `Student` object references, then our invocation of the `getName` method will fail as soon as we reach the first empty/`null` element, because in essence we'd be trying to talk to a nonexistent object. Let's look at an example:

```
// When we first instantiate an array of object references, all cells
contain the
// value null.
Student[] students = new Student[3];

// Store a Student object reference in cells 0 and 1, but leave
// cell 2 empty (i.e., it retains its default value of null).
students[0] = new Student("Elmer");
students[1] = new Student("Klemmie");

// Try to step through the array, printing each Student's name.
// There's a "land mine" lurking at element 2!!!
for (int i = 0; i < studentBody.length; i++) {
    System.out.println(studentBody[i].getName());
}
```

If this code were executed, we'd get the following ***run-time*** error message as soon as the value of `i` reached the value 2

```
Exception in thread "main" java.lang.NullPointerException
```

because the value of `studentBody[2]` is `null`; we cannot invoke a method on (i.e., "talk to") a nonexistent object.

Again, we'll cover how to handle run-time exceptions in Chapter 13.

There's a simple way to avoid such "land mines" in an array–simply test the value of a given array element to see if it's `null` before attempting to address the object at that location:

```
// Step through all elements of the array.
for (int i = 0; i < studentBody.length; i++) {
    // Check for the presence of a valid object reference before trying to
    // "talk to" it via dot notation.
    if (studentBody[i] != null) {
        System.out.println(studentBody[i].getName());
    }
}
```

A More Sophisticated Type of Collection: The ArrayList Class

`Arrays` are ideal for organizing a fixed number of like-typed elements—for example, a `String` array containing the abbreviated names of the days of the week:

```
String[] daysOfWeek = { "Mon.", "Tue.", "Wed.", "Thu.", "Fri.", "Sat.",
"Sun." };
```

However, as mentioned earlier, it's often difficult—if not impossible—for us to predict in advance the number of objects that a collection will need to hold (e.g., how many courses a given student is going to register for when they are logged on to the SRS). When using an `Array` as a collection type, however, we're required to make such a determination when we first instantiate the array, and, once sized, an array can't be expanded. To use an array under such unpredictable circumstances, we'd therefore have to ensure that it was big enough to handle the worst-case scenario, which isn't very efficient.

Fortunately, OO languages provide a wide variety of collection types besides arrays for us to choose from; each has its own unique properties and advantages. As mentioned earlier in the chapter, one important differentiating feature of ***all*** Java collection types ***besides*** arrays is that they needn't be sized in advance: when we add items to a non-array collection, it automatically ***grows*** in size, and as we remove items, the collection will ***shrink*** accordingly (recall Figure 6-3).

Let's now take a look at one of the most commonly used predefined Java collection classes, `ArrayList`, to see how it implements the notion of an ordered list collection.

Using the ArrayList Class: An Example

Here is a simple program that illustrates the use of an `ArrayList` collection to hold on to references to `Student` objects. We'll look at the program in its entirety first, and we'll then step through it to examine key points one by one:

```
import java.util.*;

public class ArrayListExample {
    public static void main(String[] args) {
        // Instantiate a collection.
        ArrayList<Student> students = new ArrayList<Student>();  // or
        simply new ArrayList<>();

        // Create a few Student objects.
        Student a = new Student("Herbie");
        Student b = new Student("Klem");
        Student c = new Student("James");

        // Store references to all three Students in the collection.
        students.add(a);
        students.add(b);
        students.add(c);

        // ... and then iterate through them one by one,
        // printing each student's name.
        for (Student s : students) {
            System.out.println(s.getName());
        }
    }
}
```

Import Directives and Packages

Our example program starts out with an **import directive**, which precedes the `ArrayListExample` class declaration:

```
import java.util.*;

public class ArrayListExample { ... }
```

When we originally discussed Java program anatomy in Chapter 2, we skipped this optional yet key element. I've revised the anatomy figure from Chapter 2 to include `import` directives, as shown in Figure 6-9.

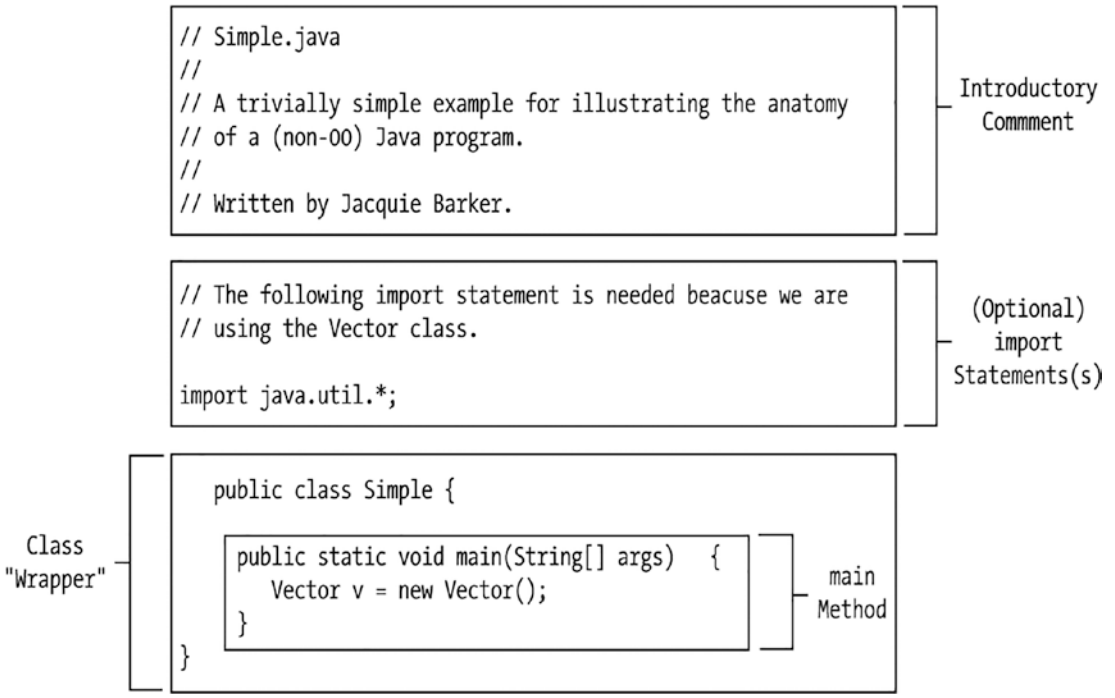


Figure 6-9. *Anatomy of a Java program including import directives*

To appreciate `import` directives, you first must understand the notion of Java **packages**. Because the Java language is so extensive, its various predefined classes are organized into logical groupings called packages. For example, we have

- `java.io`: This package contains classes related to file input/output.
- `java.util`: This package contains a number of utility classes, such as the Java collection classes that you're learning about in this chapter.
- `java.sql`: This package contains classes related to communicating with relational databases.
- And so forth.

Most built-in Java package names start with "`java`", but there are some that start with other prefixes, like "`javax`". And, if we acquire Java classes from a third party, they typically come in a package that starts with the organization's (unique) domain name, reversed. For example, an organization with a domain name of `xyz.com` would typically name their packages `com.xyz.`*`packagename`*, where *`packagename`* describes the logical purpose of the classes included in the package (e.g., `com.xyz.accounting`). By using unique domain names as the basis for package names, we ensure that no two organizations' package names will ever duplicate one another; this is important whenever we incorporate packages from more than one organization in the ***same*** application.

The built-in package named `java.lang` contains the absolute ***core*** of the Java language, and the classes contained within that package—for example, `Math`, `System`, and `String`—are always available to both the Java compiler and the JVM whenever we compile/run Java programs, so we needn't worry about importing `java.lang`. However, if we wish to reference the name of any other predefined class that isn't contained within the `java.lang` package—`ArrayList`, for example—then we must ***import*** the package to which the class belongs, as we did with the `java.util` package in our example program:

```
// Our code needs to refer to the predefined ArrayList class as a type,
and since
// the ArrayList class isn't included in the "core" java.lang
package, we must
// import the package that defines what an ArrayList is.

import java.util.*;

public class ArrayListExample { ... }
```

The asterisk (*) at the end of the `import` directive informs the Java compiler that we wish to import ***all*** of the classes contained within the `java.util` package. As an alternative, we can import individual classes from within a package, as follows:

```
// We can import individual classes by name, to better document where
each class
// that we are using originates.
import java.util.ArrayList;
import java.util.Date;
```

```
import java.io.PrintWriter;
// etc.

public class SomeClass { ... }
```

This approach, of course, requires more typing, but provides better traceability of where each class that we are using in our program originates. Either approach—listing individual classes to be imported one by one or using wildcards to import all classes in a package as a whole—is equally acceptable.

As you'll learn when we discuss the mechanics of the JVM in more detail in Chapter 13, neither approach is less efficient than the other at run time.

If we were to accidentally omit the `import` directive of our `ArrayListExample` program, we'd get the following compilation error once for every occurrence of the symbol `ArrayList` in our code:

```
cannot find symbol
symbol: class ArrayList
```

That is, our output from compiling the following code

```
// WHOOPS!  We've forgotten our import directive.

public class ArrayListExample {
    public static void main(String[] args) {
        // Instantiate a collection.
        ArrayList<Student> students = new ArrayList<>();

        // Create a few Student objects.
        Student a = new Student("Herbie");
        Student b = new Student("Klem");
        Student c = new Student("James");

        // etc.
    }
}
```

would be as follows:

```
ArrayListExample.java:6: cannot find symbol
symbol  : class ArrayList
location: class ArrayListExample
          ArrayList<Student> students = new ArrayList<>();
          ^
ArrayListExample.java:6: cannot find symbol
symbol  : class ArrayList
location: class ArrayListExample
          ArrayList<Student> students = new ArrayList<>();
                                                    ^
2 errors
```

This is because the symbol `ArrayList` is not automatically in the **namespace** of the class that we're compiling—that is, it's not one of the names/symbols that the Java compiler recognizes in the context of that class.

The Namespace of a Class

Generally speaking, the **namespace** for a given class contains the following categories of names, among others:

1. The name of the class itself (e.g., `Student`)
2. The names of all of the features (attributes, methods, etc.) declared by the class
3. The names of any local variables declared within any methods of the class (including parameters being passed in)
4. The names of all classes belonging to the ***same*** package that the class we're compiling belongs to (we'll talk more about packaging classes later in this chapter and then again in Chapter 13)
5. The names of all ***public*** classes in the `java.lang` package: `String`, `Math`, `System`, etc.

> You'll learn what constitutes a `public` (vs. a nonpublic) class in Chapter 13.

6. The names of all ***public*** classes in any ***other*** package that has been ***imported*** by the class that we're compiling
7. The names of all ***public*** features (attributes, methods) of the classes listed in points 5 and 6
8. And so forth

As a simple example, when compiling the following class

```
// Simple.java

public class Simple {
    private int foo;

    public void bar(double x) {
        boolean maybe;
        if (x < 0) maybe = true;
        else maybe = false;
    }
}
```

the compiler would recognize the following names/symbols:

1. `Simple` (the class name)
2. `foo`, `bar` (names of features of the `Simple` class)
3. `x`, `maybe` (local variables of `Simple`'s methods)
4. The names of all classes belonging to the ***default*** (unnamed) package, since `Simple` is ***in*** the default package
5. The names of all public classes in the `java.lang` package: `String`, `Math`, `System`, etc.
6. The names of all public classes in any other package that has been imported. Since `Simple` contains no `import` directives, there are no symbols in this category in `Simple`'s namespace
7. The names of all public features (attributes, methods) of the classes listed in points 4 and 5

The Java compiler compiles classes one by one and "resets" its notion of what is within scope for each new class that it compiles. Therefore, importing a package is only effective for the particular `.java` source code file in which the `import` directive resides. If, for example, you have three separate classes, each stored in its own `.java` file, that all need to manipulate `ArrayLists`, then all three `.java` files must include an appropriate `import` directive, either

```
import java.util.*;
```

or

```
import java.util.ArrayList;
```

We could avoid importing a package/class by **fully qualifying** the names of any classes, methods, etc. that we use from such a package. That is, we can prefix the name of the class or method with the name of the package from which it originates every time that we use it in our code, as shown in the next example:

```
// Note:  NO import directive!

public class Simple {
  public static void main(String[] args) {
    java.util.ArrayList<Student> x = new java.util.ArrayList<>();
    java.util.ArrayList<Professor> y = new java.util.ArrayList<>();
    // etc.
```

This, of course, requires a lot more typing and impairs the readability of our code. By importing a package, on the other hand, we're telling the compiler how to resolve ***simple/unqualified*** names—the ***real*** (***qualified***) name of the `ArrayList` class is `java.util.ArrayList`, but we're able to refer to it by the simple name `ArrayList` because of the `import` directive.

Although most built-in Java packages have names that consist of two terms separated by a period (dot)—for example, `java.awt`—some built-in Java package names consist of three or more dot-separated terms—for example, `java.awt.event`. There's really no limit to the number of terms that can be concatenated to form a package name.

When one package name is a subset of another package name—as in the case of `java.awt` and `java.awt.event`—***both*** must be imported separately if ***both*** are needed in the same class scope. That is, an asterisk at the end of an `import` directive

```
import nameA.nameB.*;
```

only serves to import ***members*** of the specified package, not to extend the package name per se:

```
// This first import directive is not sufficient to import java.awt.event
members;
// it only imports members of the java.awt package.
import java.awt.*;

// We'd need to include this second import directive, as well.
import java.awt.event.*;
```

User-Defined Packages and the Default Package

Java also provides programmers with the ability to logically group their ***own*** classes into packages. For example, we could invent a package such as `com.objectstart.srs` to house all classes related to our SRS application. Then, anyone else wishing to incorporate our SRS classes within an application that they are going to write could include the directive

```
import com.objectstart.srs;
```

in their code.

We will revisit this notion later in the book, but, as it turns out, if we do nothing in particular to take advantage of programmer-defined packages, then as long as all of the compiled `.class` files for an application reside in the ***same*** directory on our computer system, they are automatically considered to be in the same package, known as the **default package**. This is what enables us to write code such as the following

```
public class SRS {
  public static void main(String[] args) {
    Student s = new Student();
    Professor p = new Professor();
    // etc.
```

without using `import` directives, because the class definitions for all of the classes that we've written and compiled—`Student`, `Professor`, and `SRS`—all coexist within the ***same*** (default unnamed) package.

The bottom line is that `import` directives as a building block of a `.java` source code file are only needed if we are using classes that either are not found in package `java.lang` or do not coexist in our own (default) package.

Generics

Let's return to our examination of the `ArrayListExample` program from earlier in the chapter, which is repeated in its entirety here for your convenience:

```
import java.util.*;

public class ArrayListExample {
    public static void main(String[] args) {
        // Instantiate a collection.
        ArrayList<Student> students = new ArrayList<>();

        // Create a few Student objects.
        Student a = new Student();
        Student b = new Student();
        Student c = new Student();

        // Store references to all three Students in the collection.
        students.add(a);
        students.add(b);
        students.add(c);

        // ... and then iterate through them one by one,
        // printing each student's name.
        for (Student s : students) {
            System.out.println(s.getName());
        }
    }
}
```

The next bit of "unusual" syntax occurs on the line of code that declares and instantiates our `ArrayList`:

```
ArrayList<Student> students = new ArrayList<>();
```

The predefined Java collection classes are designed to operate ***generically*** on object references of any type, but then provided a syntactic means of ***constraining*** the type of element that a particular collection is to manage. Now, whenever we want to instantiate a collection such as an `ArrayList`, we can indicate the type of element that the collection is intended to hold by enclosing the type name in angle brackets `<...>` immediately after the class name:

```
ArrayList<Professor> faculty = new ArrayList<>();
ArrayList<String> names = new ArrayList<>();
```

and so forth. In essence, `ArrayList<xxx>` becomes the type of the collection that we're instantiating.

Note that there is a trick to inserting ***primitive*** types (`int`, `double`, `boolean`, etc.) into a collection. We'll discuss this when the notion of **autoboxing** is introduced a bit later in the chapter.

ArrayList Features

In our `ArrayListExample` program, we use the `add` method to insert `Student` references into the collection:

```
// Store references to all three Students in the collection.
students.add(a);
students.add(b);
students.add(c);
```

The `ArrayList` class supports a total of 38 public methods—many of which are common to all collection types—and three overloaded public constructors. Some of the more commonly used `ArrayList` methods, which we'll use in building the SRS, are as follows:

- `boolean add(E element)`: Appends the specified element to the end of the list. `E` refers to whatever type was specified inside of angle brackets `<...>` when the `ArrayList` was first declared/instantiated—for example, `Student` in the following declaration:

```
ArrayList<Student> students = new ArrayList<>();
Student s = new Student();
students.add(s);
// or:
students.add(new Student());
```

With this (and any other) method, we are permitted to pass in an argument of type `E` ***or of any subtype thereof***:

```
ArrayList<Student> students = new ArrayList<>();

// As long as GraduateStudent is derived from Student, all
is well!
students.add(new GraduateStudent());
```

- `void add(int n, E element)`: ***Inserts*** the specified element at the *n*th position in the list, shifting all subsequent items over to make room for the newly added element, for example:

```
Student s = new Student();
students.add(0, s);
// As of the preceding line of code, whatever reference was previously
// in the first (0th) position will now be in the SECOND
position (i.e.,
// in position #1), because we've inserted a NEW Student reference
// in the first (0th) position.
```

- `void clear()`: Removes all elements from the collection, rendering it empty.

Whether or not these elements subsequently get ***garbage collected*** as a result of eliminating their handles from the collection will depend on whether any ***other*** handles are being maintained on these objects. We'll revisit this topic later in this chapter.

- `boolean contains(Object element)`: Returns `true` if the specific object referenced by the argument is also referenced by the `ArrayList` and `false` otherwise:

```
// Create a collection.
ArrayList<Student> x = new ArrayList<>();

// Instantiate two Students, but only add the FIRST of them
// to ArrayList x.
Student s1 = new Student();
Student s2 = new Student();
x.add(s1);

// Declare a third reference variable of type Student, and
have it refer to
// the SAME student as s1:  that is, a Student whose
reference has already been
// added to collection x.
Student s3 = s1;
```

The situation with regard to objects `x`, `s1`, `s2`, and `s3` can be thought of conceptually as illustrated in Figure 6-10.

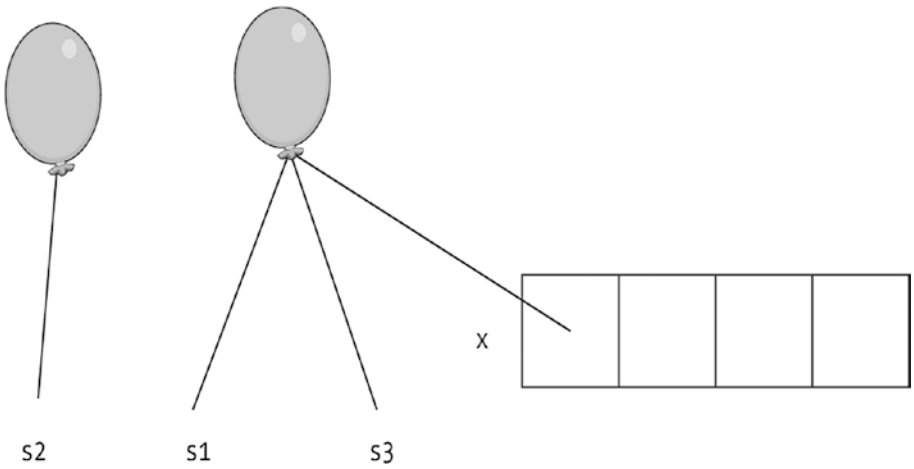


Figure 6-10. *`Student s1` was placed into collection `x`, and because `s1` and `s3` reference the* ***same*** *`Student`, `x` **contains** `s3`*

Continuing with our example, the following first `if` test will return a value of `false`, while the second will return `true`, because `s3` refers to the ***same*** `Student` object that `s1` refers to:

```
// Tests for containment:  the first test will return false ...
if (x.contains(s2)) { ... }
```

```
// ... while the second will return true.
if (x.contains(s3)) { ... }
```

- `int size()`: Returns a count of the number of elements currently referenced by the `ArrayList`. An empty `ArrayList` will report a size of 0.
- `boolean isEmpty()`: Returns `true` if the `ArrayList` in question contains no elements and `false` otherwise.
- `boolean remove(Object element)`: Locates and ***removes*** a single instance of the ***specific object*** referred to by the argument from the `ArrayList`, closing up the hole that would otherwise be left behind. It returns `true` if such an object was found and removed or `false` if the object wasn't found:

```
// Create a collection.
ArrayList<Student> x = new ArrayList<>();

// Instantiate two Students, and add both to x.
Student s1 = new Student();
Student s2 = new Student();
x.add(s1);
x.add(s2);

// Remove s1.
x.remove(s1);
// x now only contains one reference, to s2.
```

- And so forth.

Iterating Through ArrayLists

The `for` loop syntax that we use for iterating through an `ArrayList` (such as the `students ArrayList` in our `ArrayListExample` program) is as follows:

```
for (type referenceVariable : collectionName) {
    // Pseudocode.
    manipulate the referenceVariable as desired
}
```

For example:

```
for (Student s : students) {
    System.out.println(s.getName());
}
```

This `for` statement is to be interpreted as follows: "for every `Student` object (which we'll temporarily refer to as `s`) in the `students` collection, perform whatever logic is specified within the body of the loop." We are then able to refer to `s` within the body of the `for` loop to manipulate it as desired, thus processing the items in the `ArrayList` one by one.

Copying the Contents of an ArrayList into an Array

From time to time, we'll have a need to copy the contents of a collection into an array. We'll use a method declared by the `ArrayList` class with the following header:

type`[] toArray(`*type*`[] `*arrayRef*`)`

That is, we'll invoke the `toArray` method on an `ArrayList` object, passing in an array of the desired *type* as an argument, and the method will in turn hand us back an array that contains a copy of the contents of the `ArrayList`, as follows:

- If the array that we pass in is of sufficient capacity to hold the contents of the `ArrayList`, that ***same*** array object is filled and returned.
- Otherwise, a ***brand-new*** array of the appropriate type and size is ***created***, filled, and returned, and the one that we pass in as an argument is ignored.

Since it's easy to create an array whose size matches that of an existing `ArrayList`—we'll see how to do so in just a moment—we'll do so whenever we have occasion to invoke the `toArray` method on an `ArrayList` within the SRS application.

Let's look at an example. First, we'll create an `ArrayList` named `students`, "stuffing" it with three `Student` references:

```
ArrayList<Student> students = new ArrayList<>();
students.add(new Student("Herbie"));
students.add(new Student("Klemmie"));
students.add(new Student("James"));
```

Next, we'll declare and instantiate an array named `copyOfStudents` that's designed to be just the right size to hold the contents of the `students ArrayList`—note the use of a nested call to `students.size()` to accomplish this:

```
Student[] copyOfStudents = new Student[students.size()];
```

Then, to copy the contents of the `ArrayList` into the `copyOfStudents` array, we simply have to invoke the `toArray` method on `students`, passing in `copyOfStudents` as an argument:

```
students.toArray(copyOfStudents);
```

Let's verify that the copy works by iterating first through the `ArrayList` and then through the array, printing the names of the `Student` objects referenced by each:

```
System.out.println("The ArrayList contains the following students:");

for (Student s : students) {
    System.out.println(s.getName());
}

System.out.println();

System.out.println("The array contains the following students:");

for (int i = 0; i < copyOfStudents.length; i++) {
     System.out.println(copyOfStudents[i].getName());
 }
```

Here's the output:

```
The ArrayList contains the following students:

Herbie
Klemmie
James

The array contains the following students:
Herbie
Klemmie
James
```

Success! Both the array and the `ArrayList` now refer to the same three `Students`, as depicted conceptually in Figure 6-11.

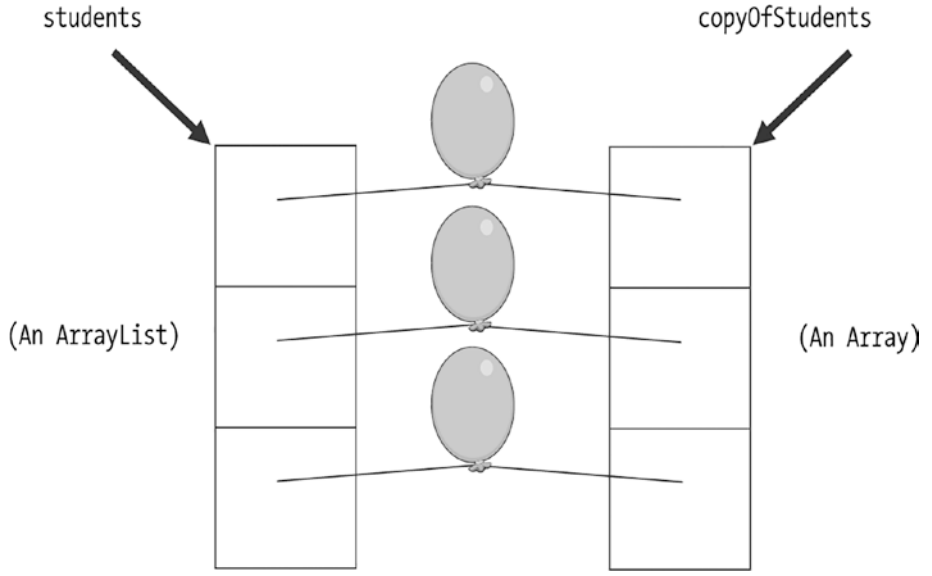


Figure 6-11. *Using the `toArray` method of the `ArrayList` class, we copy the contents of an `ArrayList` to an array*

The HashMap Collection Class

As mentioned earlier in the chapter, a Java `HashMap` is a dictionary-type collection—that is, a `HashMap` gives us direct access to a given object based on a unique key value. Both the key and the object itself can be declared to be of any type.

Let's look at a simple example program called `HashMapExample` that illustrates the basics of manipulating `HashMaps`. This program involves

- Creating and populating a `HashMap` with `Student` object references, using the value of their `idNo` attribute (a `String`) as the key
- Attempting to retrieve several individual `Students` based on specific `idNo` values
- Iterating through the entire collection of `Students`

For purposes of this example, we'll use the following simplified `Student` class declaration:

```
public class Student {
    private String idNo;
    private String name;

    // Constructor.
      public Student(String i, String n) {
        idNo = i;
        name = n;
    }

    public String getName() {
        return name;
    }

    public String getIdNo() {
        return idNo;
    }
}
```

And we'll use the value of each `Student`'s `idNo` attribute as the key.

Let's look at the program in its entirety first, and we'll walk through selected passages afterward:

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        // Instantiate a HashMap with String as the key type and Student as
        // the value type.
        HashMap<String, Student> students = new HashMap<String, Student>();

        // Instantiate three Students; the constructor arguments are
        // used to initialize Student attributes idNo and name,
        // respectively, which are both declared to be Strings.
        Student s1 = new Student("12345-12", "Fred");
        Student s2 = new Student("98765-00", "Barney");
        Student s3 = new Student("71024-91", "Wilma");
```

```
// Insert all three Students into the HashMap, using their idNo
// as a key.
students.put(s1.getIdNo(), s1);
students.put(s2.getIdNo(), s2);
students.put(s3.getIdNo(), s3);

// Retrieve a Student based on a particular (valid) ID.
String id = "98765-00";
System.out.println("Let's try to retrieve a Student with ID =
" + id);
Student x = students.get(id);

// If the value returned by the get method is non-null, then
// we indeed found a matching Student ...
if (x != null) {
    System.out.println("Found!  Name = " + x.getName());
}
// ... whereas if the value returned was null, then we didn't find
// a match on the id that was passed in as an argument to get().
else {
    System.out.println("Invalid ID:  " + id);
}

System.out.println();


// Try an invalid ID.
id = "00000-00";
System.out.println("Let's try to retrieve a Student with ID =
" + id);
x = students.get(id);

if (x != null) {
    System.out.println("Found!  Name = " + x.getName());
}
else {
    System.out.println("Invalid ID:  " + id);
}
```

```
        System.out.println();
        System.out.println("Here are all of the students:");
        System.out.println();

        // Iterate through the HashMap to process all Students.
        for (Student s : students.values()) {
            System.out.println("ID:  " + s.getIdNo());
            System.out.println("Name:  " + s.getName());
            System.out.println();
        }
    }
}
```

Here's the output:

```
Let's try to retrieve a Student with ID = 98765-00
Found!  Name = Barney

Let's try to retrieve a Student with ID = 00000-00
Invalid ID:  00000-00

Here are all of the students:

ID:  12345-12
Name:  Fred

ID:  98765-00
Name:  Barney

ID:  71024-91
Name:  Wilma
```

The first point of interest is that when we declare and instantiate a `HashMap`, we must specify types for ***two*** elements: the ***key***, which is of type `String` in our example, and the ***value*** that this key represents–the value being looked up–which is of type `Student` in our example:

```
HashMap<String, Student> students = new HashMap<String, Student>();
```

We use the put method to insert an object into a HashMap:

```
students.put(s1.getIdNo(), s1);
```

This method inserts the object represented by the ***second*** argument (s1, in the preceding example) into the collection with a retrieval key value represented by the ***first*** argument (the idNo of s1, retrieved by calling the getIdNo method, in the preceding example).

If we attempt to insert a second object into a HashMap with a key value that duplicates the key of an object that is already referenced by the HashMap, the put method will silently ***replace*** the original object reference with the new reference. If we want to ***avoid*** such inadvertent object replacement in a HashMap, we can use the containsKey method, which returns a value of true if a particular key already exists in the HashMap and false otherwise. Here's an example of this method's use:

```
// If it is NOT the case that the students HashMap already contains
// a key value matching the idNo of student s1 ...
if (!(students.containsKey(s1.getIdNo()))) {
    // ... then it is safe to add such a reference.
    students.put(s1.getIdNo(), s1);
}
else {
    // Another Student reference with the same idNo value is already in the
    HashMap.
    System.out.println("ERROR:  Duplicate student ID found in HashMap:  " +
        s1.getIdNo());
}
```

The get method is used to retrieve an object reference from the HashMap whose key value matches the value passed in as an argument to the method:

```
Student x = students.get(id);
```

If no match is found, a value of null is returned.

The syntax that we've used for iterating through the `students HashMap` in our `HashMapExample` program is very similar to the code that we used to iterate through `ArrayLists` earlier in the chapter:

```
// Iterate through the HashMap to process all Students.
for (Student s : students.values()) {
    ...
}
```

The only subtle difference is that we are invoking the `values` method on the `students` collection to access the (`Student`) objects contained within the `HashMap`, bypassing their ***keys***, as illustrated in Figure 6-12.

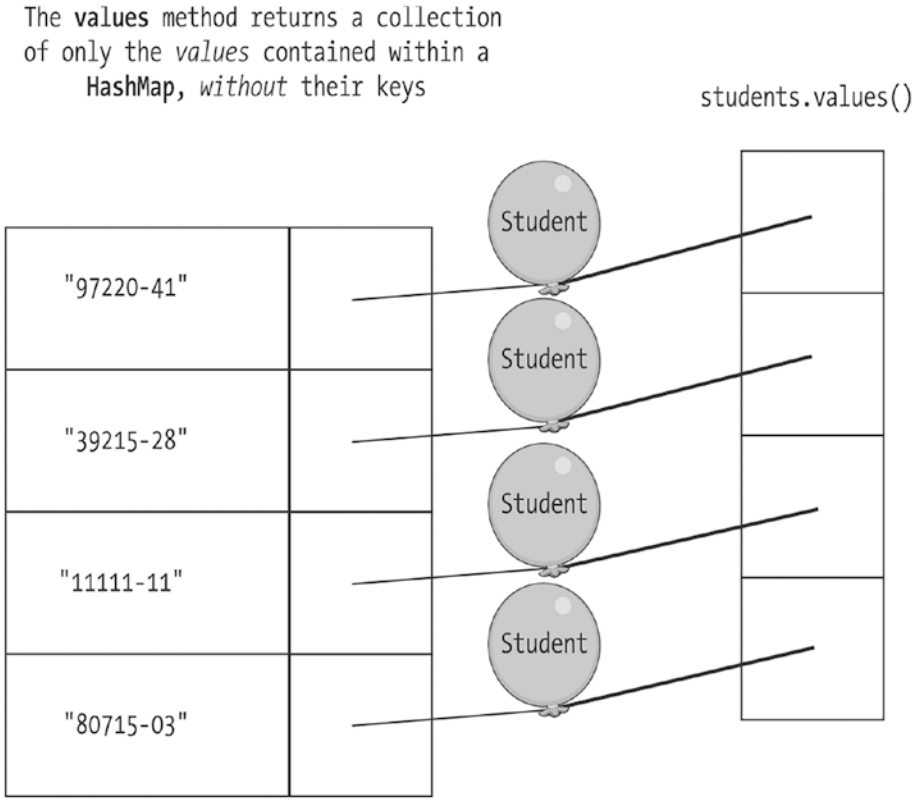


Figure 6-12. *The `values` method returns a collection of values (only) from a `HashMap`*

Some of the other commonly used methods declared by the HashMap class are as follows:

- `Object remove(Object key)`: Removes the reference to the object represented by the given key from the HashMap.
- `boolean contains(Object value)`: Returns `true` if the specific object passed in as an argument to the method is already referenced by the HashMap, ***regardless of what its key value might be***; otherwise, returns `false`. Here's an example:

```
// Instantiate a new HashMap and two Students.
HashMap<String, Student> x = new HashMap<String, Student>();
Student s1 = new Student("12345-12", "Fred");
Student s2 = new Student("98765-00", "Barney");

// Insert only the first Student into the HashMap.
x.put(s1.getIdNo(), s1);

// Maintain a second handle on each of the two Students.
Student s3 = s1;  // s1, and hence s3, is in the HashMap.
Student s4 = s2;  // s2, and hence s4, are NOT in the HashMap.
```

The situation with regard to objects `x`, `s1`, `s2`, `s3`, and `s4` can be thought of conceptually as illustrated in Figure 6-13.

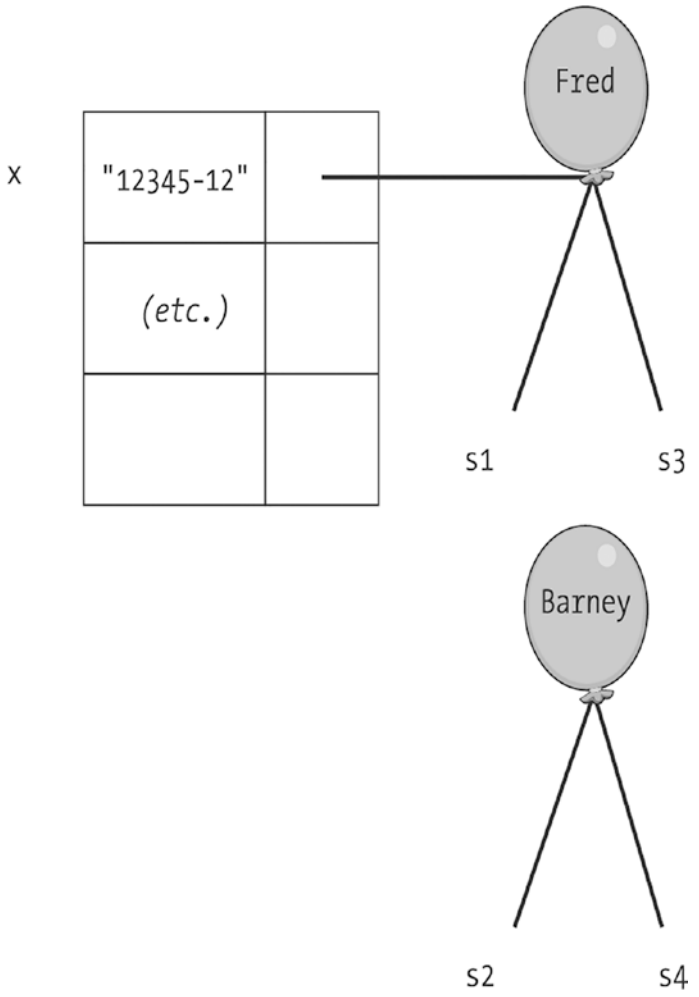


Figure 6-13. *Two* `Student` *objects, only one of which is referenced by* `HashMap` `x`

The results of calling `x.contains(...)` with respect to `s3`, then `s4`, are as follows:

```
// This first test will evaluate to true ...
if (x.contains(s3)) { ...

// ... while this second test will evaluate to false.
if (x.contains(s4)) { ...
```

- `int size()`: Returns a count of the number of key/object pairs currently stored in the `HashMap`.
- `void clear()`: Empties out the `HashMap` of all key/object pairs, as if it had just been newly instantiated.
- `boolean isEmpty()`: Returns `true` if the `HashMap` contains no entries; otherwise, returns `false`.

The TreeMap Class

The Java `TreeMap` class is another dictionary-type collection. `TreeMaps` are very similar to `HashMaps`, with one notable difference:

- When we iterate through a `TreeMap`, objects are automatically retrieved from the collection in ***ascending key (sorted) order***.
- When we iterate through a `HashMap`, on the other hand, there's no guarantee as to the order in which items will be retrieved.

Let's write a program to demonstrate this difference between `HashMaps` and `TreeMaps`. In our program, we'll instantiate one of each of these two types of collection. This time, we'll insert `Strings` into the collections rather than `Students`; we'll let the same `String` serve as ***both*** the key and the value:

```
import java.util.*;

public class TreeHash {
    public static void main(String[] args) {
    // Instantiate two collections -- a HashMap and a TreeMap -- with
    // String as both the key type and the object type.
    HashMap<String, String> h = new HashMap<String, String>();
    TreeMap<String, String> t = new TreeMap<String, String>();

    // Insert several Strings into the HashMap, where the String serves
    // as both the key and the value.
    h.put("FISH", "FISH");
    h.put("DOG", "DOG");
    h.put("CAT", "CAT");
```

```
        h.put("ZEBRA", "ZEBRA");
        h.put("RAT", "RAT");

        // Insert the same Strings, in the same order, into the TreeMap.
        t.put("FISH", "FISH");
        t.put("DOG", "DOG");
        t.put("CAT", "CAT");
        t.put("ZEBRA", "ZEBRA");
        t.put("RAT", "RAT");

        // Iterate through the HashMap to retrieve all Strings ...
        System.out.println("Retrieving from the HashMap:");
        for (String s : h.values()) {
            System.out.println(s);
        }

        System.out.println();

        // ... and then through the TreeMap.
        System.out.println("Retrieving from the TreeMap:");
            for (String s : t.values()) {
                System.out.println(s);
            }
        }
}
```

Here's the output:

```
Retrieving from the HashMap:
ZEBRA
CAT
FISH
DOG
RAT

Retrieving from the TreeMap:
CAT
DOG
```

```
FISH
RAT
ZEBRA
```

Note that the `TreeMap` did indeed sort the `String`s, whereas the `String`s were retrieved in an arbitrary order—neither in the order in which they were inserted nor in sorted order—from the `HashMap`.

All of the other methods that we discussed for the `HashMap` class work in the same fashion for `TreeMap`s.

If `TreeMap`s are effectively identical to `HashMap`s with the added benefit of sorted iteration, why don't we simply ignore the `HashMap` class and always use the `TreeMap` class to create dictionary collections instead? The answer lies in the fact that dictionaries can use ***any object type*** as a key.

If we use `String`s as keys, as we've done in all of our examples thus far, a `TreeMap` has no trouble determining how to sort them, because the `String` class defines a `compareTo` method that the `TreeMap` class takes advantage of. However, if we use a ***user-defined type*** as a key, the burden is on us to programmatically define what it means to sort that object type.

Let's say, for example, that we create a dictionary collection where a `Department` object serves as the key and the `Professor` who chairs the department is the value referenced by a given key. If we declare the collection to be a `TreeMap`, we must define what it means for one `Department` to "come before" another in sorted fashion if we plan on iterating through the collection. The code required to do so is rather advanced—certainly beyond the scope of what we've learned about Java thus far. Suffice it to say that if we don't truly ***need*** to iterate through a dictionary in sorted key order, it's not worth the extra trouble of using `TreeMap` when `HashMap` will do quite nicely.

The Same Object Can Be Simultaneously Referenced by Multiple Collections

As mentioned earlier, when we talk about inserting an object into a collection, what we really mean is that we're inserting a reference to the object, not the object itself. This implies that the ***same*** object can be referenced by ***multiple*** collections simultaneously.

Think of a person as an object and their telephone number as a reference for reaching that person. Now, as I proposed earlier in this chapter, think of an address book as a collection: it's easy to see that the ***same*** person's phone number (reference) can be recorded in many ***different*** address books (collections) simultaneously.

Let's consider an example relevant to the SRS. Given the students who are registered to attend a particular course, we may simultaneously maintain the following:

- An ordered list of these students for purposes of knowing who registered first on the waitlist for a follow-on course
- A dictionary that allows us to retrieve a given `Student` object based on the student's `name`
- Perhaps even a second SRS-wide dictionary that organizes ***all*** students at the university based on their student ID numbers

This is depicted conceptually in Figure 6-14.

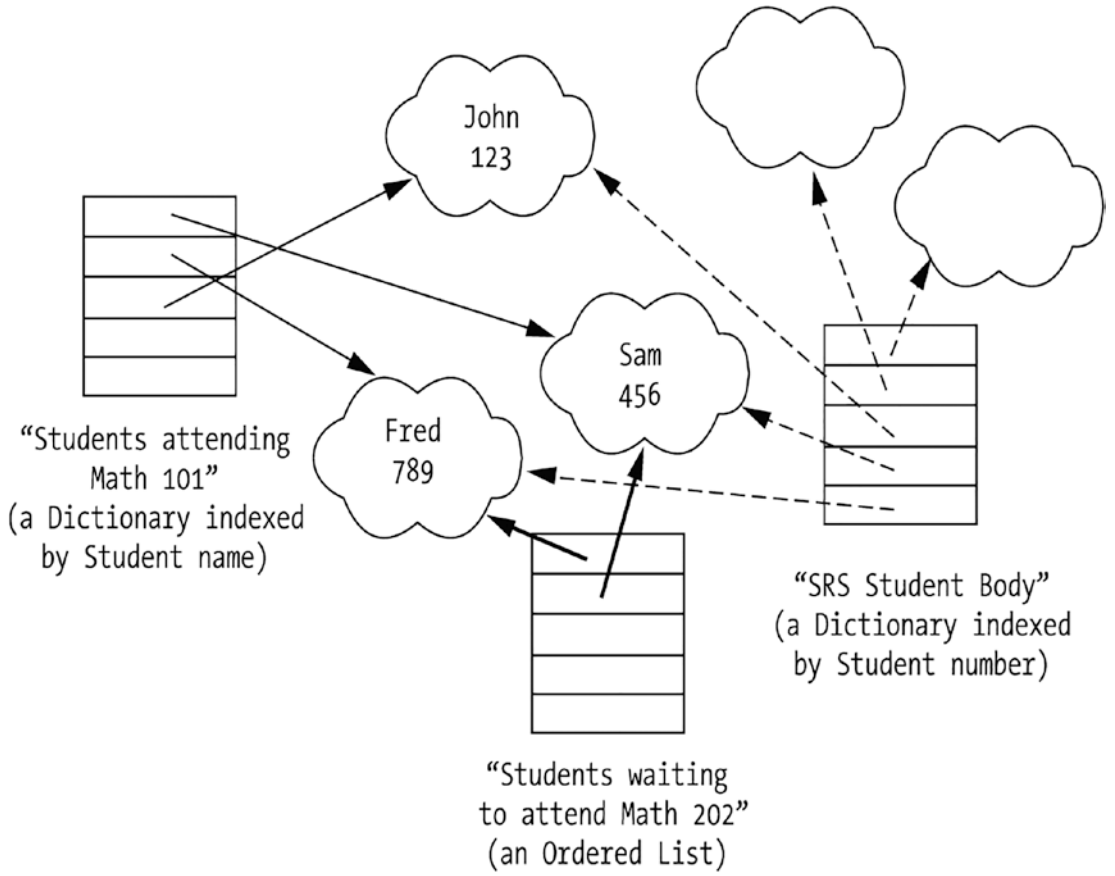


Figure 6-14. *A given object may be referenced by multiple collections simultaneously*

One common mistake made by beginning OO programmers is to assume that if a given collection is emptied (perhaps via an explicit call to its `clear` method), then the objects that the collection was previously referencing will be garbage collected. Recall our discussion of garbage collection from Chapter 3: only when there are no longer ***any*** handles on a given object will its memory will be recycled by the JVM. Given that objects are often referenced by multiple collections simultaneously, we cannot assume that clearing a single collection will free up the objects that it is referencing. For example, if we were to clear the contents of the "Students attending attend Math 101" collection of Figure 6-14, the "John," "Fred," and "Sam" `Student` objects would still be referenced by two other collections. Unless these `Student` objects were subsequently removed from those other collections, as well, they would not be garbage collected.

Inventing Our Own Collection Types

As mentioned earlier, different types of collections have different properties and behaviors. You must therefore familiarize yourself with the various predefined collection types available for your OO language of choice and choose the one that is the most appropriate for what you need in a given situation. Or, if none of them suits you, invent your own! This is where we start to get a real sense of the power of an OO language. Since we have the ability to invent our own user-defined types, it of course follows that we have free rein to define our own ***collection*** types.

We have several ways to create our own collection types:

- ***Approach #1***: We can design a brand-new collection class from scratch.
- ***Approach #2***: We can use the techniques that we learned in Chapter 5 to extend a predefined collection class.
- ***Approach #3***: We can create a "wrapper" class that encapsulates one of the built-in collection types, to "abstract away" some of the details involved with manipulating the collection.

Let's discuss each of these three approaches in turn.

Approach #1: Designing a New Collection Class from Scratch

Creating a brand-new collection class from scratch is typically quite a bit of work. Since most OO languages provide such a wide range of predefined collection types, it's almost always possible to find a preexisting collection type to use as a starting point, in which case one of the other two approaches would almost always be preferred.

If we ***were*** to want to create a new collection class from scratch, however, we'd almost certainly want such a class to take advantage of the predefined `Collection` **interface**. We'll discuss the notion of interfaces in general, and of the `Collection` interface specifically, in Chapter 7.

Note that, despite the fact that an array serves as a simple sort of ordered list collection, it is not formally a Java `Collection` in the "capital C" sense of the word.

Approach #2: Extending a Predefined Collection Class (MyIntCollection)

To illustrate this approach, let's extend the `ArrayList` class to create a collection class called `MyIntCollection`. An object of type `MyIntCollection` will be able to, at a minimum, respond to all of the same service requests that an `ArrayList` can respond to, because by virtue of inheritance, `MyIntCollection` ***is*** an `ArrayList`. However, we want our `MyIntCollection` class to do some extra work: we want it to keep track of the smallest and largest `int` values stored within a given `MyIntCollection` instance. To accomplish this, we'll add a few new features, along with overriding the `add` method that we'd otherwise inherit as is from `ArrayList`.

We'll look at the code for the `MyIntCollection` class in its entirety first, and then we'll walk through it step by step afterward:

```
import java.util.ArrayList;

public class MyIntCollection extends ArrayList<Integer> {
    // We inherit all of the attributes and methods of a standard ArrayList
    // as is, then define a few extra attributes and methods of our own:
    // two ints to keep track of the smallest and largest values
    // added to the collection, plus another int to keep a running
    // total of all values added to the collection.
    private int smallestInt;
    private int largestInt;
    private int total;

    // Replace the default constructor.
    public MyIntCollection() {
     // Do everything defined by the constructor of the ArrayList
        // base class first - we needn't know what those things are, simply
        // that we ought to do them!
        super();

        // Initialize the total.
        total = 0;
    }

    // Override the add() method as inherited from ArrayList.
```

```
    public boolean add(int i) {
        // Remember this int as the largest/smallest, if appropriate.
        // (The FIRST time we add a value, that value will, by definition,
        // be BOTH the smallest AND the largest that we've seen so far!)
        if (this.isEmpty()) {
            smallestInt = i;
            largestInt = i;
        }
        else {
            if (i < smallestInt) {
                smallestInt = i;
            }
            if (i > largestInt) {
                largestInt = i;
            }
        }

        // Include this value in the running total.
        total = total + i;

        // Insert the int into the collection using the add method as
        implemented
        // by the ArrayList base class.  Again, we needn't understand
        // the inner workings of HOW this method does so ...
        return super.add(i);
    }

    // Several new methods.

    public int getSmallestInt() {
        return smallestInt;
    }

    public int getLargestInt() {
        return largestInt;
    }

    public double getAverage() {
        // Note that we must cast ints to doubles to avoid
```

```
        // truncation when dividing.
        return ((double) total) / ((double) this.size());
    }
}
```

Now, let's walk through selected portions of the `MyIntCollection` code.

Wrapper Classes for Primitive Types

The first unusual bit of syntax that we notice is with regard to the class that we're extending:

```
public class MyIntCollection extends ArrayList<Integer> {
```

If we are going to be placing `int` values in our collection, why are we designating `Integer` as the type of element to be inserted?

Unlike arrays, whose elements may be either primitive or reference types, Java collections are designed to hold only ***reference types***. An `int` is not an object, and so if we wish to store primitive values in collections, we must "box" them ***inside*** of objects, as illustrated conceptually in Figure 6-15.

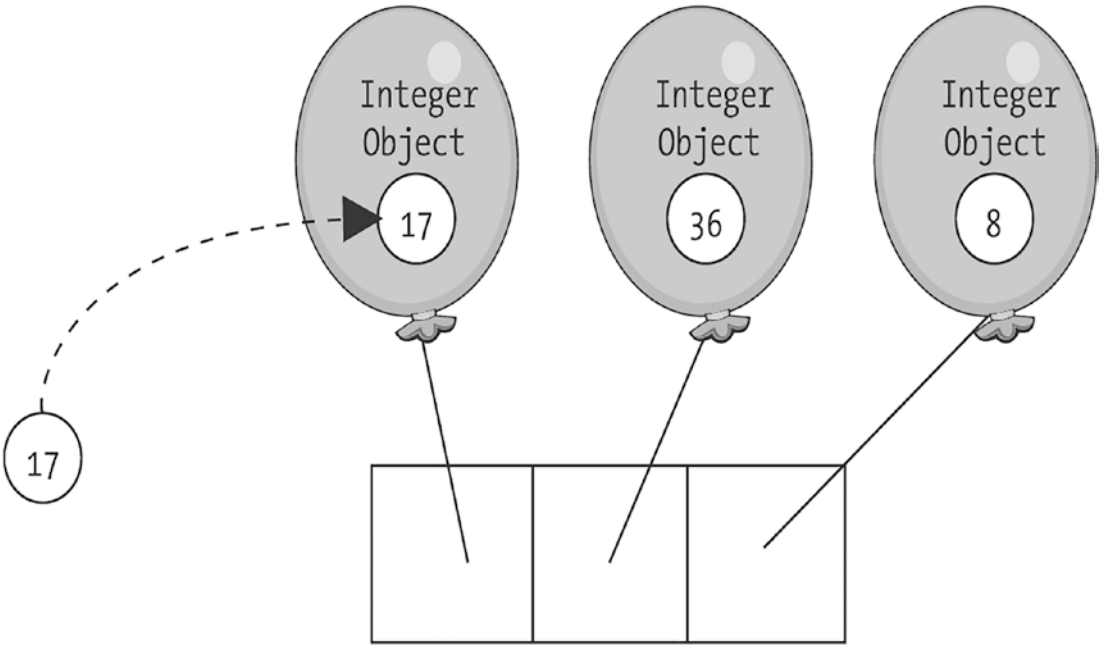


Figure 6-15. *We must "box" primitive values in objects in order to insert them into a collection*

The Java language provides a different "wrapper" class to serve as a "box" for each of the eight distinct primitive types: `Integer`, `Float`, `Double`, `Byte`, `Short`, `Long`, `Boolean`, and `Character`. All of these classes are included in the core `java.lang` package.

Prior to Java 5.0, programmers were responsible for writing the explicit code necessary to "wrap" primitive values inside of corresponding wrapper objects before inserting them into a collection, as well as the code to "unwrap" them when retrieving them from a collection; as of Java version 5.0, the **autoboxing** feature was introduced to save us the trouble of having to do this explicitly. Simply by declaring a collection as containing the appropriate wrapper type, we are free to insert and retrieve primitive values as is:

```
ArrayList<Integer> x = new ArrayList<Integer>();

// Directly add a primitive (int) value to the ArrayList;
// it automatically gets "boxed" inside of an Integer object.
x.add(17);

// Details omitted ...

// Directly retrieve a primitive (int) value from the
// ArrayList; it automatically gets "unboxed" from its
// enclosing Integer object.
int y = x.elementAt(0); // y now has the value 17
```

We'll revisit the wrapper classes for primitive types several more times in the book, as they serve many useful purposes.

Reusing a Base Class Constructor

The constructor that we've provided for the `MyIntCollection` class takes advantage of the `super` keyword to reuse the constructor code of the base `ArrayList` class, a technique that we discussed in Chapter 5:

```
    public MyIntCollection() {
        // Do everything defined by the constructor of the ArrayList
        // base class first - we needn't know what those things are, simply
        // that we ought to do them!
        super();
```

It isn't necessary for us to know the behind-the-scenes details that take place when an instance of an `ArrayList` is created. Simply by including

```
super();
```

as the first line of code in our constructor, we ensure that such details are taken care of for us.

Strictly speaking, you could omit the preceding line of code, for as you learned in Chapter 5, a call to `super()` is ***implied*** as the first line of code of a derived class's constructor. However, inserting this line of code explicitly doesn't hurt and in fact clarifies what is actually happening when this constructor is executed.

Overriding the add Method

We override the `add` method of the `MyIntCollection` class as inherited from `ArrayList` so that we may continuously monitor values as we add them to our custom collection to keep track of what the smallest and largest values have been:

```
public boolean add(int i) {
    // Remember this int as the largest and/or the smallest,
    // as appropriate.  (The FIRST time we add a value, it by default
    // will be BOTH the smallest AND the largest!)
    if (this.isEmpty()) {
        smallestInt = i;
        largestInt = i;
    }
    else {
        if (i < smallestInt) smallestInt = i;
        if (i > largestInt) largestInt = i;
    }

    // Include this value in the running total.
    total = total + i;
```

Finally, by invoking `super.add(i)` from our overridden `add` method, we're ensuring that we do everything that the `ArrayList` base class does when adding an item to its

internal collection—again without having to know the details of what is happening behind the scenes. And, because we must return a `boolean` value from our `add` method per the (overridden) method header, we can accomplish this by simply returning the value that results from this base class method call:

```
        // Insert the int into the collection using the add method as
           implemented
        // by the ArrayList base class.  Again, we needn't understand
        // the inner workings of HOW this method does so ...
        return super.add(i);
    }
```

The remainder of the code for the `MyIntCollection` class as shown earlier in this section is self-explanatory, except perhaps for the final method:

```
    public double getAverage() {
        return ((double) total) / ((double) this.size());
    }
```

Since both `total` and `this.size()` are `int`(eger) expressions, we must explicitly cast at least one of them to be a `double` value before performing the division. If we were to simply return the result of `total/this.size()`, we'd be dividing an `int` by an `int`, which would cause the fractional part of the answer to be truncated.

Putting MyIntCollection to Work

Here's sample client code to demonstrate how our new `MyIntCollection` collection type can be put to good use:

```
public class MyIntCollectionExample {
    public static void main(String[] args) {
        // Instantiate one of our newly designed collections.
        MyIntCollection mic = new MyIntCollection();

        // Add four random integers to our "special" collection.
        mic.add(3);
        mic.add(6);
        mic.add(1);
        mic.add(9);
```

```
        // Take advantage of the size method as inherited from
        ArrayList ...
        System.out.println("The collection contains " + mic.size() +
            " int values");

        // ... and then ask mic "specialized" questions about its
        contents that a
        // garden-variety ArrayList couldn't easily answer.
        System.out.println("The smallest value is:  " + mic.
        getSmallestInt());
        System.out.println("The largest value is:  " + mic.
        getLargestInt());
        System.out.println("The average is:  " + mic.getAverage());
    }
}
```

Here's the output:

```
The collection contains 4 int values
The smallest value is:  1
The largest value is:  9
The average is:  4.75
```

Approach #3: Encapsulating a Standard Collection (MyIntCollection2)

Let's now take a look at an alternative way of inventing a custom collection class such as `MyIntCollection`. Instead of ***extending*** the `ArrayList` class as we did with `MyIntCollection`, we'll design a custom class to ***encapsulate*** an instance of an `ArrayList` collection.

We'll design a class called `MyIntCollection2` to illustrate this approach, using the code for `MyIntCollection` as a starting point; as you'll see, the differences between the two approaches are rather subtle:

- The first change would, of course, be to eliminate the `extends ArrayList` clause from the class declaration:

```
// We're no longer extending the ArrayList class.
public class MyIntCollection2 {
```

- Instead, we'll encapsulate an `ArrayList` as an ***attribute***

  ```
  ArrayList<Integer> numbers;
  ```

  along with retaining the other attributes that we declared for `MyIntCollection`: `smallestInt`, `largestInt`, and `total`.

- In the constructor for our new class, we'll instantiate the embedded `numbers ArrayList` whenever we instantiate `MyIntCollection2` as a whole:

  ```
  public MyIntCollection2() {
      // Instantiate the embedded ArrayList.
      numbers = new ArrayList<Integer>();

      // Initialize the total.
      total = 0;
  }
  ```

- Since we aren't extending the `ArrayList` class any longer, we won't inherit a `size` method automatically, and so we'll declare one of our own. Our `size` method will simply delegate the task of determining collection size to the embedded `numbers ArrayList`:

  ```
  // Since we don't INHERIT a size() method any longer, let's
     add one!
  public int size() {
      // DELEGATION!
      return numbers.size();
  }
  ```

- Recall that we overrode the `add` method in our `MyIntCollection` class to specialize its behavior as compared with the generic `ArrayList` version that we'd otherwise have inherited. Since we aren't extending the `ArrayList` class in designing `MyIntCollection2`, we won't inherit an `add` method, and so we'll declare one of our own. The code for this `add` method is virtually identical to that of the `MyIntCollection` class's version of `add`, except for two minor syntactical changes

that are necessary to delegate work to the encapsulated `numbers` collection—these changes are **bolded** in the following code:

```
// Since we don't INHERIT an add() method any longer, let's
   add one!
public boolean add(int i) {
    // Remember this int as the largest/smallest,
    // if appropriate.  (The FIRST time we add a value, it
       by default
    // will be BOTH the smallest AND the largest!)
    // DELEGATE to the embedded collection.
    if (numbers.isEmpty()) {
        smallestInt = i;
        largestInt = i;
    }
    else {
        if (i < smallestInt) smallestInt = i;
        if (i > largestInt) largestInt = i;
    }

  // Increase the total.
  total = total + i;

  // Add the int to the numbers collection.
  // DELEGATE to the embedded collection.
  return numbers.add(i);
}
```

- All remaining methods as declared for `MyIntCollection`—`getSmallestInt`, `getLargestInt`, and `getAverage`—remain unchanged for `MyIntCollection2`.

Here is the code for `MyIntCollection2` in its entirety—changes as compared with `MyIntCollection` are **bolded**:

```
import java.util.ArrayList;

// We're no longer extending the ArrayList class.
public class MyIntCollection2 {
```

```
    // Instead, we're encapsulating a ArrayList inside of this class.
    ArrayList<Integer> numbers;

    // We define a few extra attributes and methods beyond those that the
    // encapsulated ArrayList will provide -- the SAME attributes and methods
    // that we declared for the MyIntCollection class:
    private int smallestInt;
    private int largestInt;
    private int total;

    public MyIntCollection2() {
        // Instantiate the embedded ArrayList.
        numbers = new ArrayList<Integer>();

        // Initialize the total.
        total = 0;
    }

    // Since we don't INHERIT a size() method any longer, let's add one!
    public int size() {
        // DELEGATION!
        return numbers.size();
    }

    // Since we don't INHERIT an add() method any longer, we can't
    override it;
    // so, let's add one instead!
    public boolean add(int i) {
        // Remember this int as the largest/smallest,
        // if appropriate.  (The FIRST time we add a value, it by default
        // will be BOTH the smallest AND the largest!)
        // DELEGATE to the encapsulated collection.
        if (numbers.isEmpty()) {
            smallestInt = i;
            largestInt = i;
        }
        else {
            if (i < smallestInt) smallestInt = i;
```

```
            if (i > largestInt) largestInt = i;
        }

      // Increase the total.
      total = total + i;

      // Add the int to the numbers collection.
      // DELEGATE to the encapsulated collection.
      return numbers.add(i);
    }

    // All remaining methods are identical to those of MyIntCollection.

    public int getSmallestInt() {
        return smallestInt;
    }

    public int getLargestInt() {
        return largestInt;
    }

    public double getAverage() {
        return ((double) total)/this.size();
    }
}
```

Putting MyIntCollection2 to Work

The client code needed to manipulate this "flavor" of custom `int` collection is ***identical*** to the client code that we used to manipulate the first version of `MyIntCollection`—***a testimonial to the power of encapsulation***! The client code is repeated here, substituting all references to `MyIntCollection` with references to `MyIntCollection2`—but that's all that had to change!

```
public class MyIntCollection2Example {
  public static void main(String[] args) {
    // Instantiate one of our newly designed collections!
    MyIntCollection2 mic = new MyIntCollection2();
```

```
    // Add four random integers to our "special" collection.
    mic.add(3);
    mic.add(6);
    mic.add(1);
    mic.add(9);

    // Take advantage of the size method ...
    System.out.println("The collection contains " + mic.size() +
        " int values");

    // ... and then ask mic "specialized" questions about its contents.
    System.out.println("The smallest value is:  " + mic.getSmallestInt());
    System.out.println("The largest value is:  " + mic.getLargestInt());
    System.out.println("The average is:  " + mic.getAverage());
  }
}
```

The output would be the same as before.

Trade-Offs of Approach #2 vs. Approach #3

As illustrated with the `MyIntCollection` and `MyIntCollection2` examples, the coding effort required with either of the two approaches to creating a custom collection—extending a predefined collection class vs. encapsulating an instance of such a collection—is comparable. What are the advantages, then, of one approach vs. the other?

One advantage of extending a predefined collection class (approach #2) is that when we instantiate such a class at run time, we create only one object in memory—an instance of `MyIntCollection`, which is ***simultaneously*** an `ArrayList` by virtue of inheritance. By comparison, when we create an instance of `MyIntCollection2`, we wind up creating two objects, as illustrated in Figure 6-16. Thus, approach #2 is a bit more economical in terms of memory usage.

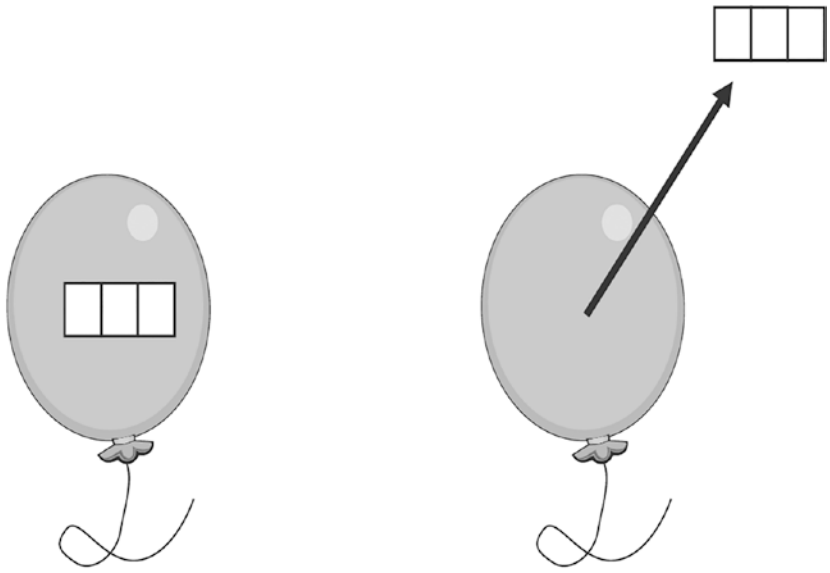


Figure 6-16. *Instantiating one vs. two objects at run time*

Alternatively, an advantage of encapsulating a predefined collection class instance (approach #3) is that we can choose to expose as few of the encapsulated collection's public behaviors as we wish to our client code.

- `MyIntCollection`, as an `ArrayList`, inherits all 30 `public` behaviors of the `ArrayList` class. Even if we see relevance only in the `size` and `add` methods of `MyIntCollection`, the other 28 methods are exposed to/accessible by client code, as well.
- In contrast, MyIntCollection2 does not define those 28 methods.
- Based on the way that we designed the `MyIntCollection2` class, on the other hand, it exposes only ***two*** of these public behaviors—`size` and `add`—thus simplifying the task of using our class from the perspective of client code. Furthermore, if we wanted to, we could ***disguise*** these methods by giving them entirely different names in `MyIntCollection2`, as follows:

```
public class MyIntCollection2 {
    // details omitted ...

    // This was formerly the size() method ...
    public int getIntCount() {
        // DELEGATION!
        return numbers.size();
    }

    // This was formerly the add() method ...
    public boolean insertAnInt(int i) {
        // Remember this int as the largest/smallest,
        // if appropriate.  (The FIRST time we add a value, it
           by default
        // will be BOTH the smallest AND the largest!)
        // DELEGATE to the encapsulated collection.
        if (numbers.isEmpty()) { ...

    // etc.
```

This is thus taking full advantage of the power of encapsulation and information hiding.

One significant advantage to approach #2 is that as a true ArrayList by virtue of inheritance, your custom collection type may be used anywhere within the Java language that a conventional ArrayList is permitted to be used.

The bottom line is that either approach #2 or approach #3 has both advantages and disadvantages. By understanding the subtle differences between the two, you'll be able to choose between them on a case-by-case basis.

Collections As Method Return Types

Collections provide a way to overcome the limitation noted in Chapter 4 about methods being able to return only a single result. If we define a method as having a return type that is a ***collection*** type, we can return an arbitrarily sized collection of object references to client code.

In the following code snippet for the `Course` class, a `getRegisteredStudents` method is provided to enable client code to request a reference to the entire collection of `Student` objects that are registered for a particular course:

```
public class Course {
    private ArrayList<Student> enrolledStudents;

    // Details omitted ...

    // The following method returns a reference to an entire collection
    // containing however many Students are registered for the Course in
    question.
    public ArrayList<Student> getRegisteredStudents() {
        return enrolledStudents;
    }

    // etc.
```

Here's an example of how client code might then use such a method:

```
// Instantiate a course and several students.
Course c = new Course();
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();

// Enroll the students in the course.
c.enroll(s1);
c.enroll(s2);
c.enroll(s3);

// Now, ask the course to give us a handle on the collection of
// all of its registered students and iterate through the collection,
// printing out a grade report for each student.
for (Student s : c.getRegisteredStudents()) {
    s.printGradeReport();
}
```

Note the use of a nested method call in the `for` statement; since `c.getRegisteredStudents()` is an expression of type `ArrayList`, this expression can be used in the `for` statement to designate the collection that we wish to iterate through.

In Chapter 7, when we discuss interfaces in general and the `Collection` interface in particular, we'll look at an alternative way of returning a collection from a method that makes our code more versatile.

Collections of Derived Types

As mentioned previously, arrays, as simple collections, contain items (either primitive values or object references) that are all of the same type: all `int`(egers), for example, or all (references to) `Student` objects. As it turns out, regardless of what type of collection we're using, we'll typically want to constrain it to contain similarly typed objects, for reasons that we'll explore in Chapter 7 when we discuss ***polymorphism***. However, the power of inheritance steps in to make collections quite flexible in terms of what they contain.

It turns out that if we declare a collection to hold objects of a given superclass—for example, `Person`—then we're free to insert objects explicitly declared to be of type `Person` ***or of any type derived from*** `Person`—for example, `UndergraduateStudent`, `GraduateStudent`, and `Professor`. This is due to the "is a" nature of inheritance; `UndergraduateStudent`, `GraduateStudent`, and `Professor` objects, as subclasses of `Person`, are simply special cases of `Person` objects. The Java compiler would therefore be perfectly happy to see code such as the following

```
Person[] people = new Person[100];

Professor p = new Professor();
UndergraduateStudent s1 = new UndergraduateStudent();
GraduateStudent s2 = new GraduateStudent();

// Add a mixture of professors and students in random order to the array.

people[0] = s1;
people[1] = p;
people[2] = s2;
// etc.
```

or for an ArrayList

```
ArrayList<Person> people = new ArrayList<>();

Professor p = new Professor();
UndergraduateStudent s1 = new UndergraduateStudent();
GraduateStudent s2 = new GraduateStudent();

// Add a mixture of professors and students in random order to the
ArrayList.

people.add(s1);
people.add(p);
people.add(s2);
// etc.
```

Revisiting Our Student Class Design

You may recall that when we talked about the attributes of the Student class back in Chapter 3, we held off on assigning types to the courseLoad and transcript attributes, as shown in Table 6-1.

Table 6-1. *Proposed Data Structure for the Student Class*

Attribute Name	Data Type
name	String
studentID	String
birthDate	Date
address	String
major	String
gpa	double
advisor	Professor
courseLoad	**???**
transcript	**???**

Armed with what we now know about collections, we can now complete our `Student` class design.

The courseLoad Attribute of Student

The `courseLoad` attribute is meant to represent a list of all `Course` objects that the `Student` is presently enrolled in. So it makes perfect sense that this attribute be declared as simply a standard collection of `Course` object references—an `ArrayList`, perhaps:

```
import java.util.ArrayList;

public class Student {
  private String name;
  private String studentId;
  private ArrayList<Course> courseLoad;
  // etc.
```

The transcript Attribute of Student

The `transcript` attribute is a bit more challenging. What is a transcript, in real-world terms? It's a report of all of the courses that a student has taken since they were first admitted to this university, along with the semester in which each course was taken, the number of credit hours that each course was worth, and the letter grade that the student received for the course. A typical transcript entry, when printed, might look as follows:

```
CS101      Beginning Objects        3.0         A
```

If we think of each line item on a printed transcript as an ***object***, we can declare a `TranscriptEntry` class to describe them, as follows:

```
public class TranscriptEntry {
    // One TranscriptEntry object represents a single line item on
       a printed
    // transcript.
    private Course courseTaken;
    private String semesterTaken;  // e.g., "Fall 2006"
    private String gradeReceived;  // e.g., "B+"
```

```
    // Constructor.
    public TranscriptEntry(Course c, String semester, String grade) {
        // Details omitted ...
    }

    // Accessor method details omitted ...

    public void printTranscriptEntry() {
        // We "talk to" the courseTaken object/attribute to obtain the
        // majority of the required information (an example of
        // delegation). Reminder:  \t is a tab character.
        System.out.println(
            this.getCourseTaken().getCourseNo() + "\t" +
            this.getCourseTaken().getTitle() + "\t" +
            this.getCourseTaken().getCreditHours() + "\t" +
            this.getGradeReceived());
  }

  // Other methods TBD ...
}
```

Since each `TranscriptEntry` object maintains a handle on a `Course` object, the `TranscriptEntry` object can avail itself of the `Course` object's course number, title, or credit hour value (needed for computing the GPA)—all privately encapsulated in the `Course` object as attributes—by calling the appropriate accessor methods on that `Course` object as needed.

Back in the `Student` class, we can now define the `Student`'s `transcript` attribute to be a ***collection*** of `TranscriptEntry` objects:

```
import java.util.*;

public class Student {
    private String name;
    private String studentId;
    private ArrayList<TranscriptEntry> transcript;
    // etc.
```

We can then equip the `Student` class with an `addTranscriptEntry` method for use in inserting a new `TranscriptEntry` into the `transcript` collection

```
    public void addTranscriptEntry(TranscriptEntry te) {
        // Store the TranscriptEntry in our ArrayList.
        transcript.add(te);
    }
```

along with a `printTranscript` method for iterating through this collection:

```
    // This method merely iterates through the collection,
    // delegating the work of printing to the individual
    // TranscriptEntry objects.
    public void printTranscript() {
        // Print header information on the transcript:
        // Student's name, name of the university, date
        // printed, etc.
        System.out.println("Academic transcript for " +
            this.getName());
        // Other transcript header details omitted ...

        // Print individual transcript line items.
        for (TranscriptEntry t : transcript) {
            t.printTranscriptEntry();
        }
    }

    // etc.
}
```

Figure 6-17 illustrates how `Student`, `ArrayList`, `TranscriptEntry`, and `Course` objects would thus be "wired together" in memory at run time.

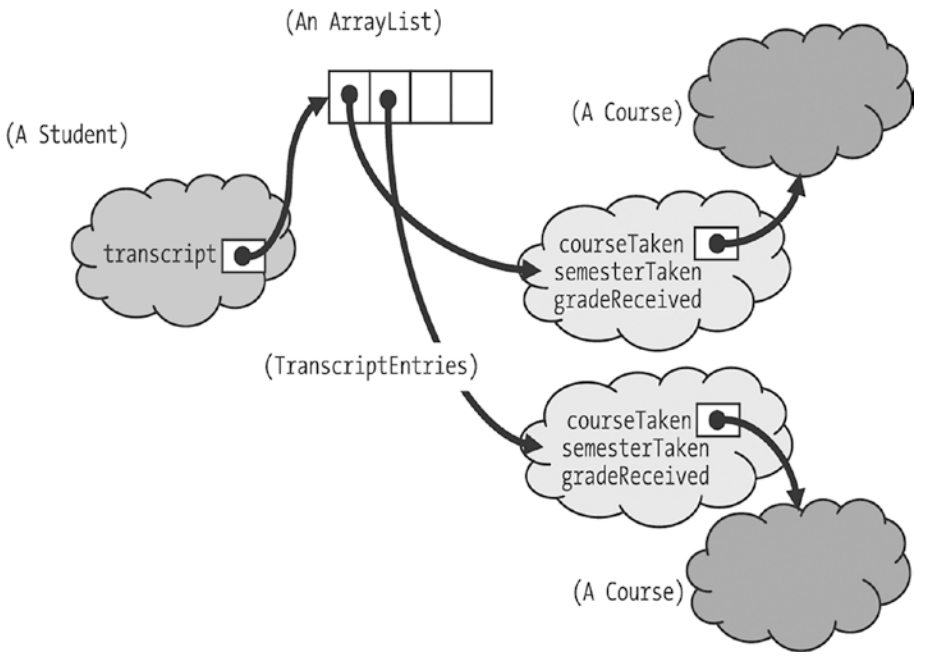


Figure 6-17. *As "wired together" in memory, a* `Student` *references an* `ArrayList`*, which in turn references* `TranscriptEntry` *objects. These in turn each reference a* `Course` *object*

Finally, let's look at the client code that might be involved in putting these classes to work:

```
Student s = new Student("1234567", "James Huddleston");
Course c = new Course("LANG 800", "Advanced Language Studies");
s.registerForCourse(c);

// Time passes ... details omitted.

// Semester is finished!  Assign a grade to this student (he's brilliant!).
TranscriptEntry te = new TranscriptEntry(c, "Spring 2006", "A+");
s.addTranscriptEntry(te);

// Additional grades assigned for other courses ... details omitted.

s.printTranscript();
```

The manner in which we're assigning a grade to a student for a course that they have completed (namely, by instantiating a `TranscriptEntry` object and then calling the `Student`'s `addTranscriptEntry` method)

```
TranscriptEntry te = new TranscriptEntry(c, "Spring 2006", "A+");
s.addTranscriptEntry(te);
```

is not as intuitive as it could be to someone reading this client code. Let's see if we can improve upon our design with a goal of rendering client code that is a bit more straightforward.

The transcript Attribute, Take 2

We'll add a bit more sophistication to our abstraction by declaring a class called `Transcript` to encapsulate a standard type of collection, the technique that we employed when creating the `MyIntCollection2` class earlier in this chapter:

```
public class Transcript {
    // The Transcript class ENCAPSULATES a garden variety ArrayList
    // of TranscriptEntry references.
    private ArrayList<TranscriptEntry> transcriptEntries;

    // Maintain a handle on the Student to whom this
    // transcript belongs.
    Student owner;

    // Constructor/accessor details omitted.

    // Rather than having client code manufacture a TranscriptEntry object
    // to pass in as an argument, we'll "disguise" what we are doing a bit.
    public void courseCompleted(Course c, String semester, String grade) {
        // Instantiate and insert a brand-new TranscriptEntry object
           into the
        // ArrayList - details hidden away!
        transcriptEntries.add(new TranscriptEntry(c, semester, grade));
    }

    // We've transferred the logic of the Student class's printTranscript
    // method into the Transcript class's print method.
```

```
    public void print() {
        for (TranscriptEntry te : transcript) {
            te.printTranscriptEntry();
        }
    }

    // etc.
  }
```

Of particular note is the fact that we've effectively hidden our use of `TranscriptEntry` objects from client code by providing a `courseCompleted` method. This method accepts the "raw materials" necessary to create a `TranscriptEntry` object—namely, a `Course` reference plus `String`s representing the semester in which the course was completed and the grade received—and ***invokes the*** `TranscriptEntry` ***constructor from within the privacy of the*** `courseCompleted` ***method body***. As you'll see shortly, this relieves client code from having to deal with the `TranscriptEntry` class; `TranscriptEntry` is now strictly a "helper" class that exists to serve the `Transcript` class behind the scenes.

Chapter 13 introduces the notion of **inner classes**, a construct used to "bury" the declaration of one class, such as `TranscriptEntry`, wholly within another so that it truly is a ***private type***.

We'll now go back to the `Student` class and change our declaration of the `transcript` attribute from an `ArrayList` to a `Transcript`:

```
public class Student {
  private String name;
  private String studentId;
  // This used to be declared as an ArrayList.
  private Transcript transcript;
  // etc.
```

We can in turn simplify the `printTranscript` method of the `Student` class, to take advantage of delegation—it's now a one-liner!

```
  public void printTranscript() {
    // We now DELEGATE the work of printing all entries to
    // the Transcript itself!
    transcript.print();
  }

  // etc.
```

Finally, let's look at the client code that might be involved in putting these ***new and improved*** classes to work. I've repeated the client code example used before, **bolding** the subset of client code that has changed as a result of our improved design:

```
Student s = new Student("1234567", "James Huddleston");
Course c = new Course("LANG 800", "Advanced Language Studies");
s.registerForCourse(c);

// Time passes ... details omitted.

// Semester is finished!  Assign a grade to this student (he's brilliant!).
// It's now accomplished as a single line of arguably more intuitive code.
s.courseCompleted(c, "Spring 2006", "A+");

// Additional grades assigned for other courses ... details omitted.

s.printTranscript();
```

The manner in which we're assigning a grade to a student for a course that they have completed—namely, by calling the `courseCompleted` method of `Student`—is arguably much clearer and more self-documenting to anyone reading this client code than the previous version of client code. Here's the code before:

```
TranscriptEntry te = new TranscriptEntry(c, "Fall 2006", "B+");
s.addTranscriptEntry(te);
```

And here's the code after:

```
s.courseCompleted(c, "Spring 2006", "A+");
```

This "Take 2" approach of introducing ***two*** new classes/abstractions—`TranscriptEntry` and `Transcript`—is a bit more sophisticated than the first approach, where we only introduced `TranscriptEntry` as an abstraction.

- We've simplified the `Student` class considerably. `Student` code needn't be complicated by the details of ***how*** transcripts are represented or managed internally or even that there is such a thing as a `TranscriptEntry` object—those details are hidden inside of the `Transcript` class, as they should be.
- ***More significantly, we've simplified our client code***. The `Student` class need be designed and coded only once, but ***client code*** written to ***manipulate*** `Student` objects will potentially occur in countless places across numerous applications.

Whenever possible, it's desirable to bury implementation details inside of a class rather than exposing client code to such details; this lessens the burden on developers/maintainers of client code by lessening the likelihood of logic errors in such code.

Figure 6-18 illustrates how `Student`, `Transcript`, `ArrayList`, `TranscriptEntry`, and `Course` objects would be "wired together" at run time, and Table 6-2 provides a side-by-side comparison of the code used in our two "takes" on representing the notion of student transcripts.

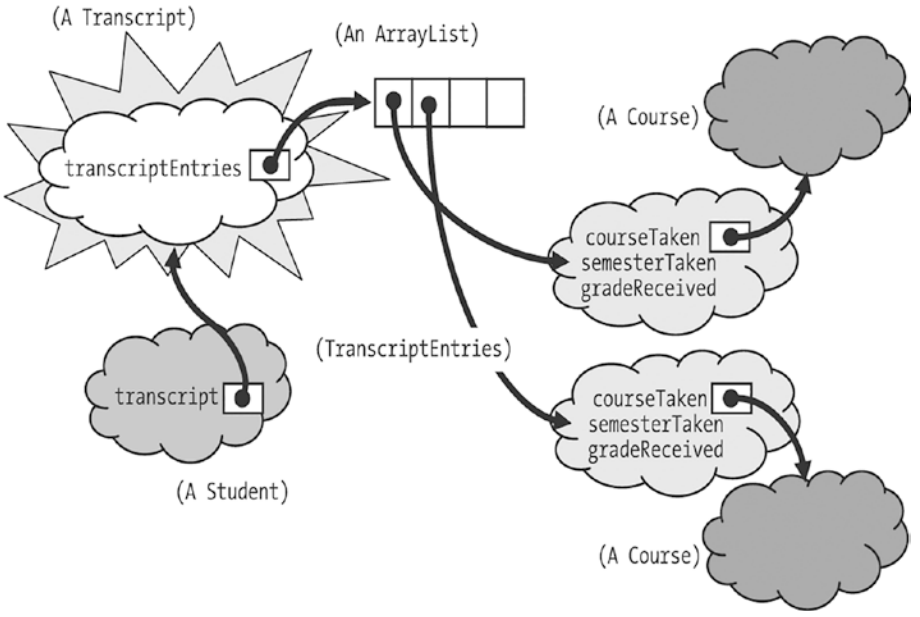


Figure 6-18. *Introducing another level of abstraction in the form of a* `Transcript` *class ultimately simplifies client code, which is an important design goal*

Table 6-2. *Comparing the "Take 1" and "Take 2" Code Versions*

Code for "Take 1"	Code for "Take 2"
The* `TranscriptEntry` *Class	

`public class TranscriptEntry {`
`  private Course courseTaken;`
`  private String semesterTaken;`
`  private String gradeReceived;`
 	
`  // Details omitted ...`
 	
`  // Constructor.`
`  public TranscriptEntry(Course c,`
`   String semester, String grade) {`
`   // Details omitted ...`
`  }`
 	
`  public void printTranscriptEntry()`
`{`
`   System.out.println((`
`    courseTaken.getCourseNo() +`
`    "\t" +`
`    courseTaken.getTitle() +`
`    "\t" +`
`    courseTaken.getCreditHours() +`
`    "\t" +`
`    getGradeReceived());`
`  }`
 	
`  // etc.`
`}`
	(The* `TranscriptEntry` ***class code for "Take 2" is the same as for "Take 1.")

(*continued*)

Table 6-2. (*continued*)

Code for "Take 1"	Code for "Take 2"
	The `Transcript` ***Class***

("Take 1" did not involve the `Transcript` ***class.)***	`public class Transcript {`
	`  private ArrayList<TranscriptEntry>`
	`  transcriptEntries;`
	
	`  // Details omitted ...`
	
	`  public void courseCompleted(Course c,`
	`  String semester, String grade) {`
	`  transcriptEntries.add(`
	`  new TranscriptEntry(c,`
	`  semester, grade);`
	`  }`
	
	`  public void print() {`
	`  print header info. ...`
	`  for (TranscriptEntry te :`
	`  transcript) {`
	`  te.printTranscriptEntry();`
	`  }`
	`  }`
	
	`  // etc.`
	`}`

(*continued*)

Table 6-2. (*continued*)

Code for "Take 1"

The `Student` ***Class***

```
import java.util.ArrayList;

public class Student {
 private String name;
 private String studentId;
```

Here, we use an `ArrayList`.

```
 private ArrayList<TranscriptEntry>
  transcript;

 // etc.
```

Client code has to be aware of the notion of a `TranscriptEntry`.

```
 public void addTranscriptEntry(
  TranscriptEntry te) {
  transcript.add(te);
 }

 public void printTranscript() {
  print header information ...
  for (TranscriptEntry t :
  transcript) {
   t.printTranscriptEntry();
  }
 }

 // etc.
}
```

Code for "Take 2"

```
public class Student {
 private String name;
 private String studentId;
```

Here, we use a `Transcript`.

```
private Transcript transcript;

// etc.
```

This method hides more "gory details" and is hence easier for client code to use. But it serves the same purpose as the `addTranscriptEntry` ***method in "Take 1."***

```
 public void courseCompleted(Course c,
  String semester, String grade) {
   Transcript.courseCompleted(
    c, semester, grade);
 }

 public void printTranscript() {
  // Delegation !
  transcript.print();
 }

 // etc.

}
```

(*continued*)

Table 6-2. (*continued*)

Code for "Take 1"	Code for "Take 2"
	Sample Client Code

`Student s = new Student(...);`
`Course c = new Course(...);`
`s.registerForCourse(c);`
`// etc.`
	`Student s = new Student(...);`
	`Course c = new Course(...);`
	`s.registerForCourse(c);`
	`// etc.`

Client code is somewhat "ugly."	***Client code is more streamlined and intuitive!***
`TranscriptEntry te =` **`  new TranscriptEntry(c,`** **`  "Fall 2006", "B+");`** **`s.addTranscriptEntry(te);`** `s.printTranscript();`	**`s.courseCompleted(c, "Fall 2006",`** **`"B+");`** `s.printTranscript();`

Our Completed Student Data Structure

Table 6-3 illustrates how we've taken full advantage of collections to round out our `Student` class definition.

Table 6-3. *Rounding Out the `Student` Class's Data Structure with Collections*

Attribute Name	Data Type
`name`	`String`
`studentID`	`String`
`birthDate`	`Date`
`address`	`String`
`major`	`String`
`gpa`	`Double`
`advisor`	`Professor`
`courseLoad`	**`ArrayList<Course>`**
`transcript`	**`Transcript`**

Summary

In this chapter, you've learned

- Collections are special types of objects used to gather up and manage references to other objects.
- Most OO languages support three generic types of collection:
  - Ordered lists
  - Sets
  - Dictionaries (a.k.a. maps)
- Arrays are a simple type of collection that have some limitations, but we also have other more powerful collection types to draw upon with OO languages, such as Java's `ArrayLists`, `TreeMaps`, etc.
- It's important to familiarize yourself with the unique characteristics of whatever collection types are available for a particular OO language, so as to make the most informed selection of which collection type to use for a particular circumstance.
- You can invent your own collection types by either ***extending*** predefined collection classes or creating "wrapper classes" to ***encapsulate*** an instance of a predefined collection class, as well as the subtle differences between the two approaches.
- You can work around the limitation that a method can return only one result by having that result be a collection.
- You can create very sophisticated composite classes through the use of collections as attributes.
- "Burying" increasing levels of detail within layers of abstraction serves to simplify client code.

There's a bit more to appreciate about collections, but we must first cover some additional Java topics. We'll revisit collections in Chapter 7.

EXERCISES

1. Given the following abstraction

A book is a collection of chapters, each of which is a collection of pages.

sketch out the code for the `Book`, `Chapter`, and `Page` classes.

   - Invent whatever attributes you think would be relevant, taking advantage of collections as attributes where appropriate.
   - Include methods in the `Chapter` class for adding pages and for determining how many pages a chapter contains.
   - Include methods in the `Book` class for adding chapters, for determining how many chapters the book contains, for determining how many pages the book contains (hint: use delegation!), and for printing out a book's table of contents.

2. [*Coding*] Code the `Book`, `Chapter`, and `Page` classes that you specified in Exercise 1, and write a simple driver program to put them through their paces.
3. What generic type(s) of collection(s)—ordered list, sorted ordered list, set, dictionary—might you use to represent each of the following abstractions? Explain your choices.

   - A computer parts catalog
   - A poker hand
   - Trouble calls logged by a technical help desk

4. What collections do you think would be important to maintain for the SRS, based on the requirements presented in the Introduction to this book?
5. What collections do you think would be important to maintain for the Prescription Tracking System (PTS) described in the Appendix?
6. What collections do you think would be important to maintain for the problem area that you described for Exercise 3 in Chapter 1?

7. [*Coding*] Modify the `MyIntCollection` class as presented in this chapter to add a method called `printSortedContents` that, when invoked, prints the contents of the collection in sorted order. You may make whatever changes you wish to the private details of the class in accommodating this new behavior.

Then, modify the `MyIntCollection2` version of the class to accommodate a `printSortedContents` method, as well.

Was accommodating this new requirement significantly easier with one version of the custom collection than with the other?

CHAPTER 7

Some Final Object Concepts

By now, you've hopefully gained a solid appreciation for how powerful object-oriented languages are for modeling complex real-world situations. By way of review

- We can create our own user-defined types, also known as classes, to model objects of arbitrary complexity, as we discussed in Chapter 3.
- We can arrange these types into class hierarchies to take advantage of the inheritance mechanism of OO languages, as we discussed in Chapter 5.
- Through encapsulation and information hiding, we can shield client code from changes that we make to the private implementation details of our classes, making objects responsible for ensuring the integrity of their own data, as we discussed in Chapter 4.
- We can design relationships between classes into their very "bone structure" so that collaborating objects can be linked together in memory at run time, as we discussed in Chapter 5.
- Classes can model the most complex of real-world concepts, particularly when we take advantage of collections, as we did when modeling the `transcript` attribute of the `Student` class in Chapter 6.

You might wonder how there could possibly be anything left in our OO bag of tricks! However, as powerful as all of the preceding OO language features are, there are still a few more important features of objects to be discussed.

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_7

In this chapter, you'll learn

- How a ***single*** line of code, representing a message—for example, `x.foo();`—can exhibit a variety of behaviors at run time
- How we can specify ***what*** an object's mission should be without going to the trouble of specifying the details of ***how*** the object is to carry out that mission and also under what circumstances we'd want to be able to do so
- How an object can have a "split personality" by exhibiting the behaviors of two or more different types of object
- Creative ways for an entire class of objects to easily and efficiently share data without breaking the spirit of encapsulation
- How features can be defined that are associated with a class as a whole rather than with an instance of a class and how we can take advantage of this capability to design ***utility classes***
- How we may declare variables whose values, once assigned, remain constant while an application is executing

Polymorphism

The term **polymorphism** refers to the ability of two or more objects belonging to ***different*** classes to respond to exactly the ***same*** message (method call) in different class-specific ways.

As an example, if we were to instruct three different people—a surgeon, a hair stylist, and an actor—to "Cut!" then

- The surgeon would begin to make an incision.
- The hair stylist would begin to cut someone's hair.
- The actor would abruptly stop acting out the current scene, awaiting directorial guidance.

These three different professionals may be thought of as `Person` objects belonging to different professional subclasses: `Surgeon`, `HairStylist`, and `Actor`. Each was given the same message—"Cut!"—but carried out the operation differently as prescribed by the subclass that each belongs to.

Let's now turn to a software example relevant to the SRS. Assume that we've defined a `Student` superclass and two subclasses, `GraduateStudent` and `UndergraduateStudent`. In Chapter 5, we discussed the fact that a `print` method designed to print the values of all of a `Student`'s attributes wouldn't necessarily suffice for printing the attribute values for a `GraduateStudent`, because the code as written for the `Student` superclass wouldn't know about any attributes that may have been added to the `GraduateStudent` subclass. We then looked at how to ***override*** the `print` method of `Student` to create specialized versions of the method for all of its subclasses. The syntax for doing so, which was first introduced in Chapter 5 with the `GraduateStudent` class, is repeated again here for your review. I've added the `UndergraduateStudent` class code, as well:

```
//-------------
// Student.java
//-------------

public class Student {
    private String name;
    private String studentId;
    private String major;
    private double gpa;

    // Public get/set methods would also be provided (details omitted) ...

    public void print() {
        // We can print only the attributes that the Student class
        // knows about.
        System.out.println("Student Name:  " + getName() + "\n" +
            "Student No.:  " +  getStudentId()  + "\n" +
            "Major Field:  " +  getMajor()  + "\n" +
            "GPA:  " + getGpa());
    }
}

//---------------------
// GraduateStudent.java
//---------------------
```

```
public class GraduateStudent extends Student {
    // Adding several attributes.
    private String undergraduateDegree;
    private String undergraduateInstitution;

    // Public get/set methods would also be provided (details omitted) ...

    // Overriding the print method.
    public void print() {
        // Reuse code from the Student superclass ...
        super.print();

        // ... and then go on to print this subclass's specific attributes.
        System.out.println("Undergrad. Deg.: " + getUndergraduateDegree() +
            "\n" + "Undergrad. Inst.:  " +
            getUndergraduateInstitution() + "\n" +
            "THIS IS A GRADUATE STUDENT ...");
    }
}

//--------------------------
// UndergraduateStudent.java
//--------------------------

public class UndergraduateStudent extends Student {
    // Adding an attribute.
    private String highSchool;

    // Public get/set methods would also be provided (details omitted) ...

    // Overriding the print method.
    public void print() {
        // Reuse code from the Student superclass ...
        super.print();

        // ... and then go on to print this subclass's specific attributes.
        System.out.println("High School Attended:  " + getHighSchool() +
            "\n" + "THIS IS AN UNDERGRADUATE STUDENT ...");
    }
}
```

In our main SRS application, we'll declare an `ArrayList` called `studentBody` to hold references to `Student` objects. We'll then populate the `ArrayList` with `Student` object references—some `GraduateStudents` and some `UndergraduateStudents`, randomly mixed—as shown here:

```
// Declare and instantiate an ArrayList of Students.
ArrayList<Student> studentBody = new ArrayList<>();

// Instantiate various types of Student object.
UndergraduateStudent u1 = new UndergraduateStudent();
UndergraduateStudent u2 = new UndergraduateStudent();
GraduateStudent g1 = new GraduateStudent();
GraduateStudent g2 = new GraduateStudent();
// etc.

// Insert them into the ArrayList in random order.
studentBody.add(u1);
studentBody.add(g1);
studentBody.add(g2);
studentBody.add(u2);
// etc.
```

Since we're storing both `GraduateStudent` and `UndergraduateStudent` objects in this `ArrayList`, we've declared the `ArrayList` to be of a base type common to all objects that the collection is intended to contain, namely, `Student`. By virtue of the "is a" nature of inheritance, an `UndergraduateStudent` object ***is a*** `Student`, and a `GraduateStudent` object ***is a*** `Student`, and so the compiler won't complain when we insert either type of object into this `ArrayList`.

Note that the compiler ***would*** object, however, if we tried to insert a `Professor` object into the same `ArrayList`, because a `Professor` isn't a `Student`, at least not in terms of the class hierarchy that we've defined for the SRS. If we wanted to include `Professors` in our `ArrayList` along with various types of `Students`, we'd have to declare the `ArrayList` as holding a base type common to ***both*** the `Student` and `Professor` classes, namely, `Person` (or `Object`, which as we discussed in Chapter 5 is the implied superclass for all inheritance hierarchies in Java).

Perhaps we'd like to print the attribute values of all of the Students in our studentBody collection. We'd want each Student object—whether it's a GraduateStudent or an UndergraduateStudent instance—to use the version of the print method appropriate for its (sub)class. The following code will accomplish this nicely:

```
// Step through the ArrayList (collection) ...
for (Student s : studentBody) {
  // ... invoking the print method of each Student object.
  s.print();
}
```

Variable s is declared in the for statement to be a reference to a ***generic*** Student object, because that's the type of reference that we declared studentBody to hold. As we step through this collection of Student objects at ***run time***, however, each object will ***automatically*** know which version of the print method it should execute, based on its own internal knowledge of the ***specific*** type/(sub)class that it belongs to (GraduateStudent vs. UndergraduateStudent, in this example). We'd wind up with a report similar to the following, where the **bolded** lines emphasize the differences in output between the GraduateStudent and UndergraduateStudent versions of the print method:

```
Student Name:  John Smith
Student No.:  12345
Major Field:  Biology
GPA:  2.7
High School Attended:  Rocky Mountain High
THIS IS AN UNDERGRADUATE STUDENT ...

Student Name:  Paula Green
Student No.:  34567
Major Field:  Education
GPA:  3.6
Undergrad. Deg.:  B.A. English
Undergrad. Inst.:  UCLA
THIS IS A GRADUATE STUDENT ...

Student Name:  Dinesh Prabhu
Student No.:  98765
```

```
Major Field:  Computer Science
GPA:  4.0
Undergrad. Deg.:  B.S. Computer Engineering
Undergrad. Inst.:  Case Western Reserve University
THIS IS A GRADUATE STUDENT ...

Student Name:  Jose Rodriguez
Student No.:  82640
Major Field:  Math
GPA:  3.1
High School Attended:  James Ford Rhodes High
THIS IS AN UNDERGRADUATE STUDENT ...
```

The term ***polymorphism*** is defined in Merriam-Webster's Dictionary as "the quality or state of being able to assume different forms." The line of code

```
s.print();
```

in the preceding example is said to be ***polymorphic*** because the logic performed in response to the message can take many different forms, depending on the class identity of the object at run time.

Of course, this approach of iterating through a collection to ask each object, one by one, to do something in its own class-specific way won't work unless all objects in the collection understand the message being sent. That is, all objects in the `studentBody` collection ***must*** have defined a method with the signature `print()`. However, we've ***guaranteed*** that every object in the `studentBody` collection at run time ***will*** have such a method, as follows:

- First of all, we declared the `studentBody ArrayList` to hold objects of type `Student` (or subclasses thereof), so the compiler will therefore not allow us to put non-`Student` object references into the `ArrayList`. That is, any attempt to add a non-`Student` reference to the `studentBody` collection will be rejected at compile time:

  ```
  ArrayList<Student> studentBody = new ArrayList<>();
  Professor p = new Professor();

  // This next line won't compile.
  studentBody.add(p);
  ```

Here's the compilation error:

```
cannot find symbol
symbol: method add(Professor)
location: class java.util.ArrayList<Student>
```

- Second, we provided the `Student` superclass with a parameterless `print` method. Had we not done so, then the Java compiler would have objected to the line of code contained within the `for` loop:

```
// Step through the ArrayList (collection) ...
for (Student s : studentBody) {
    // This next line won't compile if the Student class doesn't
       define a
    // method with the signature "print()".
    s.print();
}
```

Here's the compilation error:

```
cannot find symbol
symbol: method print()
location: class Student
```

This error arises because the compiler checks the `Student` class (of which `s` is declared to be a member) for the presence of a parameterless `print` method. At ***run time***, `s` might actually be referring to a generic `Student` or to a `GraduateStudent` or to an `UndergraduateStudent` or to any other type derived from the `Student` class; however, the compiler has no way of predicting at ***compile time*** what the true ***run-time*** type of the object referred to by `s` will be (the compiler doesn't have a crystal ball at its disposal), and so it will make a go–no go decision based on how a generic `Student` is defined.

Finally, by virtue of inheritance, any ***subclass*** of `Student` is guaranteed to either ***inherit*** the `Student`'s version of the parameterless `print` method or to optionally ***override*** it with one of its own. Either way, all classes derived from `Student` will have such a method.

The bottom line is that all objects inserted into the `studentBody ArrayList` are ***guaranteed*** to be "`print` savvy" at run time.

Reflecting for a moment, you can now see that you had previously learned about everything that's needed to facilitate polymorphism in a programming language before this discussion of polymorphism even began. ***Inheritance combined with overriding facilitates polymorphism***.

Polymorphism Simplifies Code Maintenance

To appreciate the power of polymorphism, let's look at how we might have to approach this same challenge—handling different objects in different type-specific ways—with a programming language that ***doesn't*** support polymorphism.

In the absence of polymorphism, we'd typically handle scenarios having to do with a variety of kinds of students using a series of `if` tests:

```
for (Student s : studentBody) {
    // Process the next student.
    // Pseudocode.
    if (s is an undergraduate student)
        s.printAsUndergraduateStudent();
    else if (s is a graduate student)
        s.printAsGraduateStudent();
    else if ...
}
```

As the number of cases grows, so too does the "spaghetti" nature of the resultant code. And keep in mind that this sort of `if` test would arise in ***countless*** places throughout an application, namely, wherever we are iterating through a collection declared to hold `Students` of various types.

Let's now contrast this with our ***polymorphic*** iteration through the `studentBody` collection:

```
  // Step through the ArrayList (collection) ...
  for (Student s : studentBody) {
      // ... invoking the print method of the next Student object.
      s.print(); // polymorphism at work!
}
```

Thanks to polymorphism, a single line of code—`s.print();`—can handle all types of `Students`, thus making our code much more concise. Better still, polymorphic client code is ***robust to change***. For example, let's say that, long after our SRS application has been coded, tested, and deployed, we derive classes called `PhDStudent` and `MastersStudent` from `GraduateStudent`, each of which in turn overrides the `print` method of `GraduateStudent` to provide its own "flavor" of print functionality. We're now free to randomly insert `MastersStudents` and `PhDStudents` into our `studentBody` collection, along with `GraduateStudents` and `UndergraduateStudents`, and our polymorphic code for iterating through the collection ***doesn't have to change***! The following code illustrates this:

```
// Declare and instantiate an ArrayList.
ArrayList<Student> studentBody = new ArrayList<>;

// Instantiate various types of Student object.  We're now dealing
with FOUR
// different derived types!
UndergraduateStudent u1 = new UndergraduateStudent();
PhDStudent p1 = new PhDStudent();
GraduateStudent g1 = new GraduateStudent();
MastersStudent m1 = new MastersStudent();
// etc.

// Insert them into the ArrayList in random order.
studentBody.add(u1);
studentBody.add(p1);
studentBody.add(g1);
studentBody.add(m1);
// etc.

// Then, later in our application ...

// This is the EXACT SAME CODE that we've seen before!
// Step through the ArrayList (collection) ...
for (Student s : studentBody) {
    // ... and invoke the print method of the next Student object.
    // Because of the polymorphic nature of Java, this next line
    didn't require
```

```
    // any changes!
    s.print();
}
```

The `for` loop in our client code didn't have to change to accommodate the new subclasses—`MastersStudent` and `PhDStudent`—because, as subclasses of `Student`, these new types of object are once again ***guaranteed*** to understand the ***same*** `print` message by virtue of inheritance plus (optional) overriding.

The story is quite different, however, with the nonpolymorphic example that we crafted earlier. That version of client code would indeed have to change to accommodate these new student types; specifically, we'd have to hunt through our application to find every situation where we were trying to differentiate among the various subclasses of `Student` and complicate our `if` tests even further by adding additional cases as follows

```
for (Student s : studentBody) {
    // Process each student.
    // Pseudocode.
    if (s is an undergraduate student)
        s.printAsUndergraduateStudent();
    else if (s is a Masters student)
        s.printAsMastersStudent();
    else if (s is a PhD student)
        s.printAsPhDStudent();
    else if (s is a generic graduate student)
        s.printAsGraduateStudent();
    else if ...
}
```

causing the "spaghetti piles" to grow ever taller. Maintenance of nonpolymorphic applications quickly becomes a ***nightmare***!

As we saw with encapsulation and information hiding earlier, polymorphism is another extremely effective mechanism for minimizing ripple effects on an application if requirements change after the application has been deployed. We're able to introduce new subclasses to an application's class hierarchy in order to meet such requirements, and our existing client code won't "break."

Three Distinguishing Features of an Object-Oriented Programming Language

We've now defined all three of the features required to make a language truly object oriented:

- (Programmer creation of) User-defined types
- Inheritance
- Polymorphism

By way of review, let's look at the benefits of each of these language features.

The Benefits of User-Defined Types

The following are among the benefits of user-defined types:

- User-defined types provide an intuitive way to represent real-world objects, resulting in ***easier-to-verify requirements***.
- Classes are convenient units of reusable code, which means ***less code to write from scratch when building an application***.
- Through encapsulation, we ***minimize data redundancy***—each item of data is stored once, in the object to which it belongs—thereby ***lessening the likelihood of data integrity errors*** across an application.
- Through information hiding, we ***insulate our application against ripple effects*** if private details of a class must change after deployment, thereby ***dramatically reducing maintenance costs***.
- Objects are responsible for ensuring the integrity of their own data, making it ***easier to isolate errors in an application's (business) logic***; we know to inspect the method(s) of the class to which a corrupted object belongs.
- Once defined, a user-defined type (class) can be reused again and again across applications and even across organizations.

The Benefits of Inheritance

The following are among the benefits of inheritance:

- We can extend already deployed code without having to change and then retest it, resulting in ***dramatically reduced maintenance costs***.
- Subclasses are much more succinct, which means ***less code overall to write/maintain***.

The Benefits of Polymorphism

The following is one of the benefits of polymorphism:

- It ***minimizes "ripple effects"*** on client code when new subclasses are added to the class hierarchy of an existing application, resulting in ***dramatically reduced maintenance costs***.

ONE VERY IMPORTANT CAVEAT

A common misconception is that switching to object technology will dramatically reduce the time required to develop a given application. Anecdotes abound of managers who have expected that a team using an object-oriented approach should be able to craft an application in a fraction of the time that it would take them to build its non-OO counterpart—***despite the fact that team in question might be using OO techniques for the first time ever***! Unfortunately, due to the learning curve involved in switching to the OO paradigm—particularly for software developers who've been entrenched in non-OO techniques for many years—it will typically take ***longer*** for a team inexperienced with objects to develop their first OO application.

Where economies of scale ***do*** come into play for a properly designed OO application, however, is during the ***maintenance phase*** of the application's life cycle. The maintenance phase of an application—OO or otherwise—is typically much longer, and hence more costly, than the development phase. A general rule of thumb is that most application lifetimes are split between 20 percent development and 80 percent maintenance. By dramatically reducing ripple effects through the thoughtful application of (a) encapsulation/information hiding and (b) inheritance/polymorphism, we stand to reduce maintenance costs—and hence overall software life cycle costs—significantly.

> And, once we become adept in the OO paradigm, we should indeed be able to shorten application ***development*** time, as well. By virtue of the fact that classes can readily be reused and optionally extended/specialized via inheritance—***including the vast number of predefined classes that are provided as an integral part of an OOPL framework***—we'll have less code to write overall for a given application. If we in turn we embrace the philosophy of code sharing and reuse across projects, we can gain significant productivity in terms of development as well as maintenance across the life cycles of ***multiple*** applications.

Abstract Classes

We discussed in Chapter 5 how beneficial it is to consolidate common features of two or more classes into a common superclass, a process known as ***generalization***. For example, we noticed similarities between the `Student` and `Professor` classes (e.g., both declared a `name` attribute and methods to `get`/`set` its value), and so we created the `Person` class after the fact to serve as a generalization of both `Students` and `Professors`.

Let's now assume that we know at the very outset of an application development effort that we're going to want to take advantage of specialization. For example, with regard to the SRS, perhaps we're going to want to model various types of `Course` objects: lecture courses, lab courses, independent study courses, etc. We therefore want to start out on the right foot by designing a `Course` superclass first, to handle all of the common features of these various types of courses before we set out to derive specialized subclasses.

We might determine up front that all `Courses`, regardless of type, are going to need the following common attributes

- `String courseName`
- `String courseNumber`
- `int creditValue`
- *`CollectionType`* `enrolledStudents`
- `Professor instructor`

as well as the following common behaviors:

- `enrollStudent`
- `assignInstructor`
- `establishCourseSchedule`

Some of these behaviors may be generic enough so that we can afford to program them in detail for the `Course` class, knowing that it's a pretty safe bet that any subclasses of `Course` will be able to inherit these methods as is, without needing to override them. For example, the `enrollStudent` and `assignInstructor` methods could be written generically as follows:

```
import java.util.ArrayList;

public class Course {
    private String courseName;
    private String courseNumber;
    private int creditValue;
    private ArrayList<Student> enrolledStudents;
    private Professor instructor;

    // Accessor methods would also be provided; details omitted ...

    public void enrollStudent(Student s) {
        enrolledStudents.add(s);
    }

    public void assignInstructor(Professor p) {
        setInstructor(p);
    }

    // etc.
```

When we attempt to program a generic version of the `establishCourseSchedule` method, however, we realize that the business rules governing how to establish a course schedule differ significantly for different types of courses:

- A lecture course may meet only once a week for three hours at a time.
- A lab course may meet twice a week for two hours each time.

- An independent study course may meet on a custom schedule that has been jointly negotiated by a given student and professor.

It would be a waste of time for us to bother trying to program a generic, "one-size-fits-all" version of the `establishCourseSchedule` method within the `Course` class, because no matter what generic logic we'd attempt to provide, ***all three subclasses***—`LectureCourse`, `LabCourse`, and `IndependentStudyCourse`—would wind up having to replace such logic by overriding the method to make it meaningful for them.

What other options do we have, then? Can we afford to simply ***omit*** the `establishCourseSchedule` method from the `Course` class entirely, adding such a method to each of the subclasses of `Course` as a ***new*** feature instead? Not if we want to take advantage of polymorphism with respect to this method. Consider the following example:

```
ArrayList<Course> courses = new ArrayList<>();

// Add a variety of different Course types to the collection.
courses.add(new LectureCourse());
courses.add(new LabCourse());
courses.add(new IndependentStudyCourse());
// etc.

for (Course c : courses) {
    // This next line of code is polymorphic.
    c.establishCourseSchedule("1/24/2005", "5/10/2005");
}
```

As we discussed earlier in the chapter, the polymorphic expression `c.establishCourseSchedule(...)` will be deemed valid by the Java compiler only if the `Course` class has defined such a method signature. Well, then, is it possible to "trick" the compiler by adding a "dummy" `establishCourseSchedule` method to the `Course` class that has the required method header, but that does nothing meaningful? If we program the method with an empty body

```
// This method does NOTHING!  Its method body is empty.
public void establishCourseSchedule(String startDate, String endDate) { }
```

it would indeed compile, and we could then allow each subclass to override the "do nothing" version of this method with a meaningful version. While it's ***possible*** to do so, it's ***not*** considered good programming practice to do so, for the following reasons:

- By providing the `Course` class with an `establishCourseSchedule` method, we're declaring that `Course` objects will be able to provide such a service on behalf of an application.
- However, if client code were to ever call upon a generic `Course` object to ***perform*** this service

  ```
  Course c = new Course();

  // We believe that we're calling upon c to perform the indicated
  service,
  // but behind the scenes, nothing is happening.
  c.establishCourseSchedule("1/24/2005", "5/10/2005");
  ```

  the method as implemented would do what its name ***implies*** that it will do; for that matter, it doesn't do anything at all!

Furthermore, there's no guarantee that any classes derived from `Course` will override this method to do something meaningful, either, so we could wind up with an entire hierarchy of `Course` types that are incapable of performing the `establishCourseSchedule` service in a meaningful way.

This is seemingly a dilemma! We ***know*** we'll need a type-specific `establishCourseSchedule` method to be programmed for all subclasses of `Course`. We ***don't*** want to go to the trouble of programming a ***meaningless*** version of this method in the superclass, but we must nonetheless equip the `Course` class to recognize such a method header in order to facilitate polymorphism. How do we communicate the requirement for an `establishCourseSchedule` method in all subclasses of `Course` and, more important, ***enforce its future implementation***?

OO languages such as Java come to the rescue with the concept of **abstract classes**. An abstract class is used to define ***what*** behaviors a class is required to perform without having to provide an explicit implementation of ***how*** each and every such behavior will be carried out. We program an abstract class in much the same way that we program a nonabstract class (a.k.a. a **concrete class**), with one exception: for those behaviors for which we can't (or care not to) program a generic implementation (e.g., the `establishCourseSchedule` method in the preceding example), we're permitted to

specify method ***headers*** without having to program the corresponding method ***bodies***. We refer to a "bodiless," or header-only, method declaration as an **abstract method**. And, to differentiate such methods from methods with bodies, we'll refer to the latter as **implemented methods**.

Let's go back to our `Course` class definition to add an `abstract` `establishCourseSchedule` method:

```
// Note the use of the "abstract" keyword in the class declaration.
public abstract class Course {
    private String courseName;
    private String courseNumber;
    private int creditValue;
    private ArrayList<Student> enrolledStudents;
    private Professor instructor;

    // Other details omitted.

    public void enrollStudent(Student s) {
        enrolledStudents.add(s);
    }

    public void assignInstructor(Professor p) {
        setInstructor(p);
    }

    // Note the use of the "abstract" keyword and the terminating
    // semicolon.
    public abstract void establishCourseSchedule(String startDate,
                                                 String endDate);
}
```

The `establishCourseSchedule` method is declared to be abstract by adding the `abstract` keyword to its header, just before the return type. Note that the header of an abstract method has no braces following the closing parenthesis of the parameter list. Instead, the header is followed by a semicolon (`;`)—that is, it's missing its code body, which normally contains the detailed logic of how the method is to be performed. The method must therefore be explicitly labeled as `abstract` to inform the compiler that we

didn't accidentally forget to program this method; rather, we knew what we were doing when we ***intentionally*** omitted the body.

Whenever a class contains one or more abstract methods, then we must declare the class abstract as a whole as well, by inserting the `abstract` keyword ahead of the `class` keyword in the class declaration:

```
public abstract class Course { ... }
```

If we forget to designate as `abstract` a class that contains one or more abstract methods, a compilation error such as the following will arise:

```
Course should be declared abstract; it does not define
establishCourseSchedule(String, String)
```

Note that it isn't necessary for all of the methods in an abstract class to be abstract; an abstract class can also declare implemented methods. For example, in our abstract `Course` class, both the `enrollStudent` and `assignInstructor` methods are implemented.

By providing an abstract `establishCourseSchedule` method in the `Course` class, we've specified a service that all types of `Course` objects must be able to perform, but without pinning down the private details of ***how*** the service should be performed by a given subclass. We're instead leaving it up to each of the subclasses—`LectureCourse`, `LabCourse`, and `IndependentStudyCourse`—to specify its own class-appropriate way of performing the service. This is accomplished by requiring each of the subclasses to ***override*** the ***abstract*** method with an ***implemented*** version.

Implementing Abstract Methods

When we derive a class from an abstract superclass, the subclass will inherit all of the superclass's features, including all of its ***abstract*** methods. To replace an inherited abstract method with a concrete version, the subclass need merely override it; in so doing, we drop the `abstract` keyword from the method header and replace the terminating semicolon with a method body (i.e., code enclosed in braces).

Let's illustrate this approach by deriving a class called `LectureCourse` from the `Course` class:

```
// Deriving a concrete subclass from an abstract superclass.
public class LectureCourse extends Course {
```

```
    // Details omitted.

    // Override the abstract establishCourseSchedule method with a concrete
    // version by (a) removing the abstract keyword from the method header
    // and (b) providing a method body.
    public void establishCourseSchedule(String startDate,
                                        String endDate) {
        // Logic specific to the business rules for a LectureCourse
        // would be provided here ... pseudocode.
        determine what day of the week the startDate falls on;
        determine how many weeks there are between startDate and endDate;
        schedule one three-hour class meeting per week on the appropriate
            day of the week;
    }
}
```

Note that in overriding the `establishCourseSchedule` method, we've dropped the `abstract` keyword from the method header because the method is no longer abstract; we've implemented it by providing it with a method body. In so doing, we're also able to drop the `abstract` keyword from the `LectureCourse` class declaration

```
// No need for the "abstract" keyword here!
public class LectureCourse extends Course { ... }
```

because `LectureCourse` no longer contains any abstract methods; it is now a ***concrete*** class.

Unless a class derived from an abstract class implements ***all*** of the abstract methods that it inherits, the subclass must still be declared to be abstract. For example, say that in deriving a class named `IndependentStudyCourse` from our abstract `Course` class, we neglect to implement the `abstract establishCourseSchedule` method. If we were to try to compile `IndependentStudyCourse`, we'd get the following compilation error:

```
IndependentStudyCourse should be declared abstract; it does not define
establishCourseSchedule(String, String) in Course
```

To get our `IndependentStudyCourse` class to compile properly, we have two options for how to amend it:

- We have to implement the `abstract establishCourseSchedule` method inherited from `Course`, as we did for the `LectureCourse` subclass.
- We have to declare the `IndependentStudyCourse` class as a whole to be `abstract`:

  ```
  public abstract class IndependentStudyCourse extends
  Course { ... }
  ```

Note that allowing a subclass to remain abstract isn't necessarily a mistake, as we'll discuss a bit later in this chapter.

Abstract Classes and Instantiation

Abstract classes ***cannot be instantiated***. That is, if the `Course` class is declared to be abstract, then we can't ever instantiate generic `Course` objects in our application—an attempt to do so will result in a compilation error:

```
Course c = new Course();  // Impossible!
```

Here's the error message:

```
Course is abstract; cannot be instantiated
```

Why does the compiler prevent us from creating `Course` objects? The answer lies in the fact that the `Course` class declares an `establishCourseSchedule` method ***header***, thus implying that `Course`s are able to perform this service, but providing no body to explain ***how*** the method is to be performed. If we ***were*** able to instantiate a `Course` object, it would therefore be expected to know how to respond to a service request such as the following:

```
c.establishCourseSchedule("01/24/2005", "05/10/2005");  // Behavior
undefined!
```

But because there is no executable method body associated with the abstract `establishCourseSchedule` method, the `Course` object in question wouldn't know how to behave in response to such a message at run time. So the compiler is actually doing us a favor by preventing us from creating this impossible run-time situation to begin with.

We've just hit upon the mechanism for how abstract methods serve to enforce implementation requirements! Declaring an abstract method in a superclass ultimately ***forces*** all subclasses to provide implementations of all inherited abstract methods; otherwise, the subclasses themselves will also be abstract, and we won't be able to instantiate them either. Therefore, somewhere along the line in a derivation hierarchy, a class derived from an abstract class must provide concrete implementations for all of its inherited abstract methods if we wish to "break the spell of abstractness"—that is, if we wish to instantiate objects of that particular derived type. Referring to Figure 7-1

- Class `A` is abstract because it declares an abstract method `foo`; objects of type `A` therefore ***cannot*** be instantiated.
- Class `X` is derived from `A` and is a concrete class because it concretely implements the abstract method `foo` as inherited from `A`. Thus, we ***can*** instantiate objects of type `X`.
- Class `B` is abstract because it inherits the abstract method `foo` from `A` without implementing it. Note that `B` also introduces a second abstract method, `bar`, of its own. Objects of type `B` therefore ***cannot*** be instantiated.
- Class `Y` is concrete because it implements all of the abstract methods that it has inherited from its various ancestors—`foo` from `A` and `bar` from `B`. Thus, we ***can*** instantiate objects of type `Y`.

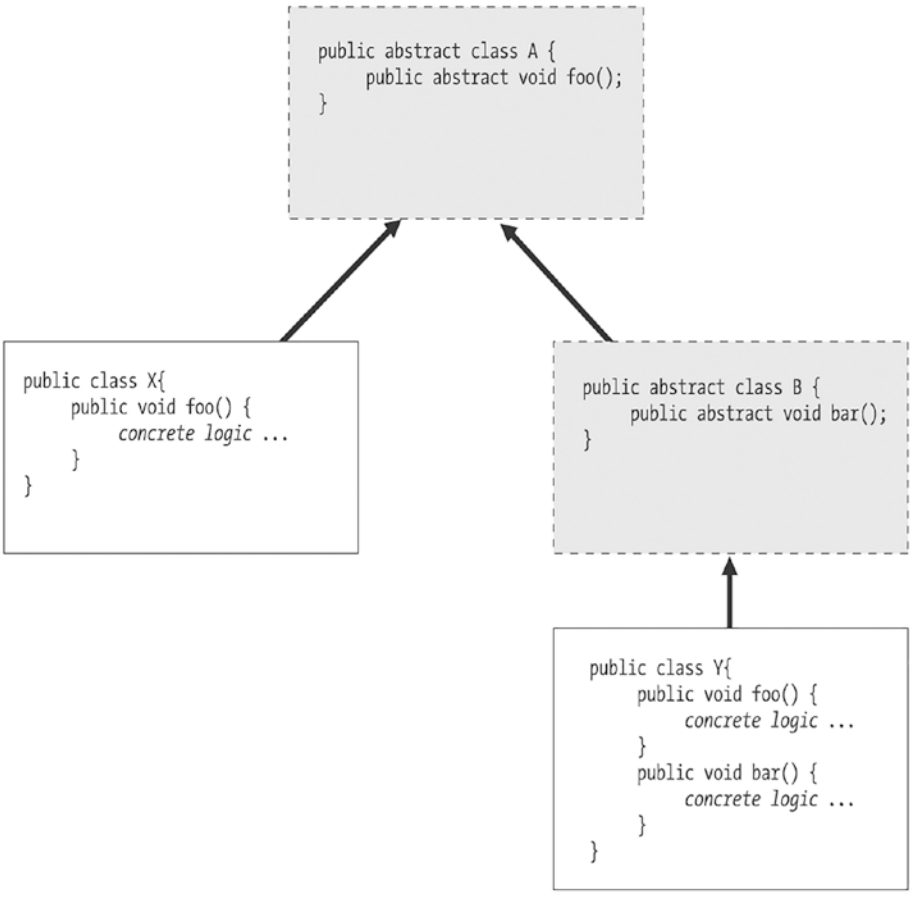


Figure 7-1. *"Breaking the spell of abstractness" by implementing abstract methods*

As mentioned earlier, allowing a subclass (e.g., `IndependentStudyCourse` from the earlier example) to remain an abstract class isn't necessarily a mistake. It's perfectly acceptable to have multiple layers of abstract classes in an inheritance hierarchy; we simply need a terminal/leaf class to be concrete in order for it to be useful in creating objects.

Declaring Reference Variables of Abstract Types

Despite the fact that we're prevented from instantiating an abstract class, we're nonetheless permitted to declare ***reference variables*** to be of an abstract type; this is necessary to facilitate polymorphism, as in the following code:

```
for (Course c : courses) {
    c.establishCourseSchedule(...);
}
```

Here, we're declaring c to be of (abstract) type Course at compile time, knowing that, at ***run time***, c will actually be referring to an object belonging to a ***specific concrete subclass*** of Course, for reasons that we discussed earlier.

An Interesting Twist on Polymorphism

Let's now explore an interesting polymorphic phenomenon specific to abstract classes: it's possible for a ***concrete*** method in an abstract class to invoke an ***abstract*** method in the ***same*** class. This is illustrated in the following code, where the concrete initializeCourse method invokes the abstract establishCourseSchedule method:

```
public abstract class Course {
    // Details omitted.

    // An abstract method ...
    public abstract void establishCourseSchedule(String startDate,
                                                 String endDate);

    // ... and a concretely implemented method that INVOKES the
    // abstract method.
    public void initializeCourse(Professor p, String s, String e) {
        // We assume that both assignInstructor and reserveClassroom are
        // implemented methods of the Course class ... details omitted.
        assignInstructor(p);
        reserveClassroom();

        // Here, we're invoking an abstract method -- HOW IS THIS
        POSSIBLE???
        establishCourseSchedule(s, e);
    }
}
```

How can this be possible? That is, how will we ever be able to invoke the `initializeCourse` method if there is no ***body*** defined for the `establishCourseSchedule` method upon which `initializeCourse` depends? The fact of the matter is that the compiler will never let us get into this predicament, for the following reasons:

- First of all, recall that it's impossible to invoke `initializeCourse` on a `Course` object specifically because we cannot ***instantiate*** a `Course` object in the first place—`Course` is abstract!
- Then, for any class ***derived*** from `Course` (e.g., `LectureCourse`), one of two sets of circumstances will be true:
  - `LectureCourse` will provide implementations of ***all*** abstract methods that it has inherited—`establishCourseSchedule` included—such that, at run time, there will be no ambiguity as to what the `initializeCourse` method is to do behind the scenes:

    ```
    LectureCourse l = new LectureCourse();

    // This next line of code is perfectly fine, because behind the
    // scenes, initializeCourse will invoke the
    establishCourseSchedule
    // method that LectureCourse has implemented.
    l.initializeCourse(p, s, e);
    ```

  - Alternatively, if `LectureCourse` ***doesn't*** implement `establishCourseSchedule`, then `LectureCourse` will by definition be an abstract class, such that we won't be able to instantiate a `LectureCourse` object in the first place:

    ```
    // Now THIS line of code won't compile, because LectureCourse
    is abstract ...
    LectureCourse l = new LectureCourse();

    // ... and so we'll never reach this ambiguous situation at
    run time!
    l.initializeCourse(p, s, e);
    ```

We thus see that implementing a method X that in turn relies on an ***abstract*** method Y is a "safe" thing to do, because by the time we're ever able to ***invoke*** method X on an object, the fact that the object exists means that all of its methods (including Y) have been implemented. Thus, in writing X, we can count on the ***future*** availability of a method Y that hasn't been implemented yet.

Interfaces

Recall that a class is an abstraction of a real-world object from which some of the unessential details have been omitted. We can therefore see that an ***abstract*** class is ***more*** abstract than a ***concrete*** class, because with an abstract class we've omitted the details for how one or more particular services are to be performed.

Now, let's take the notion of abstractness one step further. With an abstract class, we are able to avoid programming the bodies of methods that are declared to be abstract. But what about the ***data structure*** of such a class? In our abstract `Course` class example, we went ahead and prescribed the attributes that we thought would be needed generically by all types of courses, so that a common data structure would be inherited by all subclasses:

```
private String courseName;
private String courseNumber;
private int creditValue;
private ArrayList<Student> enrolledStudents;
private Professor instructor;
```

Suppose we only wanted to specify common ***behaviors*** of `Courses` and not even ***bother*** with declaring attributes. Attributes are, after all, typically declared to be private; we may not wish to mandate the private data structure that a subclass must use in order to achieve the desired public behaviors, instead leaving it up to the designer of the subclass to ultimately make this determination.

As an example, say that we wanted to define what it means to teach at a university. Perhaps, in order to teach, an object would need to be able to perform the following services:

- Agree to teach a particular course.
- Designate a textbook to be used for the course.

- Define a syllabus for the course.
- Approve the enrollment of a particular student in the course.

Each of these behaviors could be formalized by specifying a method header, representing how an object that is ***capable of teaching*** would be asked to perform each behavior:

```
public boolean agreeToTeach(Course c)
public void designateTextbook(TextBook b, Course c)
public Syllabus defineSyllabus(Course c)
public boolean approveEnrollment(Student s, Course c)
```

We could then declare an `abstract` class called `Teacher` that prescribes no data structure and only ***abstract*** methods:

```
public abstract class Teacher {
    // We omit attribute declarations entirely, allowing subclasses to
       establish
    // their own class-specific data structures.

    // We declare only abstract methods.
    public abstract boolean agreeToTeach(Course c);
    public abstract void designateTextbook(TextBook b, Course c);
    public abstract Syllabus defineSyllabus(Course c);
    public abstract boolean approveEnrollment(Student s, Course c);
}
```

We then proceed to create `Professor` as a concrete derivation of `Teacher`:

```
public Professor extends Teacher {
    // Declare relevant attributes.
    private String name;
    private String employeeID;
    private ArrayList<Section> teachingAssignments; // of Section objects
    // etc.

    // Provide concrete implementations of all inherited abstract methods.
    public boolean agreeToTeach(Course c) { ... }
    public void designateTextbook(TextBook b, Course c) { ... }
```

```
    public Syllabus defineSyllabus(Course c) { ... }
    public boolean approveEnrollment(Student s, Course c) { ... }

    // Additional methods can also be declared - details omitted.
}
```

However, if our intention is to declare a set of abstract method headers to define what it means to assume a certain ***role*** within an application (such as teaching) without imposing either data structure or concrete behavior on the subclasses, then the preferred way to do so in Java is with an **interface**.

Here's how we'd render the abstract `Teacher` class with an equivalent interface:

```
// Note use of "interface" vs. "abstract class" keywords.
public interface Teacher {
    boolean agreeToTeach(Course c);
    void designateTextbook(TextBook b, Course c);
    Syllabus defineSyllabus(Course c);
    boolean approveEnrollment(Student s, Course c);
}
```

Here are some observations about interface syntax:

- We use the keyword `interface` rather than `class` when declaring them:

  ```
  public interface Teacher { ... }
  ```

- Because all of an interface's methods are implicitly `public` and `abstract`, we needn't specify either of those two keywords when declaring them (although doing so will not generate a compiler error). We ***will*** get an error, however, if we attempt to assign something other than `public` accessibility to a method within an interface:

  ```
  public interface teacher {
      // This won't compile - interface methods must all be public.
      private void takeSabbatical();
      // etc.
  ```

Here's the compiler error:

```
modifier private not allowed here
private void takeSabbatical();
^
```

Because all of the methods prescribed by an interface are abstract, ***none*** of them have bodies.

As with classes, the source code for each interface typically goes into its own `.java` file, whose external name matches the name of the interface contained within (e.g., the `Teacher` interface would go into a file named `Teacher.java`). Interfaces are then compiled into bytecode in the same way that classes are compiled. For example, the command

```
javac Teacher.java
```

will produce a bytecode file named `Teacher.class`.

Note that interfaces may not declare variables (with one exception that we'll discuss later in the chapter) and they may not declare any implemented methods. They are, simply put, a collection of abstract method headers. Therefore, in terms of the "abstractness spectrum," ***an interface is more abstract than an abstract class*** (which is in turn more abstract than a concrete class) because an interface leaves even more details to the imagination.

Reformat as editorial note As of Java 8, the notion of a **functional interface** was introduced; such interfaces differ a bit in terms of the rules for declaring and using them beyond the scope of this chapter to discuss.

Implementing Interfaces

Once we've defined an interface such as `Teacher`, we can set about designating various classes of objects as being teachers—for example, `Professors` or `Students` or generic `Person` objects—simply by declaring that the class of interest ***implements*** the `Teacher` interface, using the following syntax:

```
// Implementing an interface ...
public class Professor implements Teacher { ... }
```

That is, rather than using the `extends` keyword, as we do when one class is derived from another, we use the `implements` keyword.

Recall our discussion of packages from Chapter 6. If we wish to implement a predefined Java interface type (the Java language provides many of these), we must remember to use an `import` directive to make that interface type known to the compiler, for example:

```
import packagename.PredefinedInterfaceType;

public class MyClass implements
PredefinedInterfaceType { ... }
```

Once a class declares that it's implementing an interface, the implementing class ***must*** implement all of the (implicitly abstract) methods declared by the interface in question in order to satisfy the compiler. As an example, let's say that we were to code the `Professor` class as follows, implementing three of the four methods called for by the `Teacher` interface but neglecting to code the `approveEnrollment` method:

```
public class Professor implements Teacher {
    private String name;
    private String employeeId;
    // etc.

    // We implement three of the four methods called for by the
    // Teacher interface, to provide method bodies.

    public boolean agreeToTeach(Course c) {
        logic for the method body goes here; details omitted ...
    }

    public void designateTextbook(TextBook b, Course c) {
        logic for the method body goes here; details omitted ...
    }

    public Syllabus defineSyllabus(Course c) {
        logic for the method body goes here; details omitted ...
    }
```

```
    // However, we FAIL to provide an implementation of the
    // approveEnrollment method.

    // Other miscellaneous methods of Professor unrelated to the Teacher
    // interface could also be declared ... details omitted.
}
```

If we were to try to compile the `Professor` class as just shown, we'd get the following compiler error:

```
Professor should be declared abstract; it does not implement
approveEnrollment(Student, Course) in Teacher
```

Recall that this is the ***exact same type*** of compiler error message that is generated if we are deriving a class from an abstract class and fail to override one of the inherited abstract methods. Here's the result from an earlier example, involving the abstract superclass `Course` and the subclass `IndependentStudyCourse`:

```
IndependentStudyCourse should be declared abstract; it does not implement
establishCourseSchedule(String, String) in Course
```

Thus, implementing an interface is conceptually similar to extending an abstract class in that both interfaces and abstract classes are alternative constructs for prescribing abstract behaviors that implementing subclasses must be able to carry out.

When should we use one vs. the other?

- If we wish to impart a particular data structure to go along with these prescribed behaviors or if we need to provide some concrete behaviors along with abstract behaviors, we'll create an abstract class.
- Otherwise, we'll create an interface.

Tables 7-1 and 7-2 summarize the syntactical differences between interfaces and abstract classes.

Table 7-1. *Syntactical Differences for Declaring Abstract Classes vs. Interfaces*

Example Using an Abstract Class	Example Using an Interface
Declaring the `Teacher` Type As an Abstract Class:	**Declaring the `Teacher` Type As an Interface:**

`public abstract class Teacher {`
`  // Abstract classes may prescribe`
`  // data structure.`
`  private String name;`
`  private String employeeId;`
`  // etc.`
 	
`  // We declare abstract methods using`
`  // the "abstract" keyword; these`
`  // must also be declared "public".`
`  public abstract void agreeToTeach(`
`  Course c);`
 	
`  // etc.`
 	
`  // Abstract classes may also declare`
`  // concrete methods.`
`  public void print() {`
`  System.out.println(name);`
`  }`
 	
`  // etc.`
`}`
	`public interface Teacher {`
	`  // Interfaces may NOT prescribe`
	`  // data structure.`
	
	`  // We needn't use the "public" or`
	`  // "abstract" keywords - all methods`
	`  // declared by an interface are`
	`  // automatically public and`
	`  // abstract by default.`
	`  void agreeToTeach(Course c);`
	
	`  // etc.`
	
	`  // Interfaces may NOT declare`
	`  // concrete methods.`
	`}`

Table 7-2. *Syntactical Differences for Extending Abstract Classes vs. Implementing Interfaces*

Example Using an Abstract Class	Example Using an Interface
Professor *Extends* Teacher:	**Professor *Implements* Teacher:**

`public class Professor extends Teacher {`
`  // Professor inherits attributes, if`
`  // any, from the abstract`
`  // superclass, and optionally`
`  // adds additional attributes.`
`  private Department worksFor;`
`  // etc.`
 	
`  // We override abstract methods`
`  // inherited from the Teacher class`
`  // to provide a concrete`
`  // implementation.`
`  public void agreeToTeach(Course c)`
`  {`
`  *logic for the method body goes here; details omitted ...*`
`  }`
`  // etc. for other abstract methods.`
 	
`  // Additional methods may be added;`
`  // details omitted.`
`}`
	`public class Professor implements Teacher {`
	`  // Professor must provide ALL of`
	`  // its own data structure, as an`
	`  // interface cannot provide this.`
	`  private String name;`
	`  private String employeeId;`
	`  private Department worksFor;`
	`  // etc.`
	`  // We implement methods required by`
	`  // the Teacher interface.`
	`  public void agreeToTeach(Course c) {`
	`  *logic for the method body goes here; details omitted ...*`
	`  }`
	`  // etc. for other abstract methods.`
	`  // Additional methods may be added;`
	`  // details omitted.`
	`}`

Another Form of the "Is A" Relationship

You learned in Chapter 5 that inheritance is often referred to as the "is a" relationship. As it turns out, implementing an interface is another form of "is a" relationship; that is

- If the `Professor` class ***extends*** the `Person` ***class***, then a `Professor` ***is a*** `Person`.
- If the `Professor` class ***implements*** the `Teacher` ***interface***, then a `Professor` ***is a*** `Teacher`.

When a class A implements an interface X, all of the classes that are subsequently derived from A may also be said to implement that same interface X. For example, if we derive a class called `AdjunctProfessor` from `Professor`, then since `Professor` implements the `Teacher` interface, an `AdjunctProfessor` is a `Teacher`, as well:

```
public class Professor implements Teacher {
    // Attribute details omitted.

    // The Professor class must implement all four of the methods
    called for by
    // the Teacher interface.
    public boolean agreeToTeach(Course c) { ... }
    public void designateTextbook(TextBook b, Course c) { ... }
    public Syllabus defineSyllabus(Course c) { ... }
    public boolean approveEnrollment(Student s, Course c) { ... }

    // Other details omitted.
}

// Even though AdjunctProfessor isn't explicitly declared to implement
Teacher,
// it does so IMPLICITLY, because it inherits all of a Teacher's
behaviors from
// the Professor class.
public class AdjunctProfessor extends Professor { ... }
```

This makes intuitive sense, because `AdjunctProfessor` will either inherit all of the methods called for by the `Teacher` interface from `Professor` as is or optionally override one or more of them. Either way, an `AdjunctProfessor` will be "equipped" to perform ***all*** of the services required to serve in the role of a `Teacher` on behalf of an application:

- Agree to teach a particular course.
- Designate a textbook to be used for the course.
- Define a syllabus for the course.
- Approve the enrollment of a particular student in the course.

Recall that this is the ***precise*** purpose for having declared the `Teacher` interface in the first place: to define a behavioral role in an application. So, even though `AdjunctProfessor` isn't explicitly declared to implement `Teacher`, it does so ***implicitly***.

Interfaces and Casting

Note that the compiler is perfectly happy for us to assign a `Professor` object to a `Teacher` reference variable

```
Teacher t = new Professor();
```

because the compiler generally allows assignments to occur if the type of the expression to the right of the equal sign (=) is a type that is compatible with the variable to the left of the equal sign. Since `Professor` ***implements*** `Teacher`, a `Professor` ***is a*** `Teacher`, and so this assignment is permitted.

The opposite is ***not*** permitted, however: we cannot directly assign a `Teacher` reference to a `Professor` reference variable, because not all `Teachers` are necessarily `Professors`—many different classes can implement the same interface. For example, assuming that both the `Student` and `Professor` classes implement the `Teacher` interface, the last line of the following code will generate a compiler error:

```
Professor p = new Professor();
Student s = new Student();
Teacher t;

// Details omitted.

// The compiler won't allow this.
p = t;
```

Here's the compiler error:

```
incompatible types
found:    Teacher
required: Professor
p = t;
    ^
```

However, if we know that `t` will be referring to an object of an appropriate type at run time, we may ***force*** such an assignment to occur through use of a cast. Recall from Chapter 2 that we use casting to convince the compiler that an assignment should occur even though precision is lost in doing so (e.g., when assigning a `double` value to an `int` variable):

```
int x;
double y;

// Cast the double value into an int before assigning it to x.
x = (int) y;
```

Recall that this was referred to as a ***narrowing conversion***. In a sense, attempting to assign a `Teacher` reference to a `Professor` reference variable is also a narrowing conversion: we're trying to narrow down all of the possible types of object that a `Teacher` variable could possibly be referencing at run time to a ***single*** type, `Professor`. In the preceding example, since both the `Professor` and `Student` classes implement the `Teacher` interface, `t` could, at run time, be referring to a `Student` object or a `Professor` object, as illustrated in Figure 7-2.

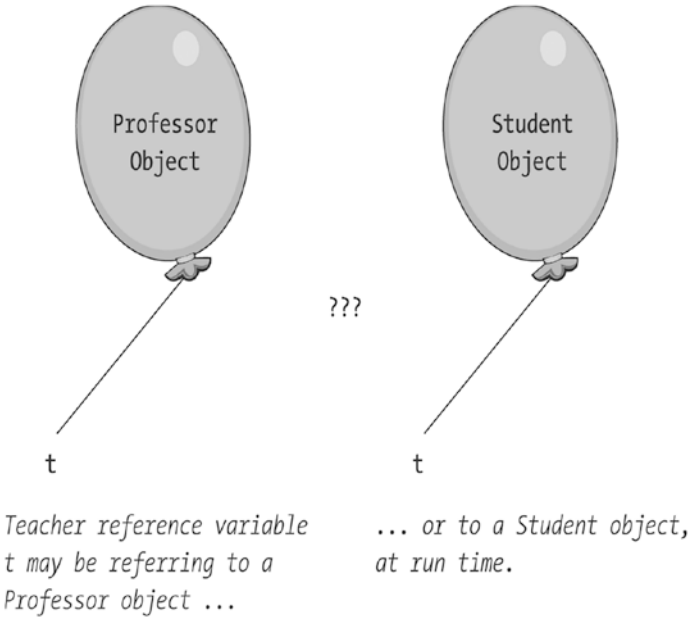


Figure 7-2. *A* `Teacher` *reference variable can refer to multiple types of object at run time*

However, if we ***know*** that, based on the way we've written our code, `t` will be referring to a `Professor` object at run time, we may force the assignment through the use of a cast as follows:

```
Professor p1 = new Professor();
Teacher t;

// We assign a Professor reference to t.
t = p1;

// Details omitted.

// Later in our application, we are confident that t is still referring
// to the same Professor, and so we assign t to p2 by using a cast.
Professor p2 = (Professor) t;
```

The resultant situation in memory is depicted in Figure 7-3. Our use of a cast on the last line of code

```
Professor p2 = (Professor) t;
```

is effectively telling the compiler "Trust me, I know that `t` will be referring to a `Professor` object at run time, so doing this assignment makes sense."

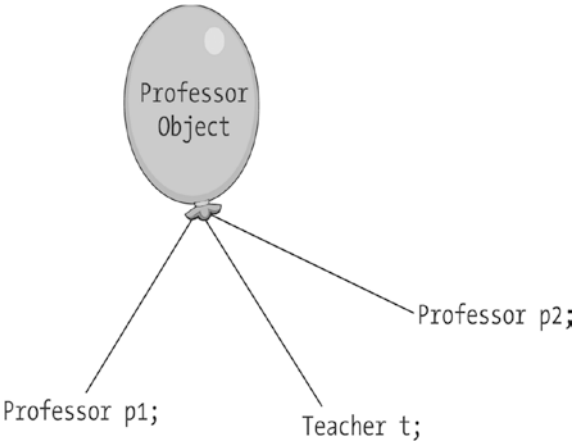


Figure 7-3. *Because `t` refers to a `Professor` object at run time, we force an assignment of `t` to `p2` via a cast*

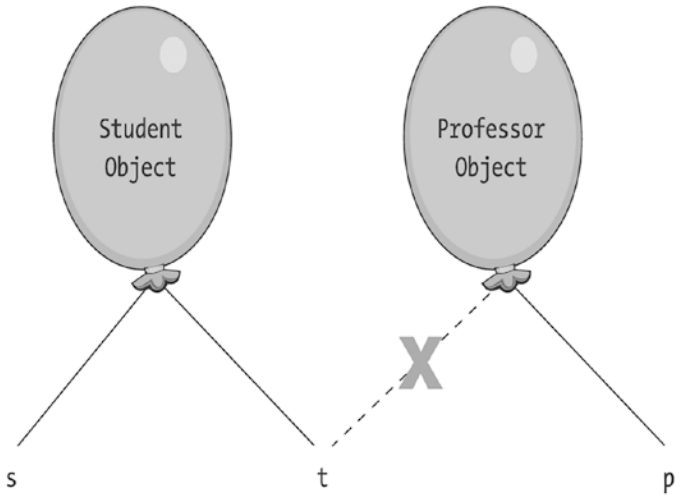
If we force a cast, but we're ***wrong***—that is, if the run-time type of `t` is ***not*** compatible with the `Professor` type—then we'll get a `ClassCastException` type of error at run time. (We'll talk about how to deal with such an error, a technique known as ***exception handling***, in Chapter 13.) Returning to our previous example, let's change the code a bit so that a cast would be inappropriate:

```
// We instantiate both a Professor and a Student object; recall that
// in this example, both classes implement the Teacher interface.
Professor p = new Professor();
Student s = new Student();
Teacher t;

// We assign a Student reference to t.  This is permitted, because
a Student
// is a Teacher.
t = s;
```

```
// Details omitted ...

// Later on, we mistakenly try to cast t as a Professor, but t is really
// referring to a Student.
p = (Professor) t;
```

The last line of code will compile, because the compiler trusts that we know what we are doing, but since the actual situation at run time is as depicted in Figure 7-4, such a cast is ***invalid***—there's no way to transform a `Student` into a `Professor` object at run time—and so we get the following error message when ***executing*** this code:

```
Exception in thread "main" java.lang.ClassCastException: Student
at classname.main(classname.java:line#)
```



t refers to a Student object at run time, and so trying to cast t as a Professor generates a ClassCastException.

Figure 7-4. *A `ClassCastException` arises at run time when trying to refer to a `Student` object as a `Professor`*

We'll revisit the use of casts with object references later in this chapter and again in Chapter 13.

Implementing Multiple Interfaces

Another important distinction between extending an abstract class and implementing an interface is that whereas a given class may only be derived from ***one*** direct superclass, a class may implement as ***many*** interfaces as desired. If a class is to implement multiple interfaces, we must name all such interfaces as a comma-separated list after the `implements` keyword in the class declaration:

```
public class ClassName implements Interface1, Interface2, ...,
InterfaceN { ... }
```

In so doing, the implementing class would then need to implement all of the methods prescribed by ***all*** of these interfaces collectively.

As an example, if we were to invent a second interface called `Administrator`, which in turn specified the following method headers

```
public interface Administrator {
    boolean approveNewCourse(Course c);
    boolean hireProfessor(Professor p);
    void cancelCourse(Course c);
}
```

we could then declare that the `Professor` class implements ***both*** the `Teacher` and `Administrator` interfaces as follows:

```
// The Professor class implements two interfaces.
public class Professor implements Teacher, Administrator {
    // Details omitted.

    // The Professor class must implement all four of the methods
    called for by
    // the Teacher interface ...
    public boolean agreeToTeach(Course c) { ... }
    public void designateTextbook(TextBook b, Course c) { ... }
    public Syllabus defineSyllabus(Course c) { ... }
    public boolean approveEnrollment(Student s, Course c) { ... }

    // ... as well as all three of the methods called for by
    // the Administrator interface.
```

```
    public boolean approveNewCourse(Course c) { ... }
    public boolean hireProfessor(Professor p) { ... }
    public void cancelCourse(Course c) { ... }

    // Details omitted.
}
```

If a class implements two or more interfaces that call for methods with identical signatures, we need only implement one such method in the implementing class—that method will do "double duty" in satisfying both interfaces' implementation requirements as far as the compiler is concerned.

When a class implements more than one interface, its objects are capable of assuming multiple identities or roles in an application; such objects can therefore be "handled" by various types of reference variables. Based on the preceding definition of a `Professor` as both a `Teacher` and an `Administrator`, the following client code would be possible:

```
// Instantiate a Professor object, and maintain a handle on it via
// a reference variable of type Professor.
Professor p = new Professor();

// We then declare reference variables of the two types of interfaces
that the
// Professor class implements.
Teacher t;
Administrator a;

t = p;  // We store a second handle on the same Professor in a reference
variable of
        // type Teacher; this is possible because a Professor IS A Teacher!

a = p;  // We store a third handle on the same Professor in a reference
variable of
        // type Administrator; this is possible because a Professor IS AN
        // Administrator!
```

This code is illustrated conceptually in Figure 7-5.

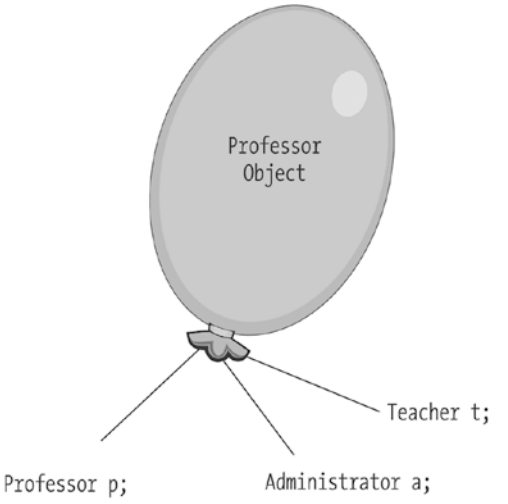


Figure 7-5. *A* `Professor` *object has three different identities/roles in our application*

This is conceptually the same thing as you, as a person, being viewed as having different roles by different people: you're viewed as an employee by your manager, as a son or daughter by your parents, perhaps as a partner by your significant other, as a parent by your children, etc.

We may then command the ***same*** object at run time as either a `Professor`

```
// setDepartment is a method defined by the Professor class ...
p.setDepartment("Computer Science");
```

or as a `Teacher`

```
// agreeToTeach is a method defined by the Teacher interface ...
t.agreeToTeach(c);
```

or as an `Administrator`

```
// approveNewCourse is a method defined by the Administrator interface ...
a.approveNewCourse(c);
```

because it's all three, rolled into one.

A class may simultaneously extend a ***single*** superclass and implement ***one or more*** interfaces, as follows:

```
public class Professor extends Person implements Teacher,
Administrator { ... }
```

Under such circumstances, `extends` *`className`* should always precede `implements` *`interfaceNameList`* in the declaration.

Interfaces and Casting, Revisited

Continuing with the previous example, note that, despite the fact that `t` would, at ***run time***, be referring to a `Professor` object, we ***cannot*** ask `t` to perform a method declared by the `Professor` class:

```
Professor p = new Professor();
Teacher t = p;

// setDepartment is a method defined for the Professor class, but t is
// declared to be of type Teacher ... this won't compile!
t.setDepartment("Computer Science");
```

The compiler will check the type of `t`, determining that `t` is ***declared*** to be of type `Teacher`, and since the `Teacher` interface doesn't declare a `setDepartment` method, the compiler will reject the last line of the preceding code with the following compilation error:

```
cannot find symbol
symbol:  method setDepartment(String)
location:  interface Teacher
```

Thus, even if the code that we've written ***guarantees*** that `t` would be referring to a `Professor` at ***run time***, we may only command `t` at ***compile*** time as a `Teacher`, because not all `Teachers` are necessarily `Professors` as far as the compiler is concerned.

Once again, casting comes to our rescue: if we are ***certain*** that `t` will indeed be referring to a `Professor` object at run time, we may invoke the `setDepartment` method on that object by casting the reference to `t` as follows:

```
Professor p = new Professor();
Teacher t = p;

// setDepartment is a method defined for the Professor class; since we know
// that t will refer to a Professor at run time, we use a cast so that
// this will compile.
((Professor) t).setDepartment("Computer Science");
```

Note the use of nested parentheses: `((Professor) t).setDepartment(...)`. We use nested parentheses to force the cast to occur ***before*** we attempt to invoke the `setDepartment` method on `t`. If we were to write the line of code ***without*** nested parentheses, as follows

```
(Professor) t.setDepartment("Computer Science");
```

then the compiler would interpret this as saying, "First, invoke the `setDepartment` method on `t`, and ***then*** cast the result that is ***returned*** from this method invocation to be a `Professor`," which is not what we want. (And, in fact, since a `set` method typically is declared to have a return type of `void`, the preceding line won't even compile.)

Interfaces and Instantiation

As with abstract classes, interfaces cannot be instantiated. That is, if we define `Teacher` to be an interface, we may not instantiate it directly

```
Teacher t = new Teacher();  // Impossible!
```

because interfaces don't have constructors—only classes do, as templates for instantiating objects—and so we'd encounter the following compilation error:

```
Teacher is abstract; cannot be instantiated
```

Recall that this is the ***exact same type*** of compiler error message that is generated if we attempt to instantiate an abstract class. Here's the result from an earlier example:

```
Course is abstract; cannot be instantiated
```

While we're indeed prevented from instantiating an interface, we're nonetheless permitted to declare reference variables to be of an interface type, as we were able to do with abstract classes:

```
Teacher t;  // This is OK.
```

This is necessary to facilitate polymorphism, as discussed in the next section.

Interfaces and Polymorphism

Let's look at an example of polymorphism as it applies to interfaces. We'll assume that

- The `Professor` and `Student` classes are both derived from the `Person` class.
- `Professor` and `Student` are sibling classes—neither derives from the other.
- `Person` implements the `Teacher` interface, and thus by virtue of inheritance, both `Professor` and `Student` ***implicitly*** implement the `Teacher` interface.

We may declare a collection to hold `Teacher` references and then populate it with a mixture of `Student` and `Professor` object references as follows:

```
ArrayList<Teacher> teachers = new ArrayList<Teacher>();
teachers.add(new Student("Becky Elkins"));
teachers.add(new Professor("Bobby Cranston"));
// etc.
```

We may then iterate through the `teachers` collection in polymorphic fashion, referring to all of its elements as `Teachers`

```
for (Teacher t : teachers) {
    // This line of code is polymorphic.
    t.agreeToTeach(c);
}
```

because we constrained the collection to contain only `Teacher`-type object references when we first declared it.

The Importance of Interfaces

Interfaces are one of the most poorly understood, and hence underused, features of the OO programming languages that support them. This is quite unfortunate, as interfaces are extremely powerful if used properly.

Whenever possible/feasible in developing an application, if we use ***interface*** types rather than specific class types in declaring

- (Private) Attributes
- Formal parameters to methods
- Method return types

our classes will be more flexible in terms of how client code can use them. Let's explore two different examples to see why this is so.

Example #1

In this example, let's assume that

- The `Professor` and `Student` classes are both immediate subclasses of `Person`, along with a third subclass, `Janitor`.
- In this example, `Person` does ***not*** implement the `Teacher` interface, because we don't wish to designate either `Janitors` or `Students` as `Teachers`; instead, we have the `Professor` class (only) ***explicitly*** implement the `Teacher` interface.

The four class declarations are as follows:

```
public class Person { ... }

// Only Professors are Teachers in this example.
public class Professor extends Person implements Teacher { ... }

public class Student extends Person { ... }

public class Janitor extends Person { ... }
```

Next, we'll design a class called `Course` with a `private` attribute of type `Professor` called `instructor`, along with accessor methods for this attribute:

```
public class Course {
    private Professor instructor;
    // Other attributes omitted from this example ...

    public Professor getInstructor() {
        return instructor;
    }

    public void setInstructor(Professor p) {
        instructor = p;
    }

    // Other methods omitted from this example ...
}
```

We'd then perhaps use this class from client code as follows to assign a specific `Professor` to teach a specific `Course`:

```
// Client code.
Course c = new Course("Math 101");
Professor p = new Professor("John Smith");
c.setInstructor(p);
```

Let's say that at some future date the university decides to permit selected students to teach courses. To implement this new business rule, we derive a new subclass of `Student` called `StudentInstructor` and have it implement the `Teacher` interface. Our classes are thus as follows:

```
public class Person { ... }

// Professors are Teachers ...
public class Professor extends Person implements Teacher { ... }

public class Student extends Person { ... }

// ... and now selected Students are Teachers, as well!
public class StudentInstructor extends Student implements Teacher { ... }

public class Janitor extends Person { ... }
```

We would not be able to assign a `StudentInstructor` to teach a `Course` given the current design of our `Course` class, however, because a `StudentInstructor` is not a `Professor` in terms of our class hierarchy; that is, the following client code would not compile:

```
Course c = new Course("Math 101");
StudentInstructor si = new StudentInstructor("Mary Jones");

// An attempt to assign a student as an instructor won't compile.
c.setInstructor(si);
```

We'd get the following compilation error:

```
setInstructor(Professor) in Course cannot be applied to StudentInstructor
```

Now, let's look at an improvement to our original `Course` class design. Instead of declaring the `instructor` attribute of `Course` to be of type `Professor` (a specific ***class*** type), let's declare it to be of type `Teacher` (an ***interface*** type) instead. We'll also adjust the return type of the "`get`" method and the parameter type of the "`set`" method for this attribute accordingly:

```
public class Course {
    // We've changed our declaration of the instructor attribute to take
    // advantage of an interface type.
    private Teacher instructor;
    // Other attributes omitted from this example ...

    // We make a corresponding change to the return type of our get
    method ...
```

```
    public Teacher getInstructor() {
        return instructor;
    }

    // ... and to the type of the parameter that we pass into the
    set method.
    public void setInstructor(Teacher t) {
        instructor = t;
    }

    // Other methods omitted from this example ...
}
```

We're thus opening up more possibilities for client code. Using our improved `Course` class design, we can assign a `Professor` as an instructor for a `Course`

```
// Client code.
Course c = new Course("Math 101");
Professor p = new Professor("John Smith");
c.setInstructor(p);
```

or a `StudentInstructor` as an instructor for a `Course`

```
// Client code.
Course c = new Course("Math 101");
StudentInstructor si = new StudentInstructor("George Jones");
c.setInstructor(si);
```

or a reference `x` to any other ***future*** type of object as yet to be invented

```
c.setInstructor(x);
```

as long as `x` is an instance of a class that implements the `Teacher` interface.

We can therefore see that using an interface type when declaring

- The (private) `instructor` attribute of `Course`
- The parameter passed into the `setInstructor` method of `Course`
- The return types of the `getInstructor` method

results in a much more flexible design for our application overall.

Example #2

The Java language provides many predefined interfaces. One such example is the `Collection` interface of the `java.util` package. The `Collection` interface enforces implementation of 14 methods, many of which—`add`, `addAll`, `clear`, `contains`, `isEmpty`, `remove`, `size`, etc.—we discussed when talking about various collection classes in Chapter 6. These 14 methods collectively define the services that an object has to be able to provide in order to perform in the role of a proper `Collection` in a Java application.

The `Collection` interface is implemented by numerous predefined Java `Collection` classes, including the `ArrayList` class. In fact, the **collections framework** is based on a total of ***12*** interfaces, including `Map`, `SortedMap`, `Collection`, `Set`, `List`, and `SortedSet`. The relationships between these interfaces and the various collection classes that we have discussed are illustrated in Figure 7-6.

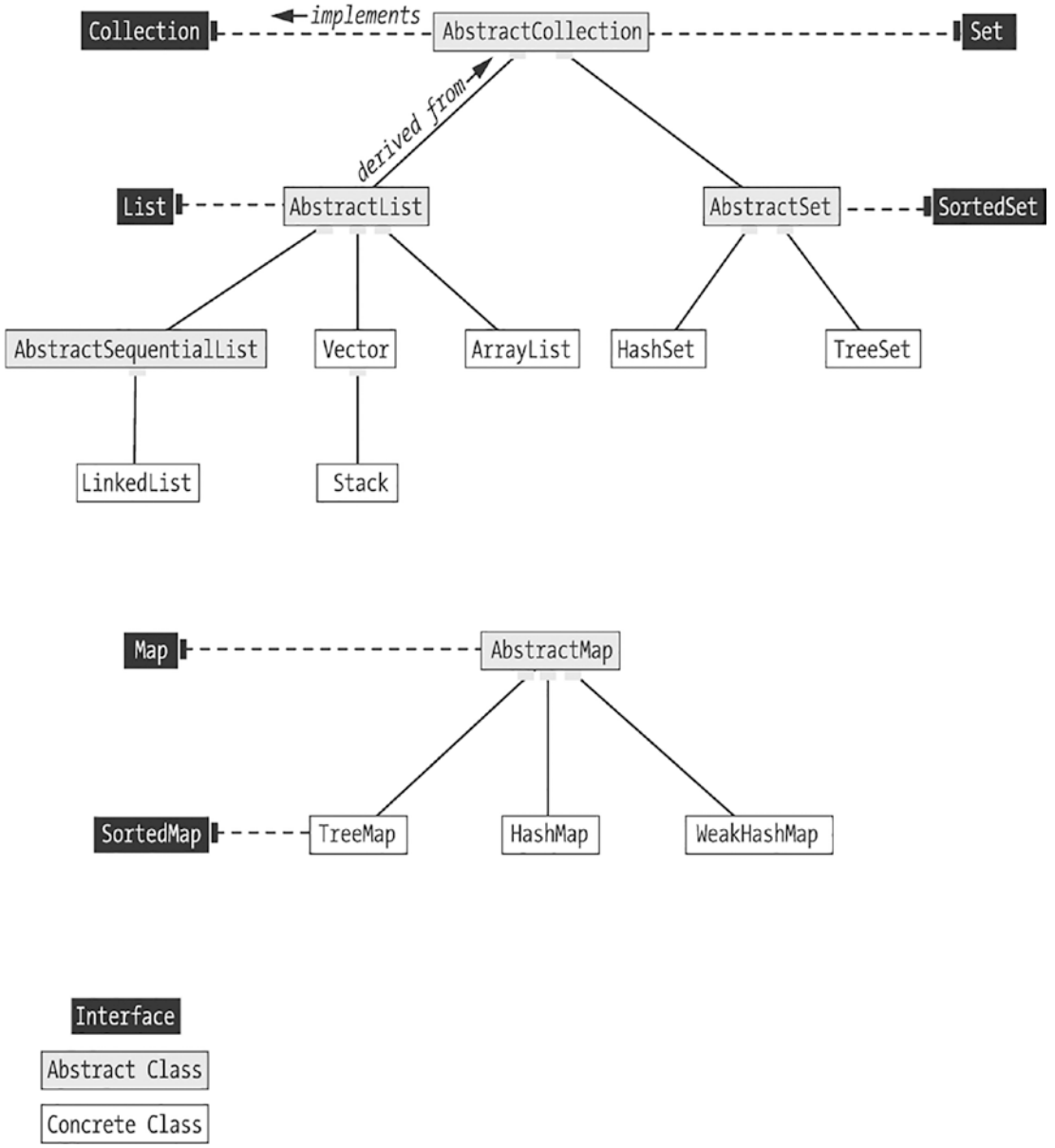


Figure 7-6. *The "family tree" of the predefined collection classes that we've discussed*

ALL JAVA COLLECTIONS ARE NOT CREATED EQUAL!

Figure 7-6 points out an interesting phenomenon regarding the `TreeMap` and `HashMap` classes, two of the predefined collection types that we discussed in Chapter 6. While `TreeMaps` and `HashMaps` are indeed collections in the ***generic*** sense in that they organize references to other objects, these classes do ***not*** implement the `Collection` interface. Hence, client code such as the following will ***not*** compile

```
Collection c = new TreeMap<String, String>();
```

because a `TreeMap`, while being a collection (in the lowercase "c" sense of the word), is not truly a `Collection` (in the formal, uppercase "C" sense). The following compilation error message would be generated:

```
incompatible types:
found:    java.util.TreeMap<java.lang.String, java.lang.String>
required: java.util.Collection
```

Similarly, an array is not truly a `Collection` in the uppercase "C" sense of the term either, and so the following won't compile either:

```
Collection c = new Student[20];
```

Here's the compilation error:

```
incompatible types:
found:    Student[]
required: java.util.Collection
```

On the other hand, the following client code ***will*** compile

```
Collection c = new ArrayList<String>;
```

because the `ArrayList` class is derived from the `AbstractCollection` class, which implements the `Collection` interface; hence, an `ArrayList` ***is a*** `Collection` (uppercase "C") in the true uppercase "C" sense.

If we design methods that are to operate on collections of objects to accept a ***generic*** `Collection` reference as an argument (rather than requiring that a ***specific*** type of collection be passed in), such methods will be much more versatile; client code will be free to pass in whatever `Collection` type it wishes.

By way of example, let's say that we wish to design an `enrollStudents` method for the `Course` class so that client code can pass in a collection of `Students` to enroll them all at once. If we were to specify a particular collection type as a parameter to the method—say, an `ArrayList`

```
import java.util.ArrayList;

public class Course {
    // Details omitted ...

    // Accept a specific collection type as an argument.
    public void enrollStudents(ArrayList x) {
        for (Student s : x)  {
            this.enroll(s);
        }
    }

    // etc.
}
```

then client code would be forced to pass in an `ArrayList` as an argument. However, if we design the method to accept a ***generic*** `Collection` as an argument instead

```
import java.util.Collection;

public class Course {
    // details omitted ...

    // Accept a generic Collection reference as an argument.
    public void enrollStudents(Collection c) {
        for (Student s : x)  {
            this.enroll(s);
        }
    }
}
```

then client code using this method will be able to pass in an `ArrayList` of `Student` references

```
Course c = new Course();
ArrayList al = new ArrayList();
// Populate al with Students ... details omitted.

// Pass in an ArrayList ...
c.enrollStudents(al);
```

or a `Vector` of `Student` references (another built-in type)

```
// Client code.
Course c = new Course();
Vector v = new Vector<Student>();
// Populate v with Students ... details omitted.

// Pass in a Vector ...
c.enrollStudents(v);
```

or any other type of `Collection` desired.

The same is true for methods that ***return*** collections of objects: if we design them to return generic `Collections` instead of specific collection types, then we are free to change the internal details of what type of collection we're crafting. Recall our discussion from Chapter 6 of the `getRegisteredStudents` method of the `Course` class. I've repeated that code here for your convenience:

```
import java.util.ArrayList;

public class Course {
    private ArrayList<Student> enrolledStudents;

    // Details omitted ...

    // The following method returns a reference to an entire collection -
    // specifically, an ArrayList - containing however many Students are
    // registered for the course in question.
    public ArrayList getRegisteredStudents() {
```

```
        return enrolledStudents;
    }

    // etc.
}
```

Because we declared `getRegisteredStudents` to have a return type of `ArrayList` (a specific collection type), we're going to run into problems later on if we decide to change the type of the encapsulated `enrolledStudents` collection from `ArrayList` to some other collection type. In essence, we've exposed client code to what should be a ***private*** detail of the `Course` class: namely, the type of collection that we're using ***internally*** to manage `Student` references.

If we instead declare `getRegisteredStudents` to return a ***generic*** `Collection` as follows

```
import java.util.ArrayList;
import java.util.Collection;

public class Course {
    // We're still maintaining an ArrayList internally.
    private ArrayList<Student> enrolledStudents;

    // Details omitted ...

    // However, we've now "hidden" the fact that we're using an ArrayList
    // internally by returning it as a generic Collection instead.
    public Collection<Student> getRegisteredStudents() {
        // We're allowed to do this, because enrolledStudents is an ArrayList,
        // and an ArrayList IS A Collection.
        return enrolledStudents;
    }

    // etc.
}
```

we're now free to change the type of internal collection that we're using (a ***private*** detail) without impacting the signature of the `getRegisteredStudents` method (a ***public*** detail). For example, we may wish to switch from an `ArrayList` to a `TreeSet` to take advantage of a set's inherent elimination of duplicate entries (recall our discussion of this aspect of set-type collections from Chapter 6):

```
import java.util.TreeSet;
import java.util.Collection;

public class Course {
    // We've switched to a different Collection type internally.
    private TreeSet enrolledStudents;

    // Details omitted ...

    // This method signature needn't change!!!
    public Collection<Student> getRegisteredStudents() {
        // We're allowed to do this, because enrolledStudents is a TreeSet,
        // and a TreeSet IS A Collection.
        return enrolledStudents;
    }

    // etc.
}
```

Be certain to master the notion of interfaces—both predefined and user-defined—to harness their power in your code!

In Chapter 6, our discussion of creating custom collections started out by mentioning that we could, if desired, invent a custom collection type from scratch, but that the Java language provides so many predefined collection types that doing so is not usually necessary. However, should you ever find yourself wanting to invent a brand-new collection type without extending one of the ***predefined*** collection types, be certain that your collection type implements the predefined `Collection` interface, at a minimum

```
import java.util.Collection;

public class MyBrandNewCollectionType implements
Collection { ... }
```

so that your collection type will be usable in any context where a generic `Collection` is required.

Static Features

Up until this point, all of the attributes that we've discussed have been associated with an individual instance of a class. Every `Student` object has its own copy of the `String` `name` attribute, for example, and can manipulate its value independently of what other `Student` objects are doing with ***their*** copies of the same attribute (see Figure 7-7).

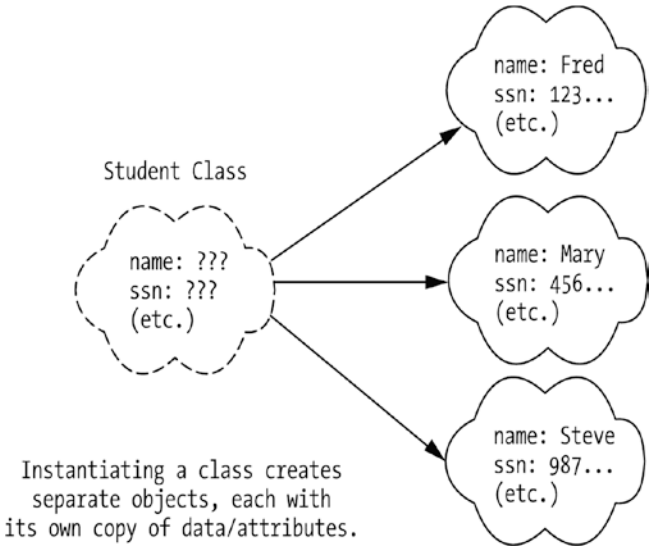


Figure 7-7. *Objects manage their individual attribute values*

Circumstances can arise in an application where we'll want all objects belonging to a particular class to ***share*** a single value of a particular variable instead of having each object maintain its own copy of that variable as an attribute. The Java language satisfies this need through **static variables** that are associated with classes as a whole rather than with individual objects.

Static Variables

Suppose there was some piece of general information—say, the count of the total number of students enrolled at the university—that we wanted ***all*** `Student` objects to have shared knowledge of. We could implement this as a simple attribute of the `Student` class, `int totalStudents`, along with code for manipulating the attribute as shown here:

```
public class Student {
    private int totalStudents;
    // Other attribute details omitted.

    // Accessor methods.

    public int getTotalStudents() {
        return totalStudents;
    }

    public void setTotalStudents(int x) {
        totalStudents = x;
    }

    // Other miscellaneous methods.

    public int reportTotalEnrollment() {
        System.out.println("Total Enrollment:  " + getTotalStudents());
    }

    public void incrementEnrollment() {
        setTotalStudents(getTotalStudents() + 1);
    }

    // etc.
}
```

This would be a less-than-desirable design approach, however, because client code would have to invoke the `incrementEnrollment` method on ***every*** `Student` object in the system any time a ***new*** `Student` was instantiated, to ensure that ***all*** `Students` were in agreement on the total student count:

```
// Client code.

// Create a Student ...
Student s1 = new Student();

// ... and increment the enrollment count.
s1.incrementEnrollment();

// Details omitted ...
```

```
// Later, we create another Student ...
Student s2 = new Student();

// ... and have to remember to increment the enrollment count for BOTH.
s1.incrementEnrollment();
s2.incrementEnrollment();

// More details omitted ...

// Still later, we create yet another Student ...
Student s3 = new Student();

// ... and have to remember to increment the enrollment count for
ALL THREE!
s1.incrementEnrollment();
s2.incrementEnrollment();
s3.incrementEnrollment();
// Phew!
```

Fortunately, there is a simple solution: we can designate `totalStudents` to be what is known as a **static variable** of the `Student` class through use of the `static` keyword:

```
public class Student {
    // totalStudents is now declared to be a static attribute.
    private static int totalStudents;
    // Other attribute details omitted.

    // The next three methods are unchanged from the previous version of
    Student.

    public int getTotalStudents() {
      return totalStudents;
    }

    public void setTotalStudents(int x) {
      totalStudents = x;
    }

    public int reportTotalEnrollment() {
        System.out.println("Total Enrollment:  " + getTotalStudents());
    }
```

```
    // This method has been declared to be static.
    public static void incrementEnrollment() {
        setTotalStudents(getTotalStudents() + 1);
    }
}
```

Static variables are also casually referred to as "static attributes," but since I prefer the notion of an "attribute" as "a data item belonging to an individual object," I generally favor the nomenclature of "static variables" instead.

This causes the `totalStudents` variable to be associated with the `Student` class as a whole, as represented conceptually in Figure 7-8.

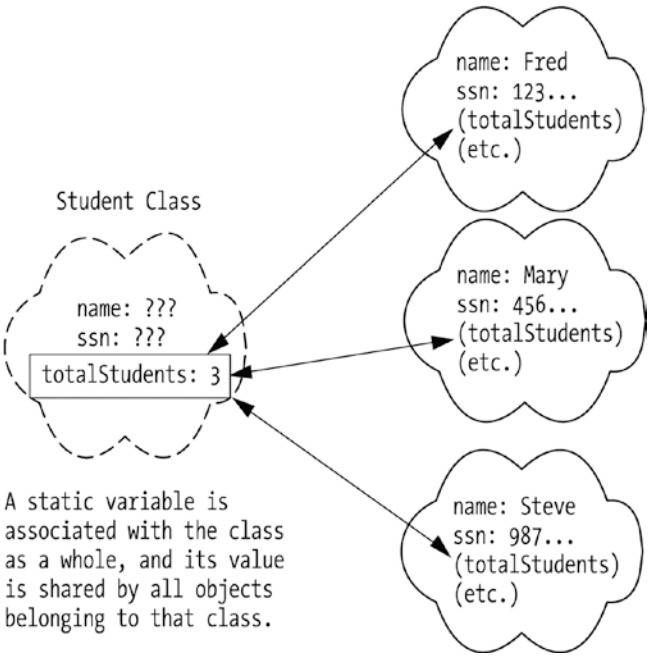


Figure 7-8. *Static variables are associated with a class as a whole*

This enables us to simplify our client code as follows:

```
// Client code.

// Create three Students, incrementing the enrollment each time
```

```
// FOR THAT STUDENT ONLY.

Student s1 = new Student();
s1.incrementEnrollment();

// Later ...
Student s2 = new Student();
s2.incrementEnrollment();

// Still later ...
Student s3 = new Student();
s3.incrementEnrollment();
```

Each time we invoke `incrementEnrollment` for ***one*** `Student`, the others will benefit, because they all share the ***same*** `totalStudents` value. To demonstrate this, let's ask each `Student` object to report on total enrollment:

```
s1.reportTotalEnrollment();
s2.reportTotalEnrollment();
s3.reportTotalEnrollment();
```

Here's the output (all `Students` agree!):

```
Total Enrollment:  3
Total Enrollment:  3
Total Enrollment:  3
```

In fact, since this variable is effectively associated with the `Student` class as a whole, we can even ask the `Student` ***class*** to `reportTotalEnrollment`, and we'll get the same output:

```
Student.reportTotalEnrollment();
```

Here's the output:

```
Total Enrollment:  3
```

Note that we're using dot notation to talk to a class as a whole vs. talking to individual objects. This is only possible, however, if the method in question—in this case, `reportTotalEnrollment`—is declared to be a **static method**. Before we discuss static methods, however, let's look at one final improvement to the design of our `Student` class.

A Design Improvement: Burying Implementation Details

Since we want to increment `totalStudentCount` whenever we create a new `Student`, we can insert such logic into the `Student` class's constructor(s) to do so automatically

```
public class Student {
    private static int totalStudents;
    // Other details omitted.

    // Constructor.
    public Student(...) {
        // Details omitted.
        // Automatically increment the student count every time we
        // instantiate a new Student.
        totalStudents++;
    }

    // etc.
}
```

thereby eliminating the need for an `incrementEnrollment` method. This makes our client code much more concise:

```
// Client code.
// Create three Students, AUTOMATICALLY incrementing the
totalStudents value
// each time.
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();

s1.reportTotalEnrollment();
s2.reportTotalEnrollment();
```

```
s3.reportTotalEnrollment();
Student.reportTotalEnrollment();
```

Here's the output:

```
Total Enrollment:  3
Total Enrollment:  3
Total Enrollment:  3
Total Enrollment:  3
```

It also makes our client code ***foolproof***: we no longer run the risk of forgetting to invoke `incrementEnrollment` explicitly after creating each `Student`. As mentioned before, ***whenever possible, it's desirable to bury such implementation details inside of a class, to lessen the burden on—and hence to lessen the likelihood for logic errors in—client code***.

Static Methods

Just as static variables are associated with a class as a whole vs. with a specific individual object, **static methods** are in turn methods that may be ***invoked*** on a class as a whole.

Let's declare all of the methods related to the `totalStudents` variable—namely, `getTotalStudents`, `setTotalStudents`, and `reportTotalEnrollment`—to be `static`:

```
public class Student {
    private static int totalStudents;
    // Other details omitted.

    // Constructor.
    public Student(...) {
        // Details omitted.
        // Automatically increment the student count every time we
        // instantiate a new Student.
        totalStudents++;
    }
```

```
    // The following three methods are now static methods; note,
    however, that
    // the method BODIES are UNCHANGED from when these were nonstatic
    methods.

    public static int getTotalStudents() {
        return totalStudents;
    }

    public static void setTotalStudents(int x) {
        totalStudents = x;
    }

    public static int reportTotalEnrollment() {
        System.out.println("Total Enrollment:  " + getTotalStudents());
    }

    // etc.
}
```

Static methods can either be invoked on a ***class as a whole***

```
Student.reportTotalEnrollment();
```

or on an ***object belonging to the class*** for which it is defined

```
s.reportTotalEnrollment();
```

and the effect will be the same.

Restrictions on Static Methods

Note that there is an important restriction on static methods: such methods are not permitted to access ***non***static features of the class to which the methods belong. Before we discuss the rationale for this, let's consider a specific example.

If we were to attempt to write a static `print` method for our `Student` class that in turn accessed the ***non***static `getName` method of `Student`, the compiler would prevent us from doing so. Here's the proposed code:

```
public class Student {
  // Two variables -- one nonstatic, one static.
  private String name;
  private static int totalStudents;
  // etc.

  // We declare accessor methods for both variables:
  // STATIC get/set methods for STATIC variable "totalStudents" ...
  public static int getTotalStudents() { … }
  public static void setTotalStudents(int x) { … }
  // ... and NONstatic get/set methods for NONstatic attribute "name".
  public String getName() { … }
  public void setName(String n) { … }

  // Another static method.
  public static void print() {
    // A static method may NOT access NONstatic features
    // such as "getName()" -- the following line won't compile.
    System.out.println(getName() + " is one of " + getTotalStudents() +
                       "students.");
  }
}
```

The compiler would generate the following error message regarding the `println` statement in the `static print` method:

```
non-static method getName() cannot be referenced from a static context
```

Why is this? As we discussed in Chapter 3, classes are empty templates as far as attributes are concerned; it's not until we instantiate an object that its attribute values get filled in. If a static method is invoked on a class as a whole and that method is in turn to try to access the value of an attribute, the value of that attribute would be undefined for the class, as illustrated conceptually in Figure 7-9. Since our `static print` method invokes the nonstatic `getName` method, which in turn accesses the (nonstatic) variable `name`, we're precluded from calling the `getName` method from `print`.

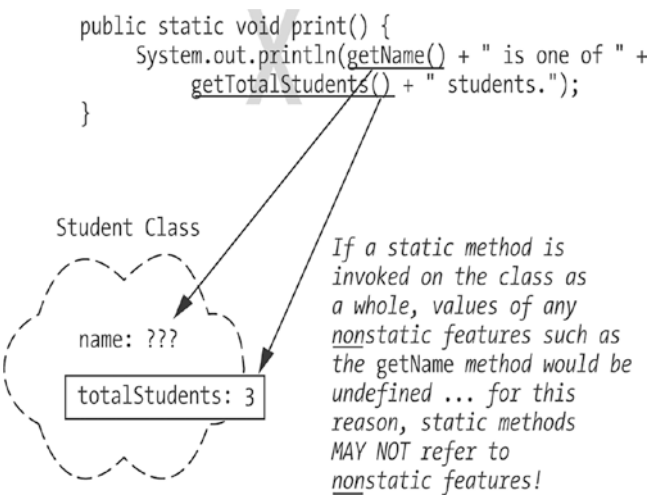


Figure 7-9. *Nonstatic attribute values are undefined in the context of a class*

Another restriction on static methods is that they may not be declared to be `abstract`:

```
public class Student {
  // This won't compile.
  public abstract static void incrementEnrollment();
}
```

The following compiler error would result:

```
illegal combination of modifiers: abstract and static
```

Utility Classes

We can take advantage of static features to design **utility classes**, which are classes that provide convenient ways of performing frequently used behaviors without having to instantiate an object to perform such behaviors. Such classes are often comprised wholly of static methods and public static variables.

The Java language includes several predefined utility classes. One example of such a class is the `Math` class of the `java.lang` package. The `Math` class declares a variety of static methods to compute trigonometric, exponential, logarithmic, and power functions; to

round numeric values; and to generate random numbers. We saw the use of one such static method—`Math.sqrt()`—in an example in Chapter 6:

```
squareRoot[i] = Math.sqrt(i);
```

The mathematical constant π is declared as a `public static` attribute of the `Math` class named `Math.PI` and can be accessed from client code as follows:

```
// Compute the area of a circle.
double area = Math.PI * radius * radius;
```

In order to prevent client code from ***modifying*** the values of such "constants"

```
Math.PI = 0.0;  // Whoops!  This isn't good!
```

utility classes make use of **final variables**, as discussed in the following section.

Chapter 6 mentioned that the Java language provides a different class to serve as a "wrapper" for each of the eight distinct primitive types: `Integer`, `Float`, `Double`, `Byte`, `Short`, `Long`, `Boolean`, and `Character`. We'll talk about their roles as ***utility classes*** in Chapter 13.

The final Keyword

The Java `final` keyword can be applied to variables, methods, and classes as a whole. A **final variable** is a variable that can be assigned a value only once in a program; after that first assignment, the variable's value cannot be changed. We declare such a variable by placing the `final` keyword just before the type of the variable, as follows:

```
public class Example {
    // A static variable can be declared to be final ...
    public static final Student x;

    // ... as can a (nonstatic) attribute.
    private final int y;

    public void someMethod() {
        // Even a local variable may be declared to be final.
        final int z;
        // etc.
```

Whereas a ***local*** `final` variable can be assigned a value separately from its declaration, we cannot do so for other final ***variables***; that is, in the following expanded example, we'll get compilation errors on the lines so marked:

```
public class Example {
    // A static variable can be declared to be final ...
    public static final Student x;

    // ... as can a (nonstatic) attribute.
    private final int y;

    public void someMethod() {
        // Even a local variable may be declared to be final.
        final int z;

        // However, whereas we ARE permitted to assign a local
        // final variable a value in a method separately from its
        // declaration ...
        z = 3;

        // .. we CANNOT do so for final static variables or for attributes.
        x = new Student();  // Compilation error!
        y = 2;  // Compilation error!
    }

    // etc.
```

The compiler will generate the following error messages:

```
cannot assign a value to final variable x
x = 1;
^

cannot assign a value to final variable y
y = 2;
^
```

To avoid such a problem, we must assign values to ***class and instance*** final variables at the time that we declare them:

```
public class Example {
    // Assign values to static final variables/final attributes at the
    // same time that we declare them.
    public static final int x = 1;
    private final int y = 2;
    // etc.
```

Going back to our `Math` class example, we can now see that if `Math.PI` is declared to be a ***final*** static variable, client code will be prohibited from modifying its value, as it should be.

As another example, recall from our discussion of arrays in Chapter 6 that arrays have a public `length` attribute whose value represents the capacity of the array:

```
int[] x = new int[9];
for (int i = 0; i < x.length; i++) { … }
```

As it turns out, the `length` attribute is declared to be `final` so that we are prevented from changing its value programmatically. Continuing the preceding example, the following won't compile:

```
// Let's try to enlarge the array!
x.length = 12;  // this won't compile
```

The compilation error that is produced is as follows:

```
cannot assign value to final variable length
x.length = 12;
  ^
```

Public Static Final Variables and Interfaces

As stated earlier in the chapter, interfaces are not permitted to declare variables, but for one exception: as it turns out, interfaces are allowed to declare `public static final` variables to serve as ***global constants***—that is, constant values that are in scope and hence accessible throughout an entire application.

Let's say, for example, that when a university administrator hires a new professor, they must designate whether the professor is going to be working full-time or part-time. If we wish, we can declare `public static final` variables of the `Administrator` interface to symbolically represent these two values, as shown in the following code, to be passed in as argument values to the `hireProfessor` method:

```
public interface Administrator {
  // Defining symbolic values to use as argument values for the second
  parameter of
  // the hireProfessor method.
  public static final int FULL_TIME = 1;
  public static final int PART_TIME = 2;

  // Valid values for workStatus are FULL_TIME (1) or PART_TIME (2).
  public boolean hireProfessor(Professor p, int workStatus);

  // Other method headers declared; details omitted.
}
```

This enables client code to take advantage of such values when invoking the `hireProfessor` method, as follows:

```
// Client code.
Administrator pAdmin = new Professor();
Professor p = new Professor();

// Hire p as a full-time faculty member.
pAdmin.hireProfessor(p, Administrator.FULL_TIME);
```

The naming convention for a `public static final` variable of either a class or an interface is rather unusual: we traditionally use all uppercase characters to name them and hence use underscore characters (_) to visually separate words in multiword names—for example, `FULL_TIME` and `PART_TIME`. This explains why the `Math` class's `public static final PI` attribute is written in all uppercase—`Math.PI` vs. `Math.pi`, for example.

In Chapter 13, you'll learn about **enum**(eration)s, a language feature that provides a more elegant way to accomplish the same purpose.

As mentioned earlier, the `final` keyword may also be applied to methods and to classes as a whole:

- A method declared to be `final` cannot be overridden in a subclass:

```
public class Person {
    // Details omitted.

    public final int computeAge() { ... }

    // etc.
}
```

- A class declared to be final cannot be subclassed:

```
// No subclasses may be derived from this class.
public final class PhDStudent extends GraduateStudent { ... }
```

Numerous predefined Java classes are declared to be `final`. One such example is the `java.lang.String` class.

Custom Utility Classes

We can take advantage of the same techniques used to create predefined Java utility classes like the `Math` class to create our own ***custom*** utility classes. For example, suppose that we are going to have a frequent need to do temperature conversions from degrees Fahrenheit to degrees Centigrade and vice versa. We could invent a utility class called `Temperature` as follows:

```
// A utility class to provide F=>C and C=>F conversions.
public class Temperature {
  public static double FahrenheitToCentigrade(double tempF) {
    return (tempF - 32.0) * (5.0/9.0);
  }

  public static double CentigradeToFahrenheit(double tempC) {
    return tempC * (9.0/5.0) + 32.0;
  }
}
```

Then, to use this class, we'd simply write client code as follows:

```
double degreesF = 212.0;
double degreesC = Temperature.FahrenheitToCentigrade(degreesF);
System.out.println("A temperature of " + degreesF + " degrees F = " +
                   degreesC + "degrees C");
```

This would give us the following output:

```
A temperature of 212.0 degrees F = 100.0 degrees C
```

We might even wish to include some commonly used constants—say, the boiling and freezing points of water in both F and C terms—as `public static final` variables in our utility class:

```
// A utility class to provide F=>C and C=>F conversions.
public class Temperature {
  // We've added some public static final variables.
  public static final double FAHRENHEIT_FREEZING = 32.0;
  public static final double CENTIGRADE_FREEZING = 0.0;
  public static final double FAHRENHEIT_BOILING = 212.0;
  public static final double CENTIGRADE_BOILING = 100.0;

  public static double FahrenheitToCentigrade(double tempF) {
    // We can utilize our new attributes in our method code.
    return (tempF  - FAHRENHEIT_FREEZING) * (5.0/9.0);
  }

  public static double CentigradeToFahrenheit(double tempC) {
    // Ditto.
    return tempC * (9.0/5.0) + FAHRENHEIT_FREEZING;
  }
}
```

We could then take advantage of these constants in our client code, as well:

```
Soup s = new Soup("chicken noodle");

// Bring the soup to a boil.
if (s.getTemperature() < Temperature.FAHRENHEIT_BOILING) {
  s.cook();
}
```

Because all of the features of the `Temperature` class are `static`, we need never instantiate a `Temperature` object in our application.

Summary

Congratulations! You've made it through all of the major object technology concepts that you'll need to know for the rest of the book, and you've learned a great deal of Java syntax in the process. Please make sure that you're comfortable with these concepts before proceeding, as they form the foundation for the rest of your object learning experience:

- These same concepts will be reinforced when you learn how to model a problem in Part 2.
- They will be reinforced yet again when you learn how to render a model as Java code in Part 3.

In this chapter, you've learned that

- Different objects can respond to the same exact message in different class-specific ways, thanks to an OO language feature known as ***polymorphism***.
- ***Abstract classes*** are useful if you want to prescribe common behaviors among a group of (derived) classes without having to go into details about ***how*** those behaviors should be performed. That is, you specify ***what*** services an object must be able to perform—the messages that an object must be able to respond to, as defined by method headers—without programming method bodies.

- ***Interfaces*** are an even more abstract way to prescribe behaviors in that they may only declare abstract methods (and constants).
- By implementing multiple interfaces, a class of objects may take on multiple roles in an application.
- Using interface types when declaring attributes, method return types, and parameters to methods makes user-defined classes much more versatile and robust to requirements changes.
- ***Static variables*** may be used to enable an entire class of objects to share data and can be manipulated on a class vs. on specific objects/instances of that class via ***static methods***.
- We can take advantage of static methods along with public static final variables to create ***custom utility classes***.

Reflecting back on the home construction example from the Introduction to this book, you now know all about the unique properties of "blue stars" (objects) and why they are superior construction materials. But you still need to learn how to lay out a blueprint for how to use them effectively in building an application—you'll learn how to do so in Part 2.

EXERCISES

1. Test yourself. Run through the following list of OO terms—some formal, some informal—and see if you can define each in your own words without referring back to the text:

Abstract class
Abstract method
Abstraction
Accessor method
Aggregation
Ancestor class
Attribute signature
Association
Attribute
Behavioral relationship
Binary association
Class
Class hierarchy
Classification
Client code
Collection
Collections framework
Composite class
Concrete class
Constructor
Delegation
Dictionary (type of collection)
Encapsulation
Final variable
Generalization
"Get" method
Handle
Implemented method
Information hiding
Inheritance
Instance
Instantiation
Interface
Local variable
Link
Message
Method
Method header
Method signature
Modeling
Multiple inheritance
Multiplicity
Object (in the software sense)
Operation
Ordered list
Overloading
Overriding
Polymorphism
Predefined type
Primitive type
Private accessibility
Protected accessibility
Public accessibility
Reference type
Reference variable
Root (of a class hierarchy)
Service
Set (type of collection)
"Set" method
Sibling class
Specialization
State (of an object)
Static attribute
Static method
Static variable
Subclass
Superclass
User-defined type

2. Which attributes, belonging to which SRS classes, might be well suited to being declared as static?

3. Which attributes, belonging to which Prescription Tracking System classes (as described in the Appendix), might be well suited to being declared as static?

4. It has been argued that the ability to implement multiple interfaces in the Java language eliminates the need for multiple inheritance support. Do you agree or disagree? Why or why not? Can you think of any ways in which implementing multiple interfaces "falls short" as compared with true multiple inheritance?

5. The following client code scenarios would cause compilation errors. Can you explain why this is so in each case? Try answering this question ***without*** compiling the code. Be as precise as possible as to the reasons—they may not be as obvious as first meets the eye!

Assume that `Professor` and `Student` are both classes that implement the `Teacher` interface.

Scenario #1:

```
Professor p;
Student s = new Student();
Teacher t;

t = s;
p = t;
```

Scenario #2:

```
Professor p = new Professor();
Student s;
Teacher t = new Student();

s = t;
```

6. [*Coding*] Test your answers to Exercise 5 by coding simple versions of the `Professor`, `Student`, and `Teacher` types plus a main class/method to house your client code.

CHAPTER 8

The Object Modeling Process in a Nutshell

Let's look in on the home builder whom we met in the Introduction to this book. They've just returned from a seminar entitled "Blue Stars: A Builder's Dream Come True." They now know about the unique properties of blue stars, and they appreciate why they are superior construction materials—just as you've learned about the unique properties of software objects as application "construction materials." But they are still inexperienced with actually using blue stars in a construction project; in particular, they don't yet know how to develop a blueprint suitable for a home that is to be built from blue stars. And we still need to see how to develop a "blueprint" for a software system that is to be constructed from objects. This is the focus of Part 2 of this book.

In this chapter, you'll learn

- The goals and philosophy behind object modeling
- How much flexibility we have in terms of selecting or devising a modeling methodology
- The pros and cons of object modeling software tools

The "Big Picture" Goal of Object Modeling

Our goal in object modeling is to render a precise, concise, understandable object-oriented model, or "blueprint," of the system to be automated. This model will serve as an important tool for communication:

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_8

- ***To the future users of the system that we are about to build, an object model communicates our understanding of the system requirements***. Having the users review and approve the model will ensure that we get off on the right foot with a project, for a mistake in judgment at the requirements analysis stage can prove much more costly to fix—by orders of magnitude—than if such a misunderstanding is found and corrected when the system is still just a "gleam in the user's eye."
- ***To the software development team, an object model communicates the structure and function of the software that needs to be built in order to satisfy those requirements***. This benefits not only the software engineers themselves but also the folks who are responsible for quality assurance, testing, and documentation.
- Long after the application is operational, an object model lives on as a "schematic diagram" ***to help the myriad folks responsible for supporting and maintaining an application to understand its structure and function.***

Of course, this last point is true only if the object model accurately reflects the system as it was actually built, not just as it was originally conceived. The design of complex systems invariably changes during their construction, so care should be taken to keep the object model up to date as the system is built.

Modeling Methodology = Process + Notation + Tool

According to Webster's Dictionary, a **methodology** is

A set of systematic procedures used by a discipline [to achieve a particular desired outcome].

A modeling methodology, OO or otherwise, ideally involves three components:

- ***A process***: The "how to" steps for gathering the requirements and determining the abstraction to be modeled
- ***A notation***: A graphical "language" for communicating the model

- ***A tool***: An automated way of rendering the notation, typically in "drag-and-drop" fashion

Although these constitute the ideal components of a modeling methodology, they are not all of equal importance:

- Adhering to a sound ***process*** is certainly critical.
- However, we can sometimes get by with a narrative text description of an abstraction without having to resort to portraying it with formal graphical ***notation***.
- And, when we ***do*** choose to depict an abstraction formally via a graphical notation, it isn't mandatory that we use a specialized ***tool*** for doing so.

In other words, following an organized process is the most critical aspect of object modeling; using a particular notation is important, but less so; and our choice of a particular tool for rendering the model is the least important aspect of the three (see Figure 8-1).

Figure 8-1. *Of the three aspects of a methodology, a sound process is by far the most important*

Many important contributions in the form of new processes, notations, and tools have been made in the OO methodology arena over the years by numerous well-known methodologists. In some sense, if you're just getting into objects for the first time now, you're fortunate, because you managed to avoid the "methodology wars" that raged for many years as methodologists and their followers argued about what were in some cases esoteric details.

Here is a partial list of contributions made in the object methodology arena over the past few decades; the list is in no particular order:

- ***James Rumbaugh et al.***: The Object Modeling Technique (OMT)
- ***Grady Booch***: The Booch Method

- ***Sally Shlaer and Stephen Mellor***: Emphasis on state diagrams
- ***Rebecca Wirfs-Brock et al.***: Responsibility-driven design, Class-Responsibilities-Collaborators (CRC) cards
- ***Bertrand Meyer***: The Eiffel programming language, the notion of programming by contract
- ***James Martin and James Odell***: Retooling of their functional decomposition methodologies for use with OO systems
- ***Peter Coad and Edward Yourdon***: As in the preceding entry
- ***Ivar Jacobson***: Use cases as a means of formalizing requirements
- ***Derek Coleman et al. (HP)***: The Fusion Method
- ***Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the "Gang of Four")***: Design pattern reuse

In recent years, there was a major push in the industry to meld the best ideas of competing methodologies into a single approach, with particular emphasis placed on coming up with a universal modeling notation. The resultant notation, known as the **Unified Modeling Language** (**UML**), represents the collaborative efforts of three of the leaders in the OO methodology field—James Rumbaugh, Grady Booch, and Ivar Jacobson—and has become the industry-standard object modeling notation. (You'll learn the basics of UML in Chapters 10 and 11.)

Along with UML, these three gentlemen—known affectionately in the industry as the "Three Amigos"—have also contributed heavily to the evolution of an overall methodology known as the **Rational Unified Process** (**RUP**), a full-blown software development methodology encompassing modeling, project management, and configuration management workflows. But I'm not going to dwell on the details of this particular methodology in this book, because it isn't my intention to teach you any one ***specific*** methodology in great detail. By learning a sound, ***generic*** process for object modeling, you'll be armed with the knowledge you need to read about, evaluate, and select a specific methodology such as RUP or to craft your own hybrid approach by mixing and matching the processes, notations, and tool(s) from various methodologies that make the most sense for your organization.

As for modeling tools, you don't need one, strictly speaking, to appreciate the material presented in this book. But I've anticipated that you'll likely want to get your "hands dirty" with a modeling tool. Because of this, I include a general discussion of modeling tool pros and cons a bit later in this chapter.

It's important to keep in mind that a methodology is but a means to an end, and it's the ***end***—a usable, flexible, maintainable, reliable, and functionally correct software system, along with thorough, clear supporting documentation—that we care most about when all is said and done.

To help illustrate this point, let's use a simple analogy. Say that our goal is to cheer people up. We decide to hand-draw (process) a smiley face (an abstraction of the desired behavior, rendered with a graphical notation) with a pencil (tool), as shown in Figure 8-2.

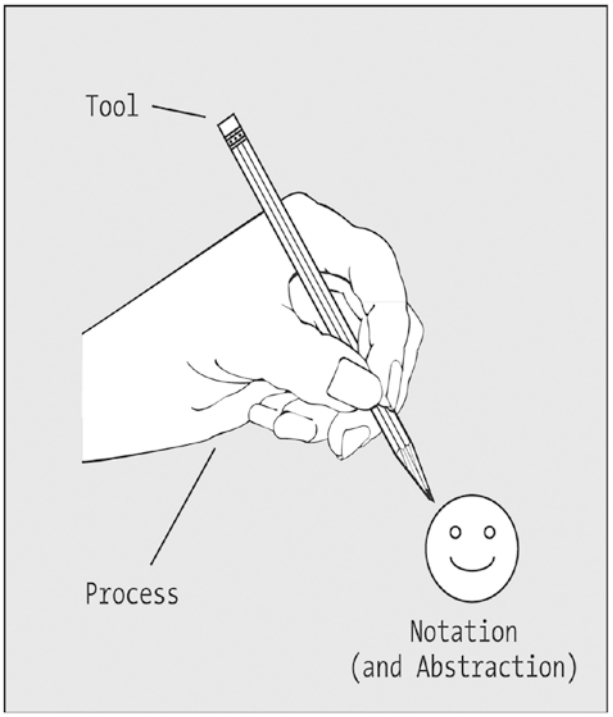


Figure 8-2. *A methodology encompasses process, notation, and tools*

After we're done, we put our paintbrush away, hang our smiley face picture on the wall, and go about our business. A few days go by, and we note that people are indeed cheered up by our picture, and so our original goal has been achieved. In hindsight, we could have accomplished this same goal using

- A variety of "processes"—hand drawing, rubber stamping, cutting pictures from a magazine
- A variety of "notations"—the graphical notation of a smiley face or a cartoon or the narrative text of a joke or sign
- A variety of "tools"—a pen, a pencil, a paintbrush, a crayon

Now, back to our homebuilding analogy. Long after the architect and construction crew have left a building site, taking their equipment and tools with them, the house that they have built will remain standing as a testimonial to the quality of the materials they used, how sound a construction approach they employed, and how elegant a blueprint they had to start with. The blueprint will come in handy later on when the time comes to remodel or maintain the home, so we certainly won't throw it away; but the "livability" and ease/affordability of maintaining the home will be the primary measure of success.

The same is true for software development: the real legacy of a software development project is the resultant software system, which is, after all, the reason for using a methodology to produce a model in the first place. We must take care to avoid getting so caught up in debating the relative merits of one methodology vs. another that we fail to produce useful software. As you can see in Figure 8-3, there are ***many*** paths to the same destination.

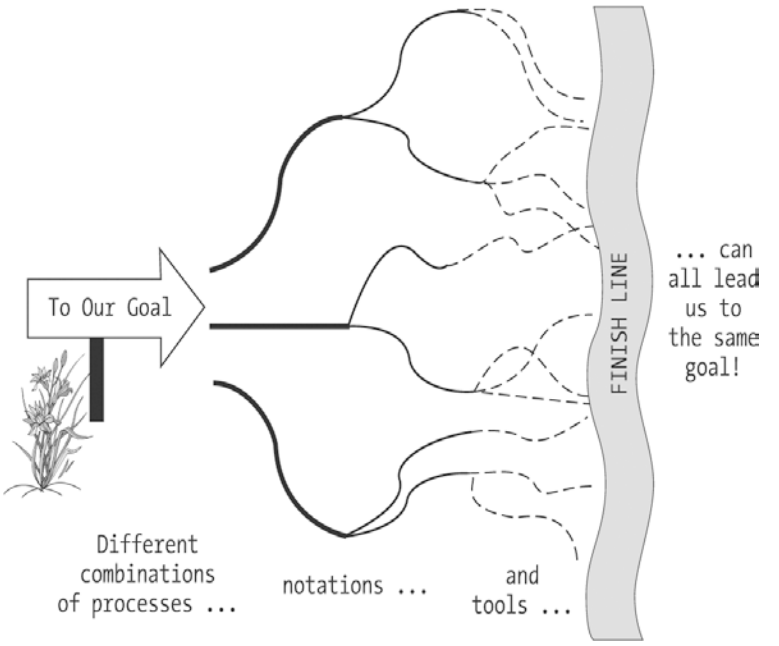


Figure 8-3. *Many different approaches can serve us well when building software*

My Recommended Object Modeling Process, in a Nutshell

I present here a basic preview of the modeling process that I advocate and that I'm going to illustrate in depth throughout the remainder of Part 2 of the book:

- Begin by obtaining or writing a narrative problem statement, similar to the Student Registration System (SRS) problem statement presented at the beginning of the book or the alternative case study problem statements included as the Appendix. Think about the different categories of users that will be interacting with the system, and the various situations in which they'll each use it, to make sure that you uncover any not-so-obvious requirements that may have been missed. (I'll discuss a formal technique for doing this—known as **use case modeling**—in Chapter 9.)
- Handle the data side of the application by identifying the different classes of "real-world" objects that your application will need to be concerned with, and determine how these interrelate. (I'll illustrate the process of creating a formal **class diagram** in Chapter 10.)
- Handle the functional side of the application by studying how objects need to collaborate to accomplish the system's mission, determining what behaviors/responsibilities will be required of each class. (I'll illustrate the process of modeling the behavioral aspects of an OO system in Chapter 11.)
- Test the model to ensure that it does indeed meet all of the original requirements. (I'll discuss testing models in Chapter 12.)

You'll see plenty of examples of each of these techniques in the chapters to follow, and you'll get an opportunity to practice these techniques based on the exercises suggested at the end of each chapter. Armed with a solid model of the SRS, you'll then be ready to render the model into Java code, which is the subject of Part 3 of the book.

Note that these process steps need not be performed in strictly sequential fashion. In fact, as you become comfortable with each of the steps, you may find yourself carrying some of them out in parallel or in shuffled order. For example, contemplating the behavioral aspects of a model may bring to light new data requirements. In fact, for

all but the most trivial models, it's commonplace to cycle through these steps multiple times, "dialing in" increased levels of understanding, hence more detail in the model and supporting documentation, with each iteration.

It's also important to note that the formality of the process should be adjusted to the size of the project team and the complexity of the requirements. If we separate the ***form*** of using a methodology from the ***substance*** of what that methodology produces in the way of **artifacts**—models, documentation, code, and so on—then a good rule of thumb is that a project team should spend no more than 10–20 percent of its time on ***form*** and 80–90 percent on ***substance***. If the team finds itself spending so much time on form that little or no progress is being made on substance, it's time to re-evaluate the methodology and its various components, to see where simplifying adjustments or improvements to efficiency may be made.

Thoughts Regarding Object Modeling Software Tools

It's worthwhile to spend a little bit of time talking about the pros and cons of using an object modeling software tool. For purposes of learning how to produce models, a generic drawing tool such as PowerPoint may be good enough; for that matter, you may simply want to sketch your models using paper and pencil. But getting some hands-on experience with using a tool specifically designed for object modeling will better prepare you for your first "industrial-strength" project, so you may wish to acquire one before embarking upon the next chapter.

You can find information about various object modeling software tools, including links to free or evaluation copies of software, by conducting an Internet search for "object modeling tools" or "UML tools."

I make it a practice not to mention specific tools, vendors, versions, etc. in this book, as they change much too rapidly. As soon as a software product is mentioned in print, it's virtually guaranteed that it will either change names, change vendors who market it, or disappear completely.

Object modeling tools fall under the general heading of **Computer-Aided Software Engineering**, or **CASE**, **tools**. CASE tools afford us many advantages, but aren't without their drawbacks.

Advantages of Using CASE Tools

There are many arguments in favor of using CASE tools; several of the more compelling are as follows.

Ease of Use

CASE tools provide a quick drag-and-drop way to create visual models. Rather than trying to render a given notation with a generic drawing tool, where your basic drawing components are simple lines, arrows, text, boxes, and other geometric shapes, CASE tools provide one or more palettes of prefabricated graphical components specific to the supported notation. For example, you can drag and drop the graphical representation for a class rather than having to painstakingly fabricate it from simpler drawing components.

Added Information Content

CASE tools produce "intelligent" drawings that enforce the syntax rules of a particular notation. This is in contrast to a generic drawing package, which will pretty much let you draw whatever you like, whether it adheres to the notational syntax or not.

The controls imposed by a CASE tool can be a mixed blessing: on the plus side, they will prevent you from making syntactic errors, but as I discuss a little later, they may also prevent you from making desired adjustments to the notation.

Also, information about the classes reflected in a diagram—their names, attributes, methods, and relationships—is typically stored in a repository that underlies the diagram. Most CASE tools provide documentation generation features based upon this repository, enabling you to automatically generate project documentation such as a data dictionary report, a type of report that I'll discuss in Chapter 10. Some tools even allow you to tap into this repository programmatically, should you find a need to do so.

Automated Code Generation

Most CASE tools provide code generation capabilities, enabling you to transition from a diagram to skeletal Java (or other language) code with the push of a button. You may or may not wish to avail yourself of this feature, however, for the following reasons:

- Depending on how much control the CASE tool gives you as to the structure that the generated code takes, the code that is generated will potentially not meet team/corporate standards.
- With most tools, you're unable to edit the generated code externally to the tool, because the tool will then be "unaware" of the changes that you've made, meaning that the next time the code is generated, your changes will be overwritten and obliterated.
- This has implications for reusing code from other projects, as well: make sure that your tool of choice allows you to import and introduce software components that didn't originate within the tool.

It's sometimes better in the end to write your code from scratch, for even though it may take a bit longer at the outset, it often is much easier to manage such code over the lifetime of the project, and you avoid become "enslaved" to a particular modeling tool for ongoing code maintenance. In the worst-case scenario, the tool vendor goes out of business, and you're left with an unsupported product and perhaps unsupportable project.

Project Management Aids

Many CASE tools provide some sort of version control, enabling you to maintain different generations of the same model. If you make a change to your model, but then after reviewing the change with your users decide that you'd prefer to return to the way things were previously, it's trivial to do if version control is in place.

CASE tools also often provide configuration management/team collaboration capabilities, to enable a group of modelers to easily share in the creation of a single model.

Some Drawbacks of Using CASE Tools

CASE tools aren't without their drawbacks, however:

- ***CASE tools can be expensive***. It's not unusual for a high-end CASE tool to cost hundreds or even thousands of dollars per "seat." The ***good news*** is that, in recent years, many shareware/free UML modeling tools have become available, as mentioned earlier in this chapter.

- ***CASE tools can sometimes be inflexible***. I talk about adapting processes, notations, and tools to suit your own needs throughout Part 2 of the book, but tools don't always cooperate. I'll point out in upcoming chapters some specific examples of situations where you might want to bend the notation a little bit, if your CASE tool will accommodate it.
- ***You run the risk of getting "locked into" a particular vendor's product*** if the CASE tool in question can't export your model in a vendor-neutral fashion (e.g., as XML).
- ***It's easy to get caught up with form over substance***! This is true of any automated tool—even a word processor tends to lure people into spending more time on the cosmetics of a document than is warranted, long after the substantive content is rock solid.

Generally speaking, however, the pros of using an OO CASE tool significantly outweigh the cons—consider the cons as "words to the wise" on how to successfully apply such a tool to your modeling efforts.

A Reminder

Although I've said it several times already in this book, it's important to remind you that the process of object modeling is ***language neutral***. I presented Java syntax in Part 1 of the book because the ultimate goal is to make you comfortable with both object modeling and Java programming. In Part 2 of the book, however, we're going to drift away from Java, because we truly are at a point where the concepts you'll be learning are just as applicable to Java as they are to Python or C# any other OO programming language. But never fear—we'll return to Java "big time" in Part 3.

Summary

By far, the most important lesson to take away from this chapter is the following:

Don't get caught up in form over substance!

The model that you produce is only a means to an end… and the process, notation, and tools that you use to produce the model are but a ***means*** to the means to this end. If you get too hung up on which notation to use or which process to use or which tool

to use, you may wind up spinning your wheels in "analysis paralysis." Don't lose sight of your ultimate goal: to build ***usable, flexible, maintainable, reliable, functionally correct software systems.***

EXERCISES

1. Briefly describe the methodology—process, notation, and tool(s)—that you used on a recent software development project. What aspects of this methodology worked well for you and your teammates, and what, in hindsight, do you think could have been approached more effectively?
2. Research one of the object modeling technologies/techniques mentioned in the "Modeling Methodology = Process + Notation + Tool" section earlier in this chapter, and report briefly on the process, notation, and tools involved.

CHAPTER 9

Formalizing Requirements Through Use Cases

When you get ready to leave on a vacation, you may run through a mental or written checklist to make sure that you've properly prepared for your departure. Did you pack everything you need to take? Did you pack ***too much***? Did you arrange to have the appropriate services (newspaper, mail delivery, etc.) stopped? Did you arrange for someone to water the plants and feed your pet rat? Once you depart on your trip, you want to enjoy yourself and know that when you arrive home again, you won't find any disasters waiting for you.

This isn't unlike a software development project: we need to organize a checklist of the things that must be provided for by the system before we embark on its development, so that the project runs smoothly and so that we don't create a disaster (in the form of unmet requirements and dissatisfied customers/users) when the system is delivered.

The art and science of requirements analysis—for it truly is both!—is so extensive a topic that an entire book could be devoted to this subject alone. There is one technique in particular for discovering and rounding out requirements known as **use case modeling**, which is a cornerstone of the Rational Unified Process (RUP) and which warrants your consideration. Use cases aren't strictly an artifact of OO methodologies; they can be prepared for any software system, regardless of the development methodology to be used. However, they made their debut within the software development community in the context of object systems and have gained widespread popularity in that context.

In this chapter, you'll learn

- How we must anticipate all of the different roles that users will play when interacting with our future system
- That we must assume each of the users' viewpoints in describing the services that a software application as a whole is to provide

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_9

- How to prepare use cases as a means of documenting all of the users' collective requirements

I'll also give you enough general background about requirements analysis to provide an appropriate context for use case modeling.

What Are Use Cases?

In determining what the desired functionality of a system is to be, we must seek answers to the following questions:

- ***Who*** (in terms of categories of user) will want to use our system?
- What ***services*** will the system need to provide in order to be of value to each category of user?
- When users interact with the system for a particular purpose, what is their expectation as to the ***desired outcome***?

Use cases are a natural way to express the answers to these questions. Each use case is a simple statement, narrative or graphical in fashion, that describes a particular goal or outcome of the system and who expects that outcome. For example, one goal of the SRS is to "enable a student user to register for a course," and thus we've just expressed our first use case! (Yes, use cases really are that straightforward. In fact, we ***need*** for them to be that straightforward, so that they are understandable by the users/sponsors of the system, as we'll discuss further in a moment.)

Functional vs. Technical Requirements

The purpose of thinking through all of the use cases for a system is to explore the system's functional requirements thoroughly, so as to make sure that a particular category of user, or potential purpose for the system, isn't overlooked. We differentiate between functional requirements and technical requirements as follows.

Functional requirements are those aspects of a system that have to do with how it is to operate or function from the perspective of someone using the system. Functional requirements may in turn be subdivided into

- ***"Goal-oriented" functional requirements***: These provide a statement of a system's purpose without regard to how the requirement will "play out" from the user's vantage point—for example, "The system must be able to produce tailorable reports." Avoid discussing implementation details when specifying goal-oriented requirements**.**
- ***"Look and feel" requirements***: These requirements get a bit more specific in terms of what the user expects the system to look like externally (e.g., how the GUI will be presented) and how the user expects it to behave, again from the user's perspective. For example, we might have as a requirement "The user will click a button on the main GUI, and a confirmation message will appear… " A good practice is to write a **concept of operations** document to serve as a "paper prototype" describing how you envision the future system will look and behave, to stimulate discussion with intended users of the as-yet-to-be-built system before you even begin modeling.

We emphasize goal-oriented functional requirements when preparing use cases**.**

Technical requirements, on the other hand, have more to do with ***how*** a system is to be built internally in order to ***meet*** the functional requirements; for instance, "The system will use the TCP/IP protocol… " or "We will use a dictionary collection as the means for tracking students… " We can think of these as requirements for how programmers should tackle the ***solution***, in contrast to functional requirements, which are a statement of what the ***problem to be tackled*** actually is.

Technical requirements such as these don't play a role in use case analysis.

Although it's certainly conceivable that the users of our system may be technically sophisticated, it's best to express functional requirements in such a way that even a user who knows nothing about the inner workings of a computer will understand them. This helps ensure that technical requirements don't creep into the functional requirements statement, a common mistake made by many inexperienced software developers. When we allow technical requirements to color the functional requirements, they artificially constrain the solution to a problem too early in the development life cycle.

Involving the Users

Because the intended users of a system are the ultimate experts in what they need the system to do, it's essential that they be involved in the use case definition process. If the intended users haven't (as individuals) been specifically defined or recruited (as with a software product that is to be sold commercially), their anticipated needs nonetheless need to be taken into account by identifying people with comparable experience to serve as "user surrogates." Ideally, the users or user surrogates will write some or all of the use cases themselves; at a minimum, you'll interview such people, write the use cases on their behalf, and then get their confirmation that what you've written is indeed accurate.

Use cases are one of the first deliverables/artifacts to emerge in a software development project's life cycle, but also one of the last things to be put to good use in making sure that the system is a success.

They turn out to be quite useful as a basis for writing testing scripts, to ensure that all functional threads are exercised during system and user acceptance testing.

They also lend themselves to the preparation of a **requirements traceability matrix**—that is, a final checklist against which the users can verify that all of their initial requirements have indeed been met when the system is delivered.

Returning to the questions posed at the outset of this section, let's answer the first question—namely, "Who (what categories of user) will want to use our system?"—which in use case nomenclature is known as identifying **actors**.

Actors

Actors represent anybody or anything that will interact with the system after it's built; actors drive use cases. Actors generally fall into two broad categories:

- Human users
- Other computer systems

"Interaction" is generally defined to mean using the system to achieve some result, but can also be thought of as simply (a) providing/contributing information to the system and/or (b) receiving/consuming information from the system.

By ***providing*** information, I mean whether or not the actor inputs substantive information that adds to the collective data stored by the system—for example, a department chairperson defining a new course offering or a student registering their

plan of study. This doesn't include the relatively trivial information that users have to provide to look things up—for example, typing in a student ID to request their transcript.

By ***consuming*** information, I mean whether or not the actor uses the system to obtain information—for example, a faculty user printing out a student roster for a course that they will be teaching or a student viewing their course schedule online.

Identifying Actors and Determining Their Roles

We must create an actor for every different role that will be assumed by various categories of user relative to the system. To identify such roles, we typically turn first to the **narrative requirements specification**, if one exists—that is, a statement of the functional requirements, such as the Student Registration System specification. The only category of user explicitly mentioned by that specification is a student user. So we would definitely consider Student to be one of the actor types for the SRS.

If we think beyond the specification, however, it isn't difficult to come up with other potential categories of user who might also benefit from using the SRS:

- Faculty may wish to get a headcount of how many students are registered for one of the upcoming classes that they are going to be teaching, or they may use the system to post final grades, which in turn are reflected by a student's transcript.
- Department chairs may wish to see how popular various courses are or, conversely, whether or not a course ought to be canceled due to lack of interest on the part of the student body.
- Personnel in the registrar's office may wish to use the SRS to verify that a particular student is projected to have met the requirements to graduate in a given semester.
- Alumni may wish to use the SRS to request copies of their transcripts.
- Prospective students—that is, those who are thinking about applying for admission but who haven't yet done so—may wish to browse the courses that are going to be offered in an upcoming semester to help them determine whether or not the university has a curriculum that meets their interests.
- And so on.

Similarly, since I said that other computer systems can be actors, we might have to build interfaces between the SRS and other existing automated systems at the university, such as

- The Billing System, so that students can be billed accurately based on their current course load
- The Classroom Scheduling System, to ensure that classes to be taught are assigned to rooms of adequate capacity based on the student headcount
- The Admissions System, so that the SRS can be notified when a new student has been admitted and is eligible to register for courses

Of course, we have to make a decision early on as to what the scope of the system we're going to build should be, to avoid "requirements inflation" or "scope creep." To try and accommodate all of the actors hypothesized earlier would result in a massive undertaking that may simply be too costly for the sponsors of the system. For example, does it make sense to provide for potential students to use the SRS to preview what the university offers in the way of courses, or is there a different system—say, an online course catalog of some sort—that is better suited to this purpose? Through in-depth interviews with all of the intended user groups, the scope of the system can be appropriately bounded, and some of the actors that we conceived of may be eliminated as a result.

In our particular case, we'll assume that the sponsors of the SRS have decided that we needn't accommodate the needs of alumni or prospective students in building the system–that is, that we needn't recognize alumni or prospective students as actors. A key point here is that the sponsors decide such things, ***not the programmers***! One responsibility of a software engineer is indeed to identify requirements, and certainly part of that responsibility may include suggesting functional enhancements that the software engineer feels will be of benefit to the user. But the sponsors of the system rightfully have the final say in what actually gets built.

Many software engineers get into trouble because they assume that they "know better" than their clients as to what the users really need. You may indeed have a brilliant idea to suggest, but think of it simply as that—a ***suggestion***—and consider your task as one of either convincing the sponsors/users of its merit or graciously accepting their decision to decline your suggestion.

Note that the same user may interact with the system on different occasions in different roles. That is, a professor who chairs a department may assume the role of a Department Chair actor when they are trying to determine whether or not a course should be cancelled. Alternatively, the same professor may assume the role of a Faculty user when they wish to query the SRS for the student headcount for a particular course that they are teaching.

Diagramming a System and Its Actors

Once we've settled on the actors for our system, we may wish to optionally diagram them. UML notation calls for representing all actors—whether a human user or a computer system—as stick figures and then connecting these via straight lines to a rectangle representing the system, as you see in Figure 9-1.

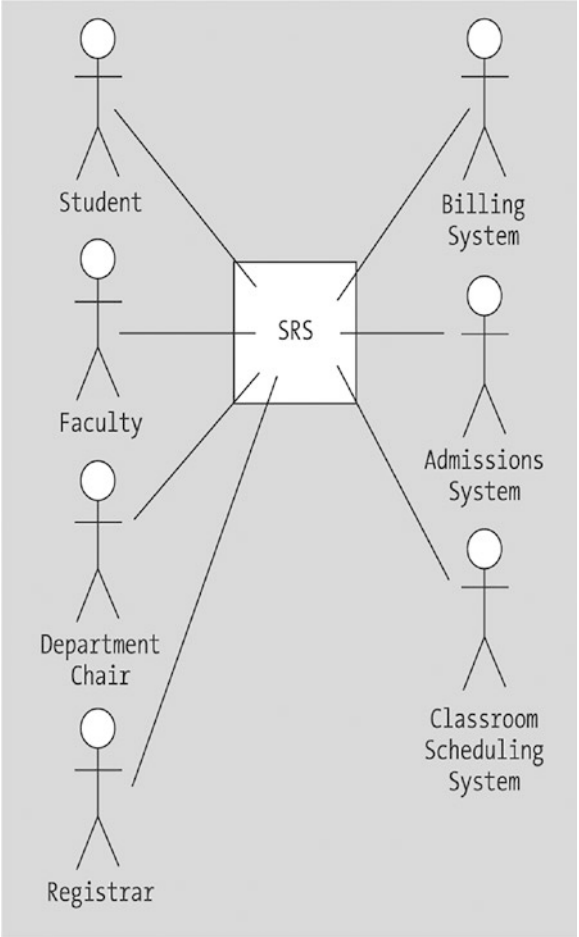


Figure 9-1. *A "proper" UML use case diagram*

This figure appears rather simplistic, and yet, this is a legitimate diagram that might be produced for a project such as the SRS development effort.

I prefer to use a slightly modified version of UML notation, as follows:

- I've extended the use of a rectangle to represent not only the core system but also all actors that are external systems, rather than representing the latter as human stick figures.
- I find that using arrowheads to reflect a directional flow of information—that is, whether an actor provides or consumes information—is a bit more communicative. For example, in the amended version of the notation as follows, I represent a student as both providing and consuming information, whereas a registrar only consumes information.

Note that the registrar does indeed provide information, but not to the SRS directly. They provide information to the Admissions System as to which students are registered at the university; this information then gets fed into the SRS by the Admissions System. So the Admissions System is shown as providing information as an actor to the SRS; but, from the standpoint of the SRS, the registrar is simply a consumer.

With these slight changes in notation, as reflected in Figure 9-2, the use case diagram becomes a much more communicative instrument.

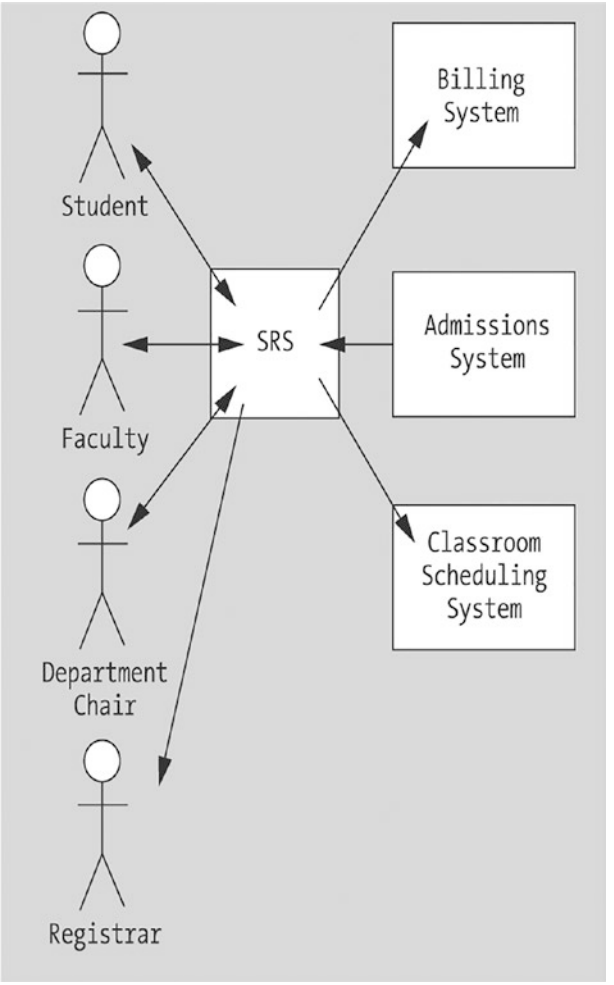


Figure 9-2. *A customized version of use case notation*

Of course, if you do decide to deviate from a widely understood notational standard such as UML, you'll need to follow these steps:

1. Reach consensus among your fellow software developers, to ensure that the team as a whole is speaking the same language.
2. Document and communicate such deviations (along with the notation as a whole) to your customers/users, so that they, too, understand your particular "dialect."

3. Make sure that such documentation is incorporated into the full documentation set for the project, so that future reviewers of the documentation will immediately understand your notational "embellishments."

If you make these enhancements intuitive enough, however, they may just speak for themselves.

Of course, as pointed out in Chapter 8, you'll also need to consider whether the CASE tool you're using, if any, will support such alterations.

Time and again throughout Part 2 of this book, I'll remind you that it's perfectly acceptable to adapt or extend any process, notation, or tool that you care to adopt to best suit your company's or project's purposes; none of these methodology components is "sacred."

Specifying Use Cases

Having made a first cut at what the SRS actors are, we'll next enumerate in what ways the system will be used by these actors—in other words, the use cases themselves.

A use case represents a logical "thread," or a series of cause-and-effect events, beginning with an actor's first contact with the system and ending with the achievement of that actor's goal for using the system in the first place. Note that an actor always initiates a use case; actions initiated by a system on its own behalf don't warrant the development of a use case (although they do warrant expression as either a functional or technical requirement, as defined earlier in the chapter).

Use cases emphasize "what" the system is to do—functional requirements—without concern for "how" such things will be accomplished internally, and they aren't unlike method signatures in this regard. In fact, you can think of a use case as a "behavioral signature" for the system as a whole.

Some example high-level use cases for the SRS might be

- Register for a course.
- Drop a course.
- Determine a student's course load.
- Choose a faculty advisor.
- Establish a plan of study.

- View the schedule of classes.
- Request a student roster for a given course.
- Request a transcript for a given student.
- Maintain course information (e.g., change the course description, reflect a different instructor for the course, etc.).
- Determine a student's eligibility for graduation.
- Post final semester grades for a given course.

Remember that a use case is initiated by an actor, which is why I didn't list other functionality called out by the SRS requirements specification, such as "Notify student by email," as use cases.

We may decompose any one of the use cases into steps, with each step representing a more detailed use case. For example, "Register for a course" may be decomposed into these steps:

1. Verify that a student has met the prerequisites.
2. Check student's plan of study to ensure that this course is required.
3. Check for availability of a seat in the course.
4. (Optionally) Place student on a waitlist.
5. And so forth.

Use cases may be interrelated in parent-child fashion, with more detailed use cases being shared by more than one general use case. For example, the "Request a student roster" and "Post final semester grades" general use cases may both involve the more detailed "Verify that professor is teaching the course in question" use case.

Unfortunately, as is true of all requirements analysis, there is no magical formula to apply in order to determine whether or not you've identified all of the important use cases or all of the actors and/or whether you've gone into sufficient depth in terms of sub–use cases. The process of use case development is iterative; when subsequent iterations fail to yield substantial changes, you're probably finished! Copious interviews and reviews with users, along with periodic team walk-throughs of the use case set as a whole, go a long way in ensuring that nothing important has been missed.

Matching Up Use Cases with Actors

The next important step is to match up use cases with actors. The relationship between actors and use cases is potentially many-to-many in that the same actor may initiate many different use cases and a single use case may be relevant to many different actors. By cross-referencing actors with use cases, we ensure that

- We didn't identify an actor who, in the final analysis, really has no use for the system after all.
- Conversely, we didn't specify a use case that nobody really cares about after all.

For each use case–actor combination, it's useful to determine whether the actor consumes information and/or provides information. Another way to view this aspect of a system is whether actors need write access (providing) to the system's information resources vs. having read-only access (consuming).

If the number of actors and/or use cases isn't prohibitive, a simple table such as Table 9-1 can be used to summarize all of the preceding.

Table 9-1. *A Simple Actor/Use Case Cross-Referencing Technique*

Initiating Actor ==>	Student	Faculty	Billing System	Etc.
Use case:				
Register for a course	Provides info	N/A	N/A	
Post final grades	Consumes info	Provides info	N/A	
Request a transcript	Consumes info	Consumes info	N/A	
Etc.				

To Diagram or Not to Diagram?

The use case concept is fairly straightforward, and hence simple narrative text as we've seen thus far in the chapter is often sufficient for expressing use cases. UML does, however, provide a formal means for diagramming use cases and their interactions with actors. As mentioned earlier, actors (whether people or systems) are represented as

stick figures; use cases are represented as ovals labeled underneath with a brief phrase describing the use case; and the box surrounding the oval(s) represents the system boundaries.

Figure 9-3 shows a sample UML use case diagram. Here, we depict three actors—Student, Faculty, and Registrar—as having occasion to participate individually in the Request Transcript use case.

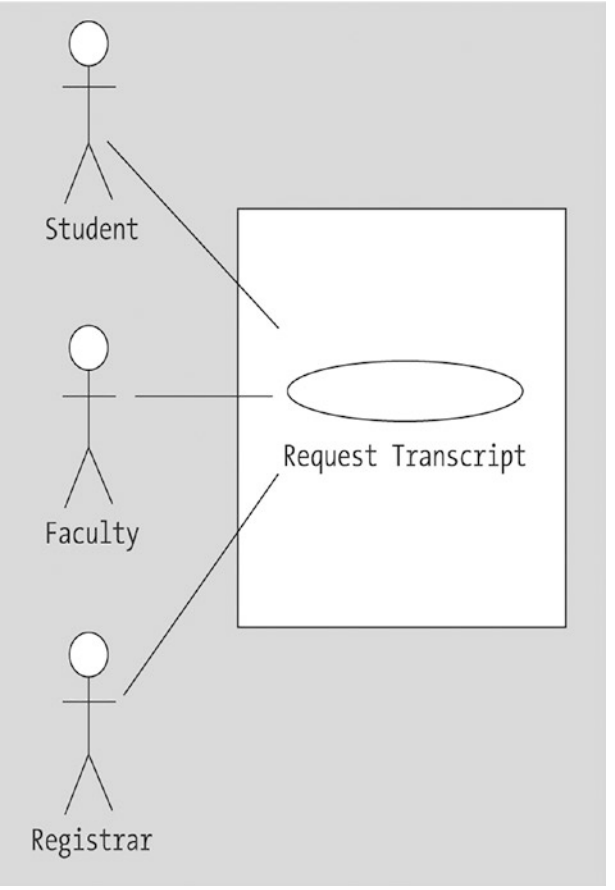


Figure 9-3. *A sample UML use case diagram*

When deciding whether or not to go to the trouble of diagramming your use cases rather than merely expressing them in narrative form, think back to the rationale for producing use cases in the first place: namely, to think through the software development team's understanding of the system requirements and to then communicate to the users/sponsors in order to obtain consensus. It's up to you, your

project team, and your users/sponsors to determine whether diagrams enhance this process or not. If they do, use them; if they don't, go with narrative use case documentation instead.

Once you've documented a system's actors and use cases, whether in text alone or with accompanying diagrams, these become part of the core documentation set defining the problem to be automated. In the next chapter, we'll examine how to use such documentation as a starting point for determining what classes we'll need to create and instantiate as our system "building blocks."

Summary

In this chapter, you've seen that

- Use case analysis is a simple yet powerful technique for specifying the requirements for a system more precisely and completely.
- Use cases are based upon the goal-oriented functional requirements for a system.
- Use cases are used to describe
  - The desired behavior/functionality of the system to be built
  - The external users or systems (known as actors) who avail themselves of these services
  - The interactions between the two

EXERCISES

1. Determine the actors that might be appropriate for the Prescription Tracking System (PTS) case study discussed in the Appendix.
2. For the problem area whose requirements you defined for Exercise 3 in Chapter 1, determine who the appropriate actors might be.
3. Based on the PTS specification in the Appendix, list (a) the use cases that are explicitly called for by the specification and (b) any additional use cases that you suspect might be worth exploring with the future users of the system.

4. Repeat Exercise 3, but in the context of the problem area whose requirements you defined for Exercise 3 in Chapter 1.

5. Create a table mapping the actors you identified for the PTS in Exercise 1 to the use cases you listed for the PTS in Exercise 3, indicating whether a particular actor's participation in a use case is as an information provider or a consumer.

6. Create a table mapping the actors you identified in Exercise 2 to the use cases you listed in Exercise 4, indicating whether a particular actor's participation in a use case is as an information provider or an information consumer.

CHAPTER 10

Modeling the Static/Data Aspects of the System

Having employed use case analysis techniques in Chapter 9 to round out the Student Registration System (SRS) requirements specification, we're ready to tackle the next stage of modeling, which is determining how we're going to meet those requirements in an OO fashion.

We saw in Part 1 of the book that objects form the building blocks of an OO system and that classes are the templates used to define and instantiate objects. An OO model, then, must specify the following:

- ***What types of objects we're going to need to create and instantiate in order to represent the proper abstraction***: In particular, their attributes, methods, and structural relationships with one another. Because these elements of an OO system, once established, are fairly static—in the same way that a house, once built, has a specific layout, a given number of rooms, a particular roofline, and so forth—we often refer to this process as preparing the **static model**.

  We can certainly change the static structure of a house over time by undertaking remodeling projects, just as we can change the static structure of an OO software system as new requirements emerge by deriving new subclasses, inventing new methods for existing classes, and so forth. However, if a structure—whether a home or a software system—is properly designed from the outset, then the need for such changes should arise relatively infrequently over its lifetime and shouldn't be overly difficult to accommodate.

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_10

- ***How these objects will need to collaborate in carrying out the overall requirements, or "mission," of the system***: The ways in which objects interact can change literally from one moment to the next based upon the circumstances that are in effect. One moment, a `Course` object may be registering a `Student` object, and the next, it might be responding to a query by a `Professor` object as to the current student headcount. We refer to the process of detailing object collaborations as preparing the **dynamic model**. Think of this as all of the different day-to-day activities that go on in a home: same structure, different functions.

The static and dynamic models are simply two different sides of the same coin: they jointly comprise the OO "blueprint" that we'll work from in implementing the model layer for an object-oriented Student Registration System application in Part 3 of the book.

In this chapter, we'll focus on building the static model for the SRS, leaving a discussion of the dynamic model for Chapter 11. You'll learn

- A technique for identifying the appropriate classes and their attributes
- How to determine the structural relationships that exist among these classes
- How to graphically portray this information as a **class diagram** using UML notation

Identifying Appropriate Classes

Our first challenge in object modeling is to determine what classes we're going to need as our system building blocks. Unfortunately, the process of class identification is rather "fuzzy"; it relies heavily on intuition, prior modeling experience, and familiarity with the subject area, or **domain**, of the system to be developed. So how does an object modeling novice ***ever*** get started? One tried and true (but somewhat tedious) procedure for identifying candidate classes is to use the "hunt and gather" method: that is, to hunt for and gather a list of all nouns/noun phrases from the project documentation set and to then use a process of elimination to whittle this list down into a set of appropriate classes.

In the case of the SRS, our documentation set thus far consists of the following:

- The requirements specification
- The use case model that we prepared in Chapter 9

Noun Phrase Analysis

Let's perform noun phrase analysis on the SRS requirements specification first, which was originally presented in the Introduction to the book, a copy of which is provided in the following sidebar. All noun phrases are highlighted in bold.

HIGHLIGHTING NOUN PHRASES IN THE SRS SPECIFICATION

We have been asked to develop an ***automated Student Registration System*** (***SRS***) for the ***university***. This ***system*** will enable ***students*** to register online for ***courses*** each ***semester***, as well as track their ***progress*** toward ***completion*** of their ***degree***.

When a ***student*** first enrolls at the ***university***, they use the ***SRS*** to set forth a ***plan of study*** as to which ***courses*** they plan on taking to satisfy a particular ***degree program*** and choose a ***faculty advisor***. The ***SRS*** will verify whether or not the proposed ***plan of study*** satisfies the ***requirements of the degree*** that the ***student*** is seeking.

Once a ***plan of study*** has been established, then, during the ***registration period*** preceding each ***semester***, ***students*** are able to view the ***schedule of classes*** online and choose whichever ***classes*** they wish to attend, indicating the ***preferred section*** (***day of the week*** and ***time of day***) if the ***class*** is offered by more than one ***professor***. The ***SRS*** will verify whether or not the ***student*** has satisfied the necessary ***prerequisites*** for each ***requested course*** by referring to the ***student***'s online ***transcript*** of ***courses completed*** and ***grades received*** (the ***student*** may review their ***transcript*** online at any time).

Assuming that (a) the ***prerequisites*** for the ***requested course(s)*** are satisfied, (b) the ***course(s)*** meet(s) one of the ***student***'s ***plan of study requirements***, and (c) there is ***room*** available in each of the ***class(es)***, the ***student*** is enrolled in the ***class(es)***.

If (a) and (b) are satisfied, but (c) is not, the ***student*** is placed on a ***first-come***, ***first-served waitlist***. If a ***class/section that they were previously waitlisted for*** becomes available (either because some other ***student*** has dropped the ***class*** or because the ***seating capacity*** for the ***class*** has been increased), the ***student*** is automatically enrolled in the ***waitlisted class***, and an ***email message*** to that effect is sent to the ***student***. It is the student's ***responsibility*** to drop the ***class*** if it is no longer desired; otherwise, the student will be billed for the ***course***.

Students may drop a ***class*** up to the ***end*** of the ***first week of the semester in which the class is being taught***.

A simple spreadsheet serves as an ideal tool for recording our initial findings; we enter noun phrases as a single-column list in the order in which they occur in the specification. Don't worry about trying to eliminate duplicates or consolidating synonyms just yet; we'll do that in a moment. The resultant spreadsheet is shown in part in Figure 10-1.

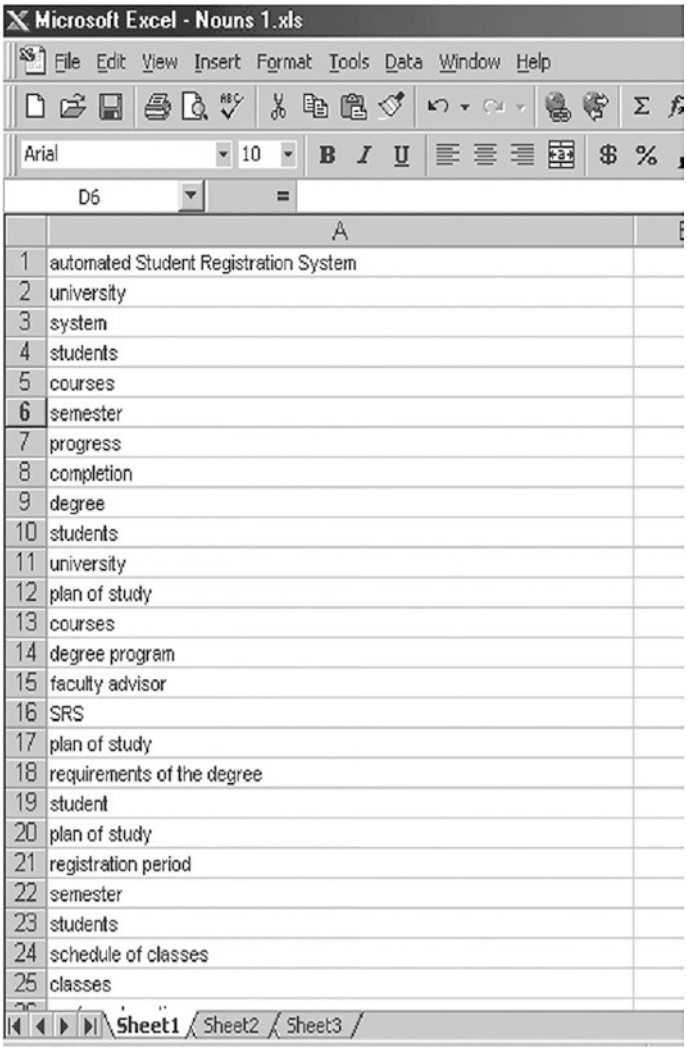


Figure 10-1. *Noun phrases found in the SRS specification*

We're working with a very concise requirements specification (approximately 350 words in length), and yet this process is already proving to be very tedious! It would be impossible to carry out an exhaustive noun phrase analysis for anything but a trivially simple specification. If you're faced with a voluminous requirements specification, start by writing an "executive summary" of no more than a few pages to paraphrase the system's mission, and then use your summary version of the specification as the starting point for your noun survey. Paraphrasing a specification in this fashion provides the added benefit of ensuring that you have read through the system requirements and understand the "big picture." Of course, you'll need to review your summary narrative with your customers/users to ensure that you've accurately captured all key points.

After we've typed all of the nouns/noun phrases into the spreadsheet, we sort the spreadsheet and eliminate duplicates; this includes eliminating plural forms of singular terms (e.g., eliminate "students" in favor of "student"). We want all of our class names to be singular in the final analysis, so if any plural forms remain in the list after eliminating duplicates (e.g., "prerequisites"), we make these singular, as well. In so doing, our SRS list shrinks to 38 items in length, as shown in Figure 10-2.

	A
1	automated Student Registration System
2	class
3	class that he/she was previously waitlisted for
4	completion
5	course
6	courses completed
7	day of the week
8	degree
9	degree program
10	email message
11	end
12	faculty advisor
13	first week of the semester in which the class is being taught
14	first-come, first-served wait list
15	grades received
16	plan of study
17	plan of study requirements
18	preferred section
19	prerequisites
20	professor
21	progress
22	registration period
23	requested course
24	requirements of the degree
25	responsibility
26	room
27	schedule of classes
28	seating capacity
29	section
30	section that he/she was previously waitlisted for
31	semester
32	SRS
33	student
34	system
35	time of day
36	transcript
37	university
38	waitlisted class

Sheet1 Sheet2 Sheet3

Figure 10-2. *Removing duplicates streamlines the noun phrase list*

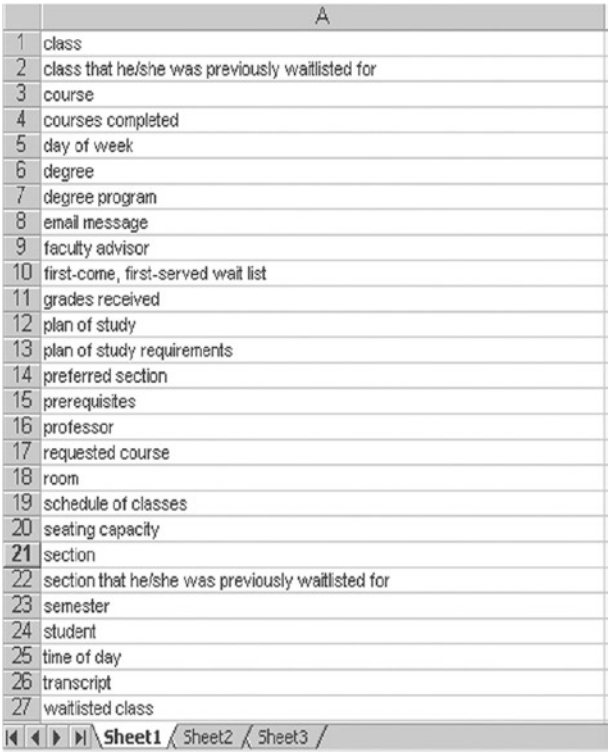
Remember, we're trying to identify both physical and conceptual objects: as stated in Chapter 3, "***something mental or physical toward which thought, feeling, or action is directed***." Let's now make another pass to eliminate the following:

- References to the system itself ("automated Student Registration System," "SRS," "system").
- References to the university. Because we're building the SRS within the context of a single university, the university in some senses "sits outside" and "surrounds" the SRS; we don't need to manipulate information about the university within the SRS, and so we may eliminate the term "university" from our candidate class list.

  Note, however, that if we were building a system that needed to span multiple universities—say, a system that compared graduate programs of study in information technology across the top 100 universities in the country—then we would indeed need to model each university as a separate object, in which case we'd keep "university" on our candidate class list.

- Other miscellaneous terms that don't seem to fit the definition of an object are "completion," "end," "progress," "responsibility," "registration period," and "requirements of the degree." Admittedly, some of these are debatable, particularly the last two; to play it safe, you may wish to create a list of rejected terms to be revisited later on in the modeling life cycle.

The list shrinks to 27 items as a result, as shown in Figure 10-3—it's starting to get manageable now!



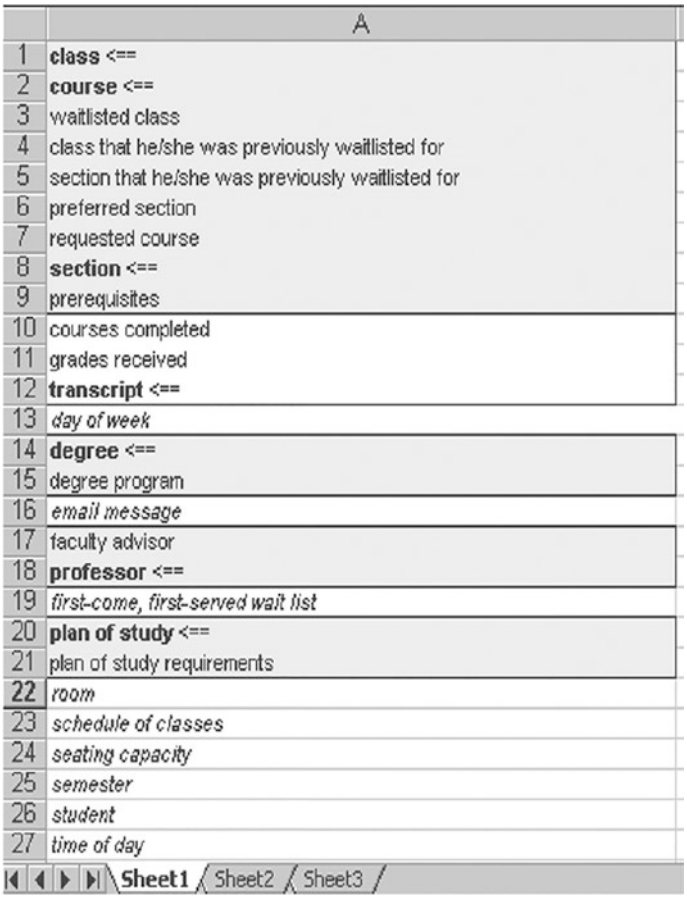
	A
1	class
2	class that he/she was previously waitlisted for
3	course
4	courses completed
5	day of week
6	degree
7	degree program
8	email message
9	faculty advisor
10	first-come, first-served wait list
11	grades received
12	plan of study
13	plan of study requirements
14	preferred section
15	prerequisites
16	professor
17	requested course
18	room
19	schedule of classes
20	seating capacity
21	section
22	section that he/she was previously waitlisted for
23	semester
24	student
25	time of day
26	transcript
27	waitlisted class

Sheet1 Sheet2 Sheet3

Figure 10-3. *Further streamlining the SRS noun phrase list*

The next pass is a bit trickier. We need to group apparent synonyms, to choose the one designation from among each group of synonyms that is best suited to serve as a class name. Having a subject matter expert on your modeling team is important for this step, because determining the subtle shades of meaning of some of these terms so as to group them properly isn't always easy.

We group together terms that seem to be synonyms, as shown in Figure 10-4, **bolding** the term in each synonym group that we're inclined to choose above the rest. *Italicized* words represent those terms for which no synonyms have been identified.



	A
1	**class** <==
2	**course** <==
3	waitlisted class
4	class that he/she was previously waitlisted for
5	section that he/she was previously waitlisted for
6	preferred section
7	requested course
8	**section** <==
9	prerequisites
10	courses completed
11	grades received
12	**transcript** <==
13	*day of week*
14	**degree** <==
15	degree program
16	*email message*
17	faculty advisor
18	**professor** <==
19	*first-come, first-served wait list*
20	**plan of study** <==
21	plan of study requirements
22	*room*
23	*schedule of classes*
24	*seating capacity*
25	*semester*
26	*student*
27	*time of day*

Sheet1 Sheet2 Sheet3

Figure 10-4. *Grouping synonyms*

Let's now review the rationale for our choices.

We choose the shorter form of equivalent expressions whenever possible—"degree" instead of "degree program" and "plan of study" instead of "plan of study requirements"—to make our model more concise.

Although they aren't synonyms as such, the notion of a "transcript" implies a record of "courses completed" and "grades received," so we'll drop the latter two noun phrases for now.

When choosing candidate class names, we should avoid choosing nouns that imply **roles** between objects. As you learned in Chapter 5, a role is something that an object belonging to class A possesses by virtue of its relationship to/association with an object belonging to class B. For example, a professor holds the role of "faculty advisor" when that professor is associated with a student via an *advises* association. Even if a professor

were to lose all of their advisees, thus losing the role of faculty advisor, their would still be a professor by virtue of being employed by the university—it's inherent in the person's nature relative to the SRS.

If a professor were to lose their job with the university, one might argue that they are no longer a professor; but then, this person would have no dealings with the SRS, either, so it's a moot point.

For this reason, we prefer "Professor" to "Faculty Advisor" as a candidate class name, but make a mental note to ourselves that faculty advisor would make a good potential association when we get to considering such things later on.

Regarding the notion of a course, we see that we've collected numerous noun phrases that all refer to a course in one form or another: "class," "course," "preferred section," "requested course," "section," "prerequisite," "waitlisted class," "class that they were previously waitlisted for," "section that they were previously waitlisted for." Within this grouping, several roles are implied:

- "Waitlisted class" in its several different forms implies a role in an association between a `Student` and a `Course`.
- "Prerequisite" implies a role in an association between two `Courses`.
- "Requested course" implies a role in an association between a `Student` and a `Course`.
- "Preferred section" implies a role in an association between a `Student` and a `Course`.

Eliminating all of these role designations, we're left with only three terms: "class," "course," and "section." Before we hastily eliminate all but one of these as synonyms, let's think carefully about what real-world concepts we're trying to represent.

- The notion that we typically associate with the term "course" is that of a semester-long series of lectures, assignments, exams, etc. that all relate to a particular subject area and that are a unit of education toward earning a degree. For example, Beginning Math is a course.
- The terms "class" and "section," on the other hand, generally refer to the offering of a ***particular*** course in a ***given*** semester on a given day

of the week and at a given time of day. For example, the course Math 101 is being offered this coming Spring semester as three classes/sections:

- Section 1, which meets Tuesdays from 4:00 to 6:00 p.m.
- Section 2, which meets Wednesdays from 6:00 to 8:00 p.m.
- Section 3, which meets Thursdays from 3:00 to 5:00 p.m.

There is thus a one-to-many association between `Course` and `Class/Section`. The same course is offered potentially many times in a given semester and over many semesters during the "lifetime" of the course.

Therefore, "course" and "class/section" truly represent different abstractions, and we'll keep ***both*** concepts in our candidate class list. Since "class" and "section" appear to be synonyms, however, we need to choose one term and discard the other. Our initial inclination would be to keep "class" and discard "section," but in order to avoid confusion when referring to a class named `Class` (!), we'll opt for "section" instead.

Refining the Candidate Class List

A list of candidate classes has begun to emerge. Here is our remaining "short list" (please disregard the trailing symbols [*, +] for the moment—I'll explain their significance shortly):

- Course
- Day of week*
- Degree*
- Email message+
- Plan of study
- Professor
- Room*
- Schedule of classes+
- Seating capacity*

- Section
- Semester*
- Student
- Time of day*
- Transcript
- (First-come, first-served) Waitlist

Not all of these will necessarily survive to the final model, however, as we're going to scrutinize each one very closely before deeming it worthy of implementation as a class. One classic test for determining whether or not an item can stand on its own as a class is to ask these questions:

- Can we think of any ***attributes*** for this class?
- Can we think of any ***services*** that would be expected of objects belonging to this class?

One example is the term "room." We could invent a `Room` class as follows:

```
public class Room {
  // Attributes.
  int roomNo;
  String building;
  int seatingCapacity;
  // etc.
}
```

Or we could simply represent a room location as a `String` attribute of the `Section` class:

```
public class Section {
  // Attributes.
  Course offeringOf;
  String semester;
  char dayOfWeek;  // 'M', 'T', 'W', 'R', 'F'
  String timeOfDay;
  String classroomLocation; // building name and room name:  e.g.,
                            // "Innovation Hall Room 333"
```

```
  // etc.
}
```

Which approach to representing a room is preferred? It all depends on whether or not a room needs to be a focal point of our application. If the SRS were meant to also serve as a Classroom Scheduling System, then we might indeed wish to instantiate `Room` objects so as to be able to ask them to perform such services as printing out their weekly usage schedules or telling us their seating capacities. However, since these services weren't mentioned as requirements in the SRS specification, we'll opt for eliminating `Room` as a candidate class and instead making a room designation a simple `String` attribute of the `Section` class. We reserve the right, however, to change our minds about this later on; it's not unusual for some items to "flip-flop" over the life cycle of a modeling exercise between being classes on their own vs. being represented as simple attributes of other classes.

Following a similar train of thought for all of the items marked with an asterisk (*) in the preceding candidate class list, we'll opt to treat them all as attributes rather than making them classes of their own:

- "Day of week" will be incorporated as either a `String` or `char` attribute of the `Section` class.
- "Degree" will be incorporated as a `String` attribute of the `Student` class.
- "Seating capacity" will be incorporated as an `int` attribute of the `Section` class.
- "Semester" will be incorporated as a `String` attribute of the `Section` class.
- "Time of day" will be incorporated as a `String` attribute of the `Section` class.

When we're first modeling an application, we want to focus exclusively on functional requirements to the exclusion of technical requirements, as defined in Chapter 9; this means that we need to avoid getting into the technical details of how the system is going to function behind the scenes. Ideally, we want to focus solely on what are known as **domain classes**—that is, abstractions that an end user will recognize and that represent

"real-world" entities—and to avoid introducing any extra classes that are used solely as behind-the-scenes "scaffolding" to hold the application together, known alternatively as **implementation classes** or **solution space classes**. Examples of the latter would be the creation of a collection object to organize and maintain references to all of the `Professor` objects in the system or the use of a dictionary to provide a way to quickly find a particular `Student` object based on the associated student ID number. We'll talk more about solution space objects in Part 3 of the book; for the time being, the items flagged with a plus sign (+) in the candidate class list earlier—"email message," "schedule of classes"—seem arguably more like implementation classes than domain classes:

- An email message is typically a transient piece of data, not unlike a pop-up message that appears on the screen while using an application. An email message gets sent out of the SRS, and after it's read by the recipient, we have no control over whether the email is retained or deleted. It's unlikely that the SRS is going to archive copies of all email messages that have been sent—there certainly was no requirement to do so—so we won't worry about modeling them as objects at this stage in our analysis.
- Email messages will resurface in Chapter 11, when we talk about the behaviors of the SRS application, because sending an email message is definitely an important behavior; but emails don't constitute an important structural piece of the application, so we don't want to introduce a class for them at this stage in the modeling process. When we actually get to programming the system, we might indeed create an `EmailMessage` class in Java, but it needn't be modeled as a domain class. (If, on the other hand, we were modeling an email messaging system in anticipation of building one, then `EmailMessage` could indeed be a key domain class in our model.)
- We could go either way with the schedule of classes—include it as a candidate class, or drop it from our list. The schedule of classes, as a single object, may not be something that the user will manipulate directly, but there will be some notion behind the scenes of a schedule of classes collection controlling which `Section` objects should be presented to the user as a GUI pick list when they register in a given semester. We'll omit `ScheduleOfClasses` from our candidate class list for now, but we can certainly revisit our decision as the model evolves.

Determining whether or not a candidate class constitutes a domain class instead of an implementation class is admittedly a gray area, and either of the preceding candidate class "rejects" could be successfully argued into or out of the list of core domain classes for the SRS. In fact, this entire exercise of identifying classes hopefully illustrates a concept that was first introduced in Chapter 2; because of its importance, it is repeated again in the following sidebar.

THE CHALLENGES OF OBJECT MODELING

Developing an appropriate model for a software system is perhaps the most difficult aspect of software engineering, because

There are an unlimited number of possibilities. Abstraction is to a certain extent in the eye of the beholder: several different observers working independently are almost guaranteed to arrive at different models. Whose is the best? Passionate arguments have ensued!

To further complicate matters, ***there is virtually never only one "best" or "correct" model***, only "better" or "worse" models (including incorrect models) relative to the problem to be solved. The same situation can be modeled in a variety of equally valid ways.

Finally, there is no "acid test" to determine if a model has ***adequately captured all of a user's requirements.***

As we continue along with our SRS modeling exercise, and particularly as we move from modeling to implementation in Part 3 of the book, we'll have many opportunities to rethink the decisions that we've made here. The key point to remember is that the model isn't "cast in stone" until we actually begin programming, and even then, if we've used objects wisely, the model can be fairly painlessly modified to handle most new requirements. Think of a model as being formed out of modeling clay: we'll continue to reshape it in Agile fashion over the course of the analysis and design phases of our project until we're satisfied with the result.

Meanwhile, back to the task of coming up with a list of candidate classes for the SRS. The terms that have survived our latest round of scrutiny are as follows:

- `Course`
- `PlanOfStudy`
- `Professor`

- `Section`
- `Student`
- `Transcript`
- `WaitList`

Let's examine `WaitList` one last time. There is indeed a requirement for the SRS to maintain a student's position on a first-come, first-served waitlist. But it turns out that this requirement can actually be handled through a combination of an association between the `Student` and `Section` classes, plus something known as an **association class**, which you'll learn about later in this chapter. This would not be immediately obvious to a beginning modeler, and so we'd fully expect that the `WaitList` class might make the final cut as a suggested SRS class. But we're going to assume that we have an experienced object modeler on the team, who convinces us to eliminate the class; we'll see that this is a suitable move when we complete the SRS class diagram at the end of the chapter.

So we'll settle on the following list of classes, based on our noun phrase analysis of the SRS specification:

- `Course`
- `PlanOfStudy`
- `Professor`
- `Section`
- `Student`
- `Transcript`

Revisiting the Use Cases

One more thing that we need to do before we deem our class list good to go is to revisit our use cases—in particular, the actors—to see if any of ***these*** ought to be added as classes. You may recall that we identified seven potential actors for the SRS in Chapter 9:

- Student
- Faculty
- Department Chair

- Registrar
- Billing System
- Admissions System
- Classroom Scheduling System

Do any of these deserve to be modeled as classes in the SRS? Here's how to make that determination: if any user associated with any actor type `A` is going to need to manipulate (access or modify) information concerning an actor type `B` when `A` is logged on to the SRS, then `B` needs to be included as a class in our model. This is best illustrated with a few examples:

- When a student logs on to the SRS, might they need to manipulate information about faculty? Yes, when a student selects an advisor, for example, they might need to view information about a variety of faculty members in order to choose an appropriate advisor. So ***the Faculty actor role must be represented as a class in the SRS***; indeed, we have already designated a `Professor` class, so we're covered there. But student users are not concerned with department chairs per se.
- Following the same logic, ***we'd need to represent the Student actor role as a class*** because when professors log on to the SRS, they will be manipulating `Student` objects when printing out a course roster or assigning grades to students, for example. Since `Student` already appears in our candidate class list, we're covered there, as well.
- When ***any*** of the actors—Faculty, Students, the Registrar, the Billing System, the Admissions System, or the Classroom Scheduling System—access the SRS, will there be a need for any of them to manipulate information about the registrar? No, at least not according to the SRS requirements that we've seen so far. Therefore, ***we needn't model the Registrar actor role as a class.***
- ***The same holds true for the Billing, Admissions, and Classroom Scheduling Systems***: they require "behind-the-scenes" access to information managed by the SRS, but nobody logging on to the SRS expects to be able to manipulate any of these three systems directly, so ***they needn't be represented by domain classes in the SRS***.

> Again, when we get to implementing the SRS in code, we may indeed find it appropriate to create "solution space" Java classes to represent interfaces to these other automated systems; but such classes don't belong in a ***domain*** model of the SRS.

Therefore, our proposed candidate class list remains unchanged after revisiting all actor roles:

- `Course`
- `PlanOfStudy`
- `Professor`
- `Section`
- `Student`
- `Transcript`

Is this a "perfect" list? No, there is no such thing. In fact, before all is said and done, the list may—and in fact probably will—evolve in the following ways:

- We may add classes later on: terms we eliminated from the specification or terms that don't even appear in the specification, but which we'll unearth through continued investigation.
- We may see an opportunity to generalize—that is, we may see enough commonality between two or more classes' respective attributes, methods, or relationships with other classes to warrant the creation of a common base class.
- In addition, as mentioned earlier, we may rethink our decisions regarding representing some concepts as simple attributes (semester, room, etc.) instead of as full-blown classes and vice versa.

The development of a candidate class list is, as illustrated in this chapter thus far, fraught with uncertainty. For this reason, it's important to have someone experienced with object modeling available to your team when embarking on your first object modeling effort. Most experienced modelers don't use the rote method of noun phrase analysis to derive a candidate class list; such folks can pretty much review a specification and directly pick out significant classes, in the same way that a professional jeweler can

easily choose a genuine diamond from among a pile of fake gemstones. Nevertheless, what does "significant" really mean? That's where the "fuzziness" comes in. It's impossible to define precisely what makes one concept significant and another less so. I've tried to illustrate some rules of thumb by working through the SRS example, but you ultimately need a qualified mentor to guide you until you develop—and trust—your own intuitive sense for such things.

The bottom line, however, is that even expert modelers can't really confirm the appropriateness of a given candidate class until they see its proposed use in the full context of a class diagram that also reflects associations, attributes, and methods, which we'll explore later in this chapter as well as in Chapter 11.

Producing a Data Dictionary

Early on in our analysis efforts, it's important that we clarify and begin to document our use of terminology. A **data dictionary** is ideal for this purpose. For each candidate class, the data dictionary should include a simple definition of what this item means in the context of the model/system as a whole; include an example if it helps illustrate the definition.

The following sidebar shows our complete SRS data dictionary so far.

THE SRS DATA DICTIONARY, TAKE 1: CLASS DEFINITIONS

- ***Course:*** A semester-long series of lectures, assignments, exams, etc. that all relate to a particular subject area and that are typically associated with a particular number of credit hours, a unit of study toward a degree. For example, Beginning Objects is a required **course** for the Master of Science degree in information systems technology.
- ***PlanOfStudy:*** A list of the **courses** that a student intends to take to fulfill the **course** requirements for a particular degree.
- ***Professor:*** A member of the faculty who teaches **sections** or advises **students**.
- ***Section:*** The offering of a particular **course** during a particular semester on a particular day of the week and at a particular time of day (e.g., **course** Beginning Objects as taught in the Spring 2025 semester on Mondays from 1:00 to 3:00 p.m.).

- ***Student:*** A person who is currently enrolled at the university and who is eligible to register for one or more **sections**.
- ***Transcript:*** A record of all of the **courses** taken to date by a particular **student** at this university, including which semester each **course** was taken in, the grade received, and the credits granted for the **course**, as well as a reflection of an overall total number of credits earned and the **student's** grade point average (GPA).

Note that it's permissible, and in fact encouraged, for the definition of one term to include one or more of the other terms; when we do so, we highlight the latter in **bold text**.

The data dictionary joins the set of other SRS narrative documents as a subsequent source of information about the model. As our model evolves, we'll expand the dictionary to include definitions of attributes, associations, and methods.

It's a good idea to also include the dictionary definition of a class as a header comment in the Java code representing that class. Make sure to keep this inline documentation in sync with the external dictionary definition, however.

Determining Associations Between Classes

Once we've settled on an initial candidate class list, the next step is to determine how these classes are interrelated. To do this, we go back to our narrative documentation set (which has grown to consist of the SRS requirements specification, use cases, and data dictionary) and study ***verb*** phrases this time. Our goal in looking at verb phrases is to choose those that suggest structural relationships, as we've defined in Chapter 5—associations, aggregations, and inheritance—but to eliminate or ignore those that represent (transient) actions or behaviors. (We'll focus on behaviors, but from the standpoint of use cases, in Chapter 11.)

For example, the specification states that a student "chooses a faculty advisor." This is indeed an action, but the result of this action is a lasting structural relationship between a professor and a student, which can be modeled via the association "a `Professor` *advises* a `Student`."

As a student's advisor, a professor also meets with the student, answers the student's questions, recommends courses for the student to take, approves the student's plan of study, etc.—these are behaviors on the part of a professor acting in the role of an advisor, but don't directly result in any new relationships being formed between objects.

Let's try the verb phrase analysis approach on the requirements specification. All relevant verb phrases are highlighted in the sidebar that follows (note that I omitted such obviously irrelevant verb phrases as "We've been asked to develop an automated SRS… ").

HIGHLIGHTING VERB PHRASES IN THE SRS SPECIFICATION

We have been asked to develop an automated Student Registration System (SRS) for the university. This system will **enable students to register** online **for courses** each semester, as well as **track their progress toward completion of their degree**.

When a student first **enrolls at the university**, they use the SRS to **set forth a plan of study** as to which **courses they plan on taking** to **satisfy a particular degree program** and **choose a faculty advisor**. The SRS will **verify whether or not the proposed plan of study satisfies the requirements of the degree that the student is seeking**.

Once a **plan of study has been established**, then, during the registration period preceding each semester, a student is able to **view the schedule of classes** online and **choose whichever classes they wish to attend, indicating the preferred section** (day of the week and time of day) if the **class is offered by more than one professor**. The SRS will **verify whether or not the student has satisfied the necessary prerequisites** for each requested course by **referring to the student's online transcript** of courses completed and grades received (the **student may review their transcript** online at any time).

Assuming that (a) the **prerequisites for the requested course(s) are satisfied**, (b) the **course(s) meet(s) one of the student's plan of study requirements**, and (c) **there is room available** in each of the class(es), the **student is enrolled in the class(es)**.

If (a) and (b) are satisfied, but (c) is not, the **student is placed on a first-come, first-served waitlist**. If a **class/section that they were previously waitlisted for becomes available** (either because some other **student has dropped the class** or because the **seating capacity for the class has been increased**), the **student is automatically enrolled in the waitlisted**

class, and an **email message** to that effect **is sent** to the student. It is the student's responsibility to **drop the class** if it is no longer desired; otherwise, **they will be billed for the course**.

Students may drop a class up to the end of the first week of the semester in which the **class is being taught**.

Let's scrutinize a few of these:

- ***"Students [...] register [...] for courses"***: Although the act of registering is a behavior, the end result is that a static relationship is created between a `Student` and a `Section`, as represented by the association "a `Student` *registers* for a `Section`." Note that the specification mentions registering for "courses," not "sections," but as we stated in our data dictionary, a `Student` registers for concrete `Sections` as embodiments of `Courses`. Keep in mind when reviewing a specification that natural language is often imprecise and that as a result we have to read between the lines as to what the author really meant in every case. (If we're going to be the ones to write the specification, here is an incentive to keep the language as clear and concise as possible.)
- ***"[Students track] their progress toward completion of their degree"***: Again, this is a behavior, but it nonetheless implies a structural relationship between a `Student` and a `Degree`. However, recall that we didn't elect to represent `Degree` as a class—we opted to reflect it as a simple `String` attribute of the `Student` class—and so this suggested relationship is immaterial with respect to the candidate class list that we've developed.
- ***"Student first enrolls at the university"***: This is a behavior that results in a static relationship between a `Student` and the `University`; but we deemed the notion of "university" to be external to the system and so chose not to create a `University` class in our model. So we disregard this verb phrase, as well.

- ***"[Student] sets forth a plan of study"***: This is a behavior that results in the static relationship "a `Student` *pursues/observes* a `Plan of Study`."
- ***"Students are able to view the schedule of classes online"***: This is strictly a transient behavior of the SRS; no lasting relationship results from this action, so we disregard this verb phrase.
- And so on.

Association Matrices

Another complementary technique for both determining and recording what the relationships between classes should be is to create an ***n*** × ***n*** **association matrix,** where ***n*** represents the number of candidate classes that we've identified. Label the rows and the columns with the names of the classes, as shown for the empty matrix represented by Table 10-1.

Table 10-1. *An "Empty" Association Matrix for the SRS*

	`Section`	`Course`	`PlanOfStudy`	`Professor`	`Student`	`Transcript`
`Section`						
`Course`						
`PlanOfStudy`						
`Professor`						
`Student`						
`Transcript`						

Then, complete the matrix as follows.

In each cell of the matrix, list all of the associations that you can identify between the class named at the head of the row and the class named at the head of the column. For example, in the cell highlighted in Table 10-2 at the intersection of the `Student` "row" and the `Section` "column," we have listed three potential associations:

- A `Student` *is waitlisted for* a `Section`.

- A `Student` *is registered for* a `Section`. (This could be alternatively phrased as "a `Student` *is currently attending* a `Section`.")
- A `Student` *has previously taken* a `Section`. This third association is important if we plan on maintaining a history of all of the classes that a student has ever taken in their career as a student, which we must do if we are to prepare a student's transcript online. (As it turns out, we'll be able to get by with a single association that does "double duty" for the latter two of these, as you'll see later on in this chapter.)

Mark a cell with an × if there are no known relationships between the classes in question or if the potential relationships between the classes are irrelevant. For example, in Table 10-2 the cells representing the intersection between `Professor` and `Course` are marked with an ×, even though there is an association possible—"a `Professor` *is qualified to teach* a `Course`"—because it isn't relevant to the mission of the SRS.

As mentioned in Chapter 4, all associations are inherently bidirectional. This implies that if a cell in row *j*, column *k* indicates one or more associations, then the cell in row *k*, column *j* should reflect the reciprocal of these relationships. For example, since the intersection of the `PlanOfStudy` "row" and the `Course` "column" indicates that "a `PlanOfStudy` *calls for* a `Course`," then the intersection of the `Course` "row" and the `PlanOfStudy` "column" must indicate that "a `Course` *is called for by* a `PlanOfStudy`."

It's not always practical to state the reciprocal of an association; for example, our association matrix shows that "a `Student` *plans to take* a `Course`," but trying to state its reciprocal—"a `Course` *is planned to be taken by* a `Student`"—is quite awkward. In such cases where a reciprocal association would be awkward to phrase, simply indicate its presence with the symbol ✓.

Table 10-2. *Our Completed Association Matrix*

	Section	Course	PlanOfStudy	Professor	Student	Transcript
Section	×	*instance of*	×	*is taught by*	✓	*included in*
Course	✓	*prerequisite for*	*is called for by*	×	✓	×
PlanOf Study	×	*calls for*	×	×	*observed by*	×
Professor	*teaches*	×	×	×	*advises, teaches*	×
Student	*registered for, waitlisted for, has previously taken*	*plans to take*	*observes*	*is advised by, studies under*	×	*owns*
Trans cript	*includes*	×	×	×	*belongs to*	×

We'll be portraying these associations in graphical form shortly. For now, let's go back and extend our data dictionary to explain what each of these associations means. The following sidebar shows one such example.

ADDITIONS TO THE SRS DATA DICTIONARY

Calls for (a Plan of Study calls for a Course): In order to demonstrate that a **student** will satisfy the requirements for their chosen degree program, the **student** must formulate a **plan of study**. This **plan of study** lays out all of the **courses** that a **student** intends to take and possibly specifies in which semester the **student** hopes to complete each **course**.

Identifying Attributes

To determine what the attributes for each of our domain classes should be, we make yet another pass through the requirements specification looking for clues. We already stumbled upon a few attributes earlier, when we weeded out some nouns/noun phrases from our candidate class list:

- For the `Section` class, we identified "day of week," "room," "seating capacity," "semester," and "time of day" as attributes.
- For the `Student` class, we identified "degree" as an attribute.

We can also bring any prior knowledge that we have about the domain into play when assigning attributes to classes. Our knowledge of the way that universities operate, for example, suggests that all students will need some sort of student ID number as an attribute, even though this isn't mentioned anywhere in the SRS specification. This is a detail that we'd have to go back to our end users for clarification on.

Finally, we can also look at how similar information has been represented in existing legacy systems for clues as to what a class's attributes should be. For example, if a Student Billing System already exists at the university based on a relational database design, we might wish to study the structure of the relational database table housing student information. The columns that have been provided in that table—name, address, birthDate, etc.—are potential attribute choices.

UML Notation: Modeling the Static Aspects of an Abstraction

Now that we have a much better understanding about the static aspects of our model, we're ready to portray these in graphical fashion to complement the narrative documentation that we've developed for the SRS. We'll be using the UML to produce a **class diagram.** Here are the rules for how various aspects of the model are to be portrayed.

Classes, Attributes, and Operations

We represent classes as rectangles. When we first conceive of a class—before we know what any of its attributes or methods are going to be—we simply place the class name in the rectangle, as illustrated in Figure 10-5.

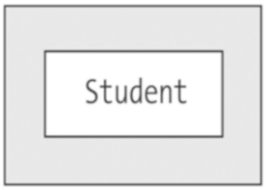


Figure 10-5. *UML depiction of the* `Student` *class*

An ***abstract*** class is denoted by presenting the class name in italics, as shown in Figure 10-6.

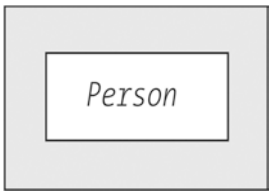


Figure 10-6. *UML depiction of an abstract class*

When we're ready to reflect the attributes and behaviors of a class, we divide the class rectangle into three **compartments**—the class name compartment, the attributes compartment, and the operations compartment—as shown in Figure 10-7. Note that UML favors the nomenclature of "operations" vs. "methods" to reinforce the notion that the diagram is intended to be programming language independent.

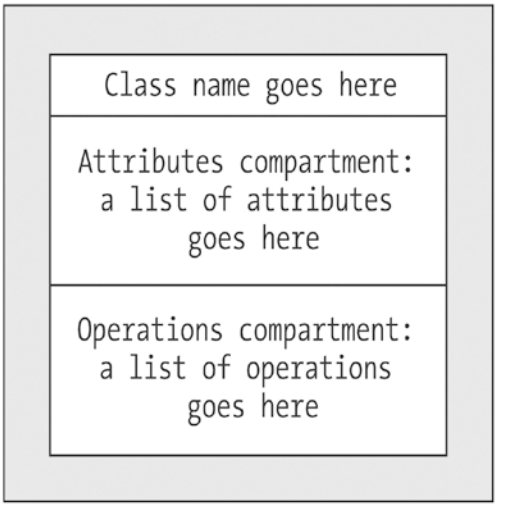


Figure 10-7. *Class rectangles are divided into three compartments*

Some CASE tools automatically portray all three (empty) compartments when a class is first created, even if we haven't specified any attributes or operations yet, as shown in Figure 10-8.

Figure 10-8. *Alternative UML class depiction as rendered by some CASE tools*

As we begin to identify what the attributes and/or operations need to be for a particular class, we can add these to the diagram in as much or as little detail as we care to.

We may choose simply to list attribute names (see Figure 10-9), or we may specify their names along with their types (see Figure 10-10).

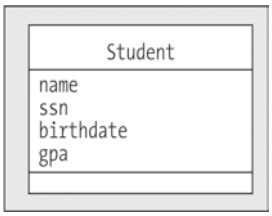


Figure 10-9. *Sometimes just attribute names are presented*

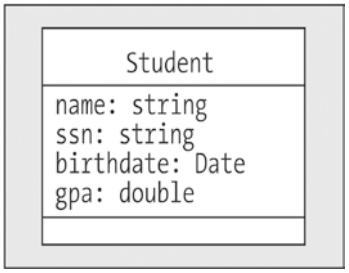


Figure 10-10. *Sometimes both attribute names and types are shown*

We may even wish to specify an initial starting value for an attribute, as in `gpa : double = 0.0`, although this is less common.

Static attributes are identified as such by underlining their names (see Figure 10-11).

Figure 10-11. *Identifying static attributes by underlining*

We may choose simply to list operation names in the operations compartment of a class rectangle, as shown in Figure 10-12, or we may optionally choose to use an expanded form of operation definition, as we have for the `registerForCourse` operation in Figure 10-13.

Figure 10-12. *Sometimes just method names are presented*

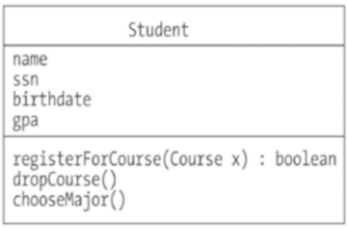


Figure 10-13. *Sometimes argument signatures and return types are also reflected*

Note that the formal syntax for operation specifications in a UML class diagram

`[visibility] name [(parameter list)] [: return type]`

for example

`registerForCourse(Course x) : boolean`

differs from the syntax that we're used to seeing for Java method headers:

`returnType methodName(parameter list)`

For example:

`boolean registerForCourse(Course x)`

Note in particular that the UML refers to the combination of operation name, parameters, and return type as the **operation signature**, but that in Java the return type is part of the method header but ***not*** part of the ***method signature***.

The rationale for making these operation signatures generic vs. language specific is so that the same model may be rendered in any of a variety of target programming languages. It can be argued, however, that there is nothing inherently better or clearer about the first form vs. the second. Therefore, if you know that you're going to be programming in Java, it might make sense to reflect standard Java method headers in your class diagram, if your object modeling tool will accommodate this.

It's often impractical to show all of the attributes and operations of every class in a class diagram, because the diagram will get so cluttered that it will lose its power as a communications tool. Consider the data dictionary to be the official, complete source of information concerning the model, and reflect in the diagram only those attributes and operations that are particularly important in describing the mission of each class. In particular, "get" and "set" operations are implied for all attributes and shouldn't be explicitly shown.

Also, just because the attribute or operation compartment of a class is empty, don't assume that there are no features of that type associated with a class; it may simply mean that the model is still evolving.

Relationships Between Classes

Chapter 4 defined several different types of structural relationship that may exist between classes: associations, aggregations (a specific type of association), and inheritance. Let's explore how each of these relationship types is represented graphically.

Associations

Binary associations—in other words, relationships between two different classes—are indicated by drawing a line between the rectangles representing the participating classes and labeling the line with the name of the association. Role names can be reflected at either end of the association line if they add value to the model, but should otherwise be omitted.

We also mark each end of the line with the appropriate **multiplicity designator**, to reflect whether the relationship is one-to-one, one-to-many, or many-to-many (see Figure 10-14); we'll look at how to do this a bit later in the chapter.

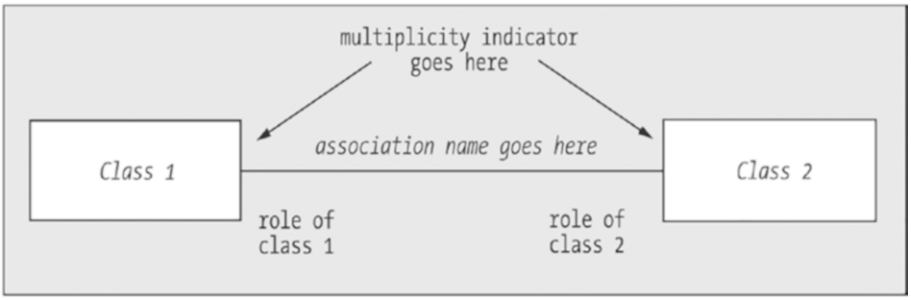


Figure 10-14. *Representing associations between classes*

All associations are assumed to be bidirectional at this stage in the modeling effort, and it doesn't matter in which order the participating classes are arranged in a class diagram. So, to depict the association "a `Professor` *advises* a `Student`," the graphical notations in Figure 10-15 are all considered equivalent.

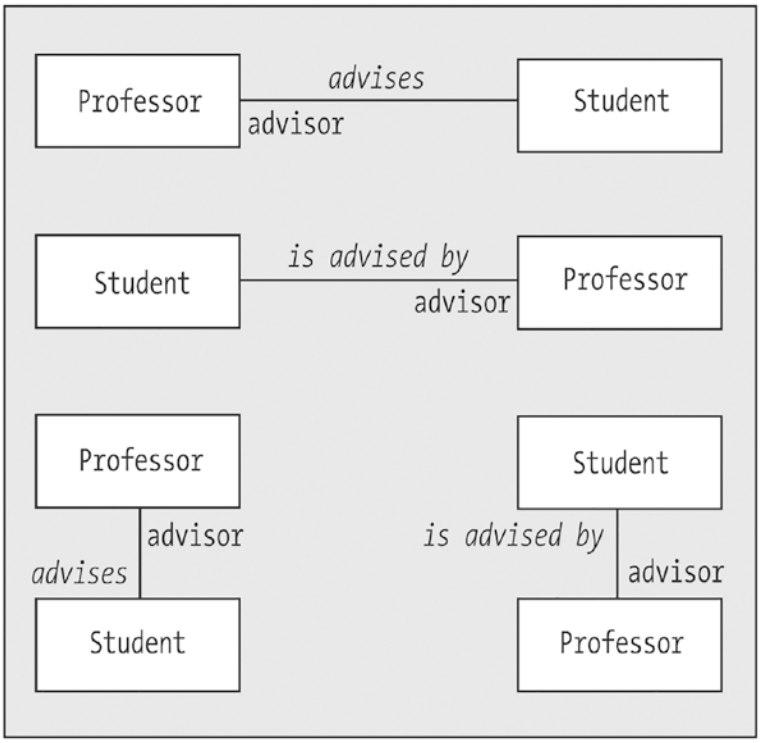


Figure 10-15. *Equivalent depictions of the advises association between the* `Professor` *and* `Student` *classes*

Achieving an optimal placement of classes for purposes of simplifying all of the association names in a diagram is often not possible in an elaborate diagram. Therefore, UML has introduced the simple convention of using a small arrowhead (◄) to reflect the direction in which the association name is to be interpreted, giving us a lot more freedom in how we place our class rectangles in a diagram, as shown in Figure 10-16.

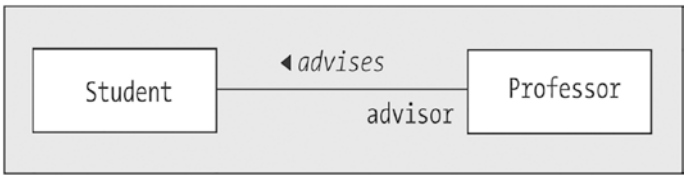


Figure 10-16. *Using an arrowhead to indicate the direction of an association label*

With UML, no matter how the preceding two rectangles are situated, we can still always label the association "*advises*."

> It's easy to get caught up in the trap of trying to make diagrams "perfect" in terms of how classes are positioned, to minimize crossed lines, etc. Try to resist the urge to do so early on, because the diagram will inevitably get changed many times before the modeling effort is finished.

Unary (reflexive) associations—that is, relationships between two different objects belonging to the same class—are drawn with an association line that loops back to the same class rectangle from which it originates. For example, to depict the association "a `Course` *is a prerequisite for* a (different) `Course`," we'd use the notation shown in Figure 10-17.

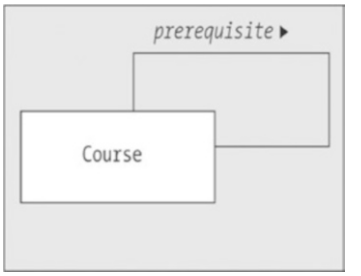


Figure 10-17. *A reflexive association involving the* `Course` *class*

Aggregation

Aggregation, which as you learned in Chapter 5 is a specialized form of association that happens to imply containment, is differentiated from a "normal" association by placing a diamond at the end of the association line that touches the "containing" class. For example, to portray the fact that a university is comprised of schools—School of Engineering, School of Law, School of Medicine, etc.—we'd use the notation shown in Figure 10-18.

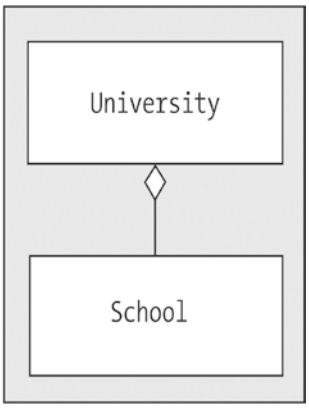


Figure 10-18. *Indicating aggregation with a diamond*

An aggregation relationship can actually be oriented in any direction, as long as the diamond is properly anchored on the "containing" class as shown in Figure 10-19.

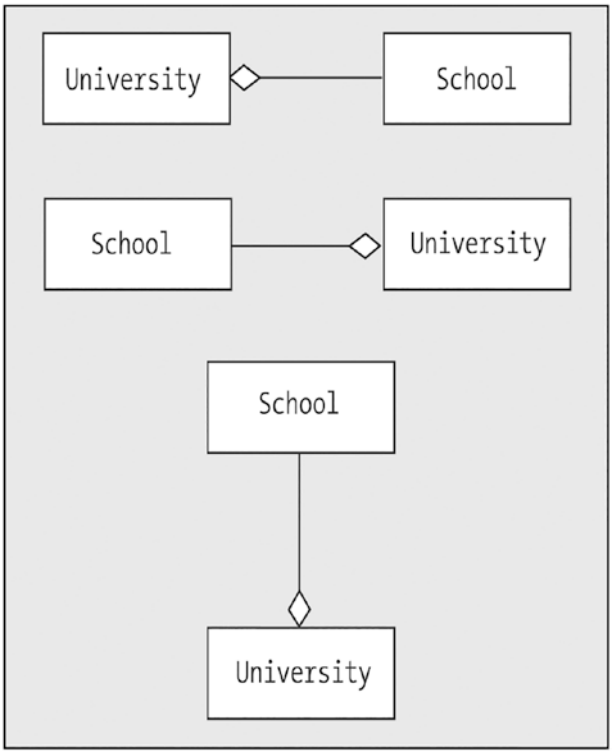


Figure 10-19. *Aggregations can be oriented in any direction*

An aggregation line needn't be labeled, as it is understood that an aggregation implies the "part"-"whole" relationship.

As mentioned in Chapter 5, however, we can get by without ever using aggregation! To represent the preceding concept, we could have just created a simple association between the `University` and `School` classes and labeled it "*is composed of*" as shown in Figure 10-20.

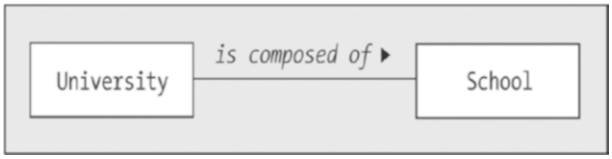


Figure 10-20. *A simple association as an alternative to an aggregation*

The decision of whether to use aggregation vs. plain association is subtle, because it turns out that both can be rendered in code in essentially the same way, as you'll see in Part 3 of the book.

Unlike association lines, which should always be labeled with the name of the association that they represent, aggregation lines are typically not labeled, since an aggregation by definition implies containment.

When two or more different classes represent "parts" of some other "whole," each "part" is involved in a separate aggregation with the "whole," as shown in Figure 10-21.

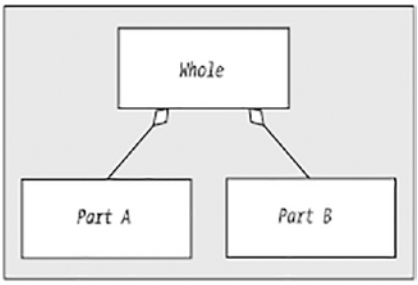


Figure 10-21. *Two aggregations, drawn using two diamonds*

However, we often join such aggregation lines into a single structure that looks something like an organization chart, as shown in Figure 10-22.

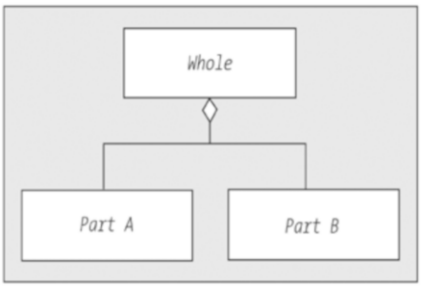


Figure 10-22. *Two aggregations involving the same "whole" class, drawn using a single diamond*

Doing so is not meant to imply anything about the relationship of Part A to Part B; it's simply a way to clean up the diagram.

Composition, which as you learned in Chapter 5 is a strong form of aggregation in which the "parts" cannot exist without the "whole," uses a "filled-in"/"black" diamond rather than an "open"/"white" diamond, as illustrated in Figure 10-23.

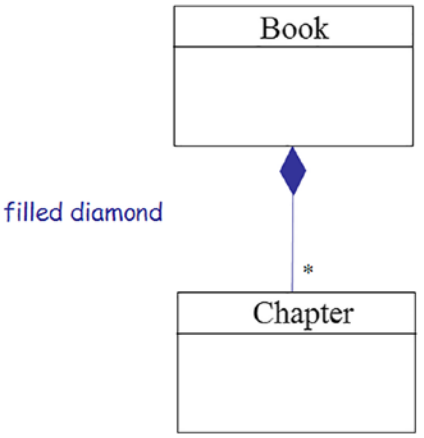


Figure 10-23. *A filled diamond signals composition, a strong form of aggregation*

Inheritance

Inheritance (generalization/specialization) is illustrated by connecting a derived class to its base class with a line and then marking the line with a triangle that touches the base class (see Figure 10-24).

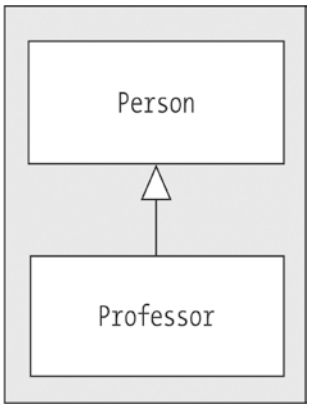


Figure 10-24. *Inheritance is designated with a triangle*

As with aggregation, the classes involved in an inheritance relationship can be portrayed in any orientation, as long as the triangle points to/touches the base class.

Inheritance lines should ***not*** be labeled, as they unambiguously represent the "is a" relationship.

As with aggregation, when two or more different classes represent derived classes of the same parent class, each derived class is involved in a separate inheritance relationship with the parent, as shown in Figure 10-25, but we often join the inheritance lines into a single structure, as illustrated in Figure 10-26.

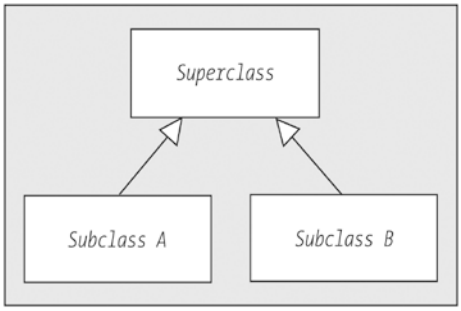


Figure 10-25. *Depicting two derived classes with two different triangles*

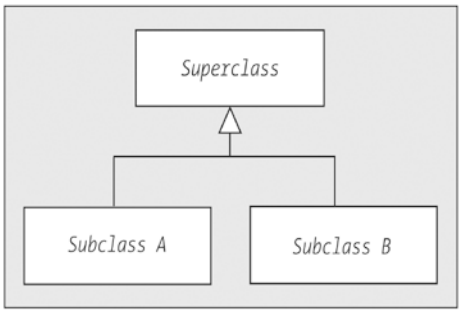


Figure 10-26. *Depicting two derived classes with a single triangle*

Doing so isn't meant to imply anything different about the relationship of derived class A to derived class B as compared with the previous depiction—these classes are considered to be sibling classes with a common parent class in both cases. It's simply a way to clean up the diagram.

Reflecting Multiplicity

You learned in Chapter 5 that for a given association type X between classes A and B, the term "multiplicity" refers to the number of instances of objects of type A that must/may be associated with a given instance of type B and vice versa. When preparing a class diagram, we mark each end of an association line to indicate what its multiplicity should be from the perspective of an object belonging to the class at the other end of the line. In other words

- We mark the number of instances of B that can relate to a single instance of A at B's end of the line.
- We mark the number of instances of A that can relate to a single instance of B at A's end of the line.

This is depicted in Figure 10-27.

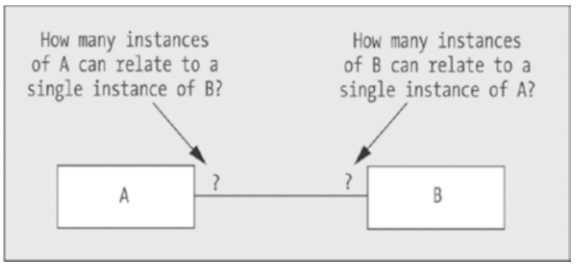


Figure 10-27. *Indicating multiplicity between classes*

By way of review, given a single object belonging to class A, there are four different scenarios for how object(s) of type B may be related to it:

- The A-type object may be related to ***exactly one*** instance of a B-type object, as in the situation "a Student (A) *has* a Transcript (B)." Here, the existence of an instance of B for every instance of A is ***mandatory***.
- The A-type object may be related to ***at most one*** instance of a B-type object, as in the situation "a Professor (A) *chairs* a Department (B)." Here, the existence of an instance of B for every instance of A is ***optional***.
- The A-type object may be related to ***one or more*** instances of a B-type object, as in the situation "a Department (A) *employs* ***many*** Professors (B)." Here, the existence of at least one instance of B for every instance of A is ***mandatory***.
- The A-type object may be related to ***zero or more*** instances of a "B" type object, as in the situation "a Student (A) *is attending* ***many*** Sections (B)." (At our hypothetical university, a student is permitted to take a semester off.) Here, the existence of at least one instance of B for every instance of A is ***optional***.

In UML notation, multiplicity symbols are as follows:

- "Exactly one" is represented by the notation "1".
- "At most one" is represented by the notation "0..1", which is alternatively read as "zero or one."
- "One or more" is represented by the notation "1..*".

- "Zero or more" is represented by the notation "0..*".
- We use the notation "*" when we know that the multiplicity should be "many" but we aren't certain (or we don't care to specify) whether it should be "zero or more" or "one or more."
- It's even possible to represent an arbitrary range of explicit numerical values *x*..*y*, such as using "3..7" to indicate, for example, that "a `Department` employs no fewer than three, and no more than seven, `Professors`."

Here are some UML examples:

"A `Student` *has* exactly one `Transcript`, and a `Transcript` *belongs to* exactly one `Student`." (See Figure 10-28.)

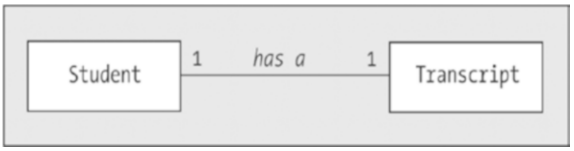


Figure 10-28. *An example of mandatory one-to-one multiplicity*

"A `Professor` *works for* exactly one `Department`, but a `Department` *has* many (one or more) `Professors` as employees." (See Figure 10-29.)

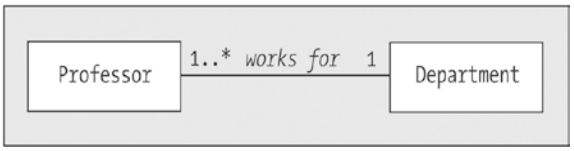


Figure 10-29. *An example of mandatory one-to-many multiplicity*

"A `Professor` optionally *chairs* at most one `Department`, while a `Department` *has* exactly one `Professor` in the role of chairman." (See Figure 10-30.)

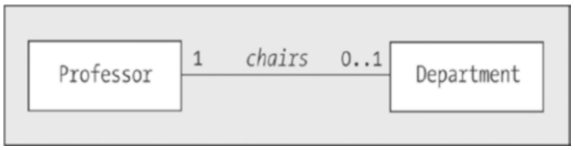


Figure 10-30. *An example of optional one-to-many multiplicity*

"A Student *attends* many (zero or more) Sections, and a Section *is attended by* many (zero or more) Students." (See Figure 10-31.)

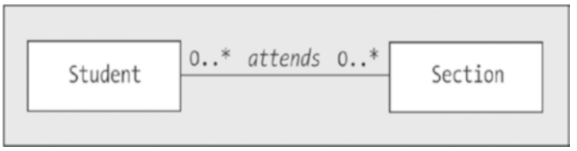


Figure 10-31. *An example of optional many-to-many multiplicity*

A Section that continues to have zero Students signed up to attend will most likely be canceled; nonetheless, there is a period of time after a Section is first made available for enrollment via the SRS that it will have zero Students enrolled.

"A Course *is a prerequisite for* many (zero or more) Courses, and a Course *can have* many (zero or more) *prerequisite* Courses." (See Figure 10-32.)

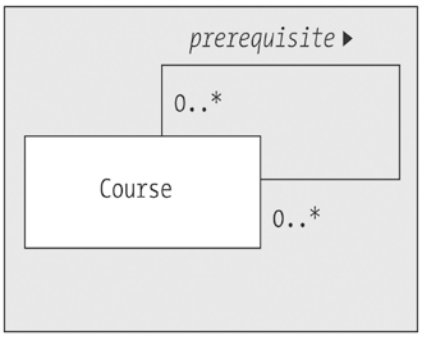


Figure 10-32. *An example of optional many-to-many multiplicity on a reflexive association*

We reflect multiplicity on aggregations as well as on simple associations. For example, the UML notation shown in Figure 10-33 would be interpreted as follows: "A (Student's) Plan of Study *is composed of* many Courses; any given Course *can be included in* many different (Students') Plans of Study."

Figure 10-33. *Reflecting multiplicity on an aggregation*

It makes no sense to reflect multiplicity on inheritance relationships, however, because as discussed in Chapter 4, inheritance implies a relationship between ***classes***, but ***not*** between ***objects***. That is, the notation shown in Figure 10-34 implies that any object belonging to *Subclass B* is also ***simultaneously*** an instance of *Superclass A* by virtue of the "is a" relationship.

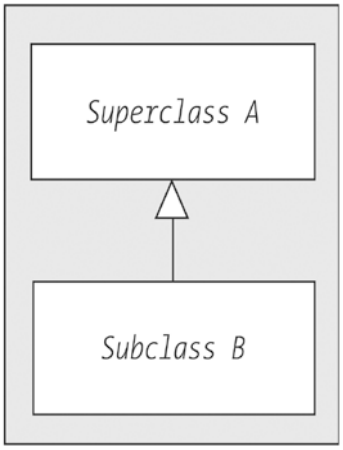


Figure 10-34. *Multiplicity adornments are inappropriate for inheritance relationships*

If we wanted to illustrate some sort of relationship between different objects of types `A` and `B`, for example, "a `Manager` *supervises* an `Employee`," we'd need to introduce a separate association between these classes independent of their inheritance relationship, as shown in Figure 10-35.

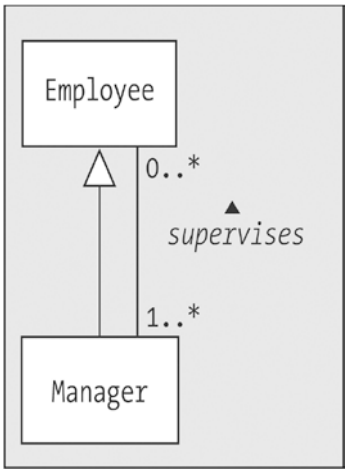


Figure 10-35. *Indicating both inheritance and an association between the* `Manager` *and* `Employee` *classes*

Object/Instance Diagrams

When describing how objects can interact, we sometimes find it helpful to sketch out a scenario of specific objects and their linkages, and for that we create an **object diagram**, a.k.a. **instance diagram**. An instance, or object, looks much the same as a class in UML notation, the main differences being that

- We typically provide both the name of the object and its type, separated by a colon. We underline the text to emphasize that this is an object, not a class (see Figure 10-36).

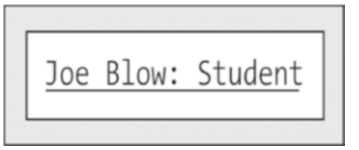


Figure 10-36. *Representing an object*

- The object's type may be omitted if it's obvious from the object's name; for example, the name "student x" implies that the object in question belongs to the `Student` class (see Figure 10-37).

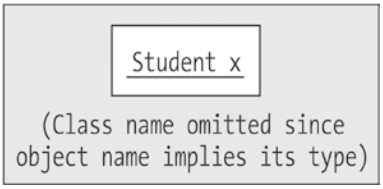


Figure 10-37. *We omit the class name if it's otherwise obvious*

- Alternatively, the object's name may be omitted if we want to refer to a "generic" object of a given type; such an object is known as an ***anonymous object***. Note that we must precede the class name with a colon (:) in such a situation (see Figure 10-38).

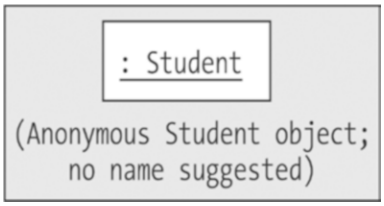


Figure 10-38. *Representing an anonymous object*

Therefore, if we wanted to indicate that Dr. Brown, a `Professor`, is the advisor for three `Students`, we could create the object diagram shown in Figure 10-39.

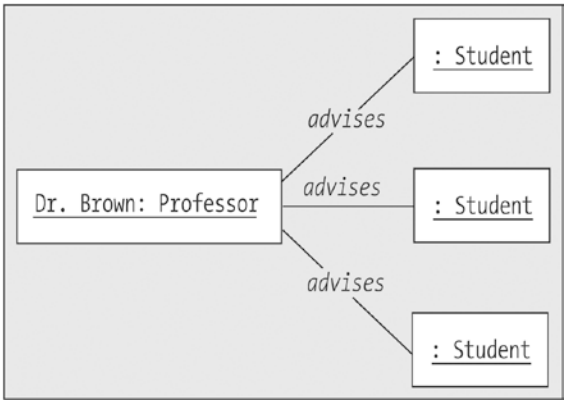


Figure 10-39. *Dr. Brown advises three students*

To reflect that a `Student` by the name of Joe Blow is attending two `Sections` this semester, one of which is also attended by a `Student` named Mary Green, we could create the diagram in Figure 10-40.

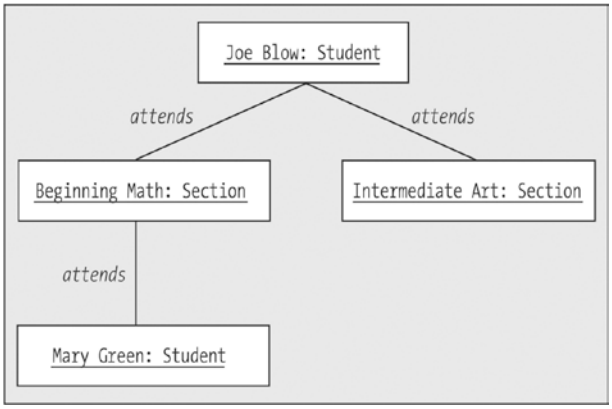


Figure 10-40. *An instance diagram involving numerous objects*

Associations As Attributes

Given Figure 10-41, which shows the association "a `Course` *is offered as* a `Section`," we see that a `Course` object can be related to many different `Section` objects, but that any one `Section` object can be related to a ***single*** `Course` object.

Figure 10-41. *A one-to-many association between the* `Course` *and* `Section` *classes*

By way of review, what does it mean for two objects to be related? It means that they maintain "handles" on one another so that they can easily find one another to communicate and collaborate, a concept that we covered in detail in Chapter 4. If we were to sketch out the attributes of the `Course` and `Section` classes based solely on the diagram in Figure 10-41, we'd need to allow for these handles as reference variables, as follows:

```
public class Section {
  // Attributes.
```

```
  private Course represents;   // A "handle" on a single related Course
                                                    // object.

  // etc.
}

public class Course {
  // Attributes.
  private Collection<Section offeredAs;   // A collection of
related Section
                                                  // object "handles."
  // etc.
}
```

So we see that the presence of an association between two classes A and B in a class diagram implies that class A ***potentially*** has an attribute declared to be either

- A reference to a ***single*** instance/object of type B
- A ***collection*** of references to ***many*** objects of type B

depending on the multiplicity involved and vice versa. I say "potentially" because, when we get to the point of actually programming this application, we may or may not wish to code this relationship bidirectionally, even though at the analysis stage all associations are presumed to be bidirectional. We'll look at the pros and cons of coding bidirectional relationships in Chapter 14.

Because the presence of an association line implies attributes as object references in both related classes, it's inappropriate to additionally list such attributes in the attributes compartment of the respective classes (see Figure 10-42).

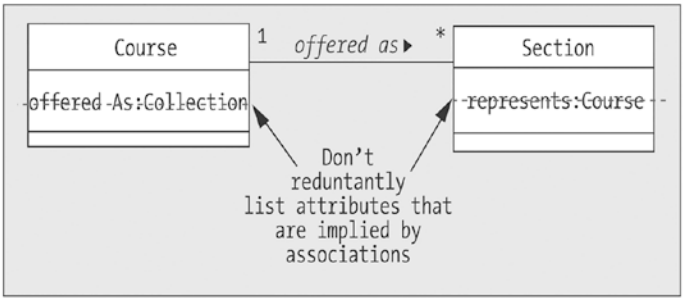


Figure 10-42. *Redundantly reflecting references as attributes is inappropriate; the presence of an association implies these*

This is a mistake commonly made by beginners. The biggest resultant problem with doing so arises when using the code generation capability of a CASE tool: if the attribute is listed explicitly in a class's attributes compartment and is also implied by an association, it may appear in the generated code ***twice***, as shown in the following snippet representing code that might be generated from the erroneous UML diagram shown in Figure 10-42:

```
public class Course {

    // Redundant attributes are generated in the code.

    Collection<Section> offeredAs;   // by virtue of an
    explicit attribute

    Collection<Section> offered_as;  // by virtue of the
    association

    // etc.

    }
```

Information "Flows" Along an Association "Pipeline"

Beginning modelers also tend to make the mistake of introducing undesired redundancy when it comes to attributes in general. In the association portrayed in Figure 10-43, we see that the `name` attribute of the `Professor` class is inappropriately mirrored by the `chairmanName` attribute of the `Department` class.

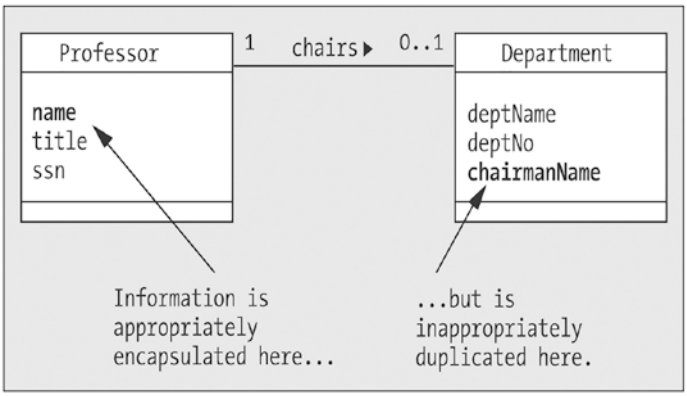


Figure 10-43. *The* `name` *and* `chairmanName` *attributes are redundant*

While it's true that a `Department` object needs to know the name of the `Professor` object that chairs that `Department`, it's inappropriate to explicitly create a `chairmanName` attribute to reflect this information. Because the `Department` object maintains a reference to its associated `Professor` object as an attribute, the `Department` has ready access to this information any time it needs it, simply by invoking the `Professor` object's `getName` method. This piece of information is rightfully encapsulated in the `Professor` class, where it belongs, and shouldn't be duplicated anywhere else. A corrected version of the preceding diagram is shown in Figure 10-44, with the redundancy eliminated.

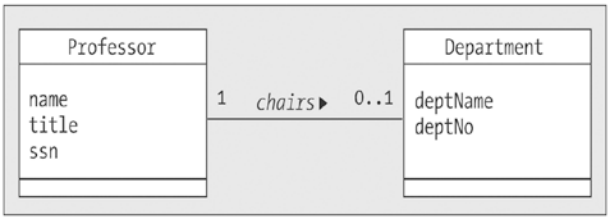


Figure 10-44. *The redundancy of Figure 10-43 has been eliminated*

In essence, whenever you see an association/aggregation line in a diagram, you can think of this as a conceptual "pipeline" across which information can "flow" between related objects as needed.

> At the analysis stage, we don't typically worry about the accessibility (public, private) of attributes or about the directionality of associations—we usually assume that the values of all of the attributes reflected in a diagram are obtainable by calling the appropriate "get" methods on an object.

Sometimes, this "pipeline" extends across ***multiple*** objects, as illustrated by the next example.

In Figure 10-45, we have a diagram involving three classes.

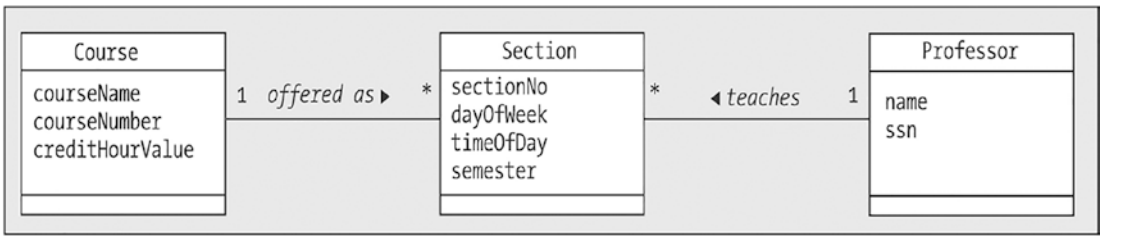


Figure 10-45. *An association "pipeline" between the* `Course`*,* `Section`*, and* `Professor` *classes*

Let's say that someone wishes to obtain a list of all of the `Professors` who have ever taught the `Course` entitled "Beginning Objects." Because each `Course` object maintains a handle on all of its `Section` objects, past and present, the `Course` object representing Beginning Objects can ask each of its `Section` objects the name of the `Professor` who previously taught, or is currently teaching, that `Section`. The `Section` objects, in turn, each maintain a handle on the `Professor` object who taught/teaches the `Section` and can use the `Professor` object's `getName` method to retrieve the name. So information flows along the association "pipeline" from the `Professor` objects to their associated `Section` objects and from there back to the `Course` object that we started with (see Figure 10-46).

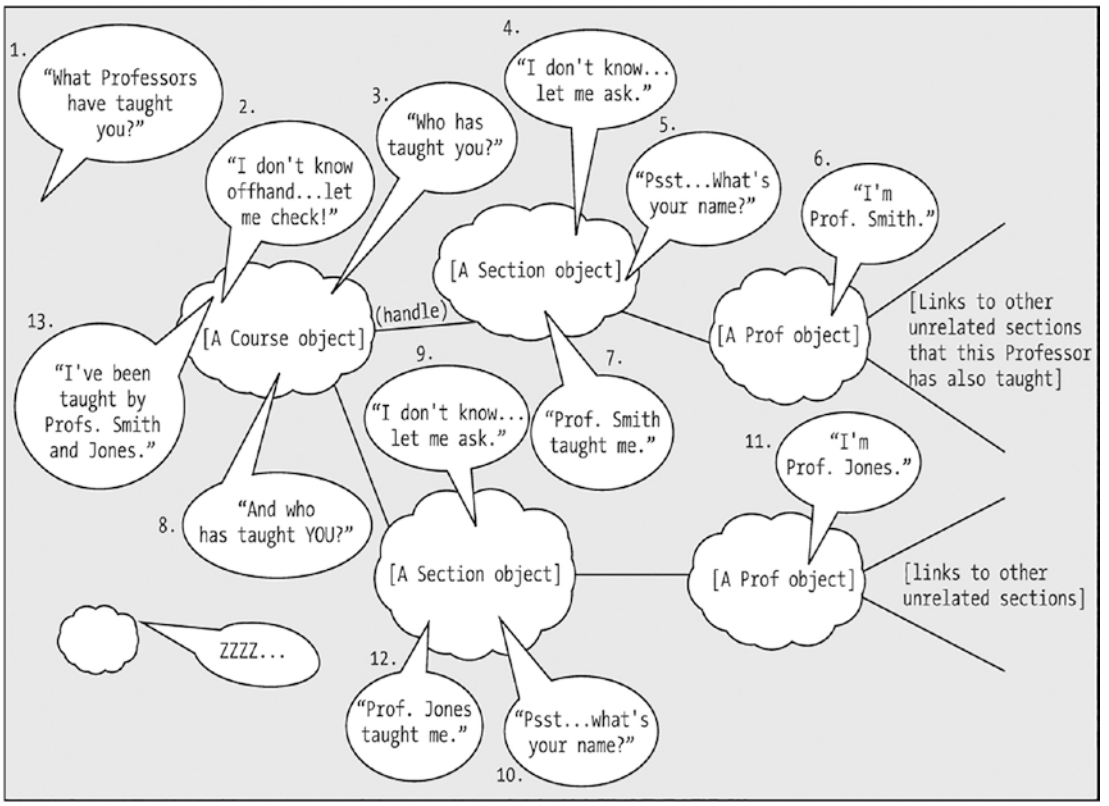


Figure 10-46. *Association "pipelines" can be quite elaborate!*

You'll learn a formal, UML-appropriate way to analyze and depict such "object conversations" in Chapter 11.

These three classes' attributes are modeled in the code that follows, highlighting all of the association-driven attributes:

```
public class Course {
  // Attributes.
  // Pseudocode.
  private Collection<Section> offeredAs;   // a collection of
                                              Section object

// "handles"
```

```
  private String courseName;
  private int courseNumber;
  private double creditHourValue;
  // etc.
}

public class Section {
  // Attributes.
  private Course represents;   // a "handle" on the related Course
                                                    // object
  private int sectionNo;
  private String dayOfWeek;
  private String timeOfDay;
  private String semester;
  private Professor taughtBy;  // a "handle" on the related Prof. object

  // etc.
}

public class Professor {
  // Pseudocode.
  private Collection<Section> sectionsTaught; // a collection of
Section obj.

// "handles"
  private String name;
  private String ssn;

  // etc.
}
```

If we knew that the `Course` class was going to regularly need to know who all the `Professors` were that had ever taught the `Course`, we might decide to introduce the redundant association "a `Professor` *has taught* a `Course`" into our diagram, as illustrated in Figure 10-47.

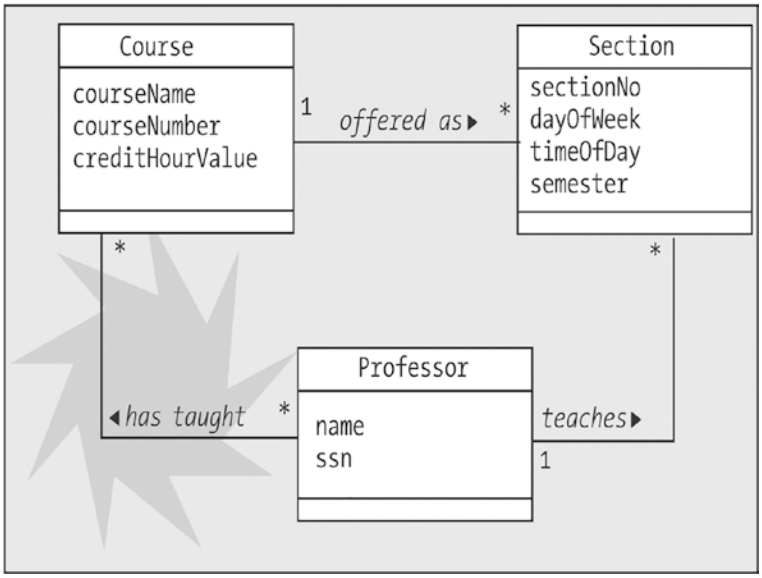


Figure 10-47. *We add redundant associations when objects frequently need a more direct "pipeline" for communication*

This has the advantage of improving the speed with which a `Course` object can determine who has ever taught it: with the addition of the redundant association in Figure 10-47, `Course` objects can now talk ***directly*** to `Professor` objects without using `Section` objects as "go-betweens"—but the cost of this performance improvement is that we've just introduced additional complexity to our application, reflected by the highlighted additions to the following code:

```
public class Course {
  // Attributes.
  // Pseudocode.
  private Collection<Section> offeredAs;      // a collection of
Section object

// "handles"
  private String courseName;
  private int courseNumber;
  private float creditHourValue;
  // Pseudocode.
```

```
  private Collection<Professor> professors;    // a collection of
                                                  Professor obj.

// "handles"
  // etc.
}

public class Section {
  // Attributes.
  private Course represents;   // a "handle" on the related Course
                               // object
  private int sectionNo;
  private String dayOfWeek;
  private String timeOfDay;
  private String semester;
  private Professor taughtBy;  // a "handle" on the related Prof. object

  // etc.
}

public class Professor {
  // Pseudocode.
  private Collection<Course> coursesTaught;     // a collection of
                                                   Course obj.

// "handles"
  private Collection<Section> sectionsTaught;      // a collection of
                                                      Section obj.

// "handles"
  private String name;
  private String ssn;

  // etc.
}
```

By adding the redundant association, we now have extra work to do in terms of maintaining referential integrity. That is, if a different `Professor` is assigned to teach a particular `Section`, we have two links to update rather than one: the link between the `Professor` and the `Section` and the link between the `Professor` and the related `Course`.

The bottom line is that deciding which associations to include, and which to eliminate as derivable from others, is similar to the decision of which web pages you might wish to create a bookmark for in your web browser: you bookmark those that you visit frequently, and type out the URL longhand, or alternatively traverse a ***chain*** of links, for those that you only occasionally need to access. The same is true for object linkages: the decision of which to implement in code depends on which "communication pathways" through the application you're going to want to use most frequently. You'll get a much better sense of what these communication patterns are when we move on to modeling object ***behaviors*** in Chapter 11.

"Mixing and Matching" Relationship Notations

It's possible to intertwine the various relationship types in some rather sophisticated ways. To appreciate this fact, let's study the model in Figure 10-48 to see what it's telling us.

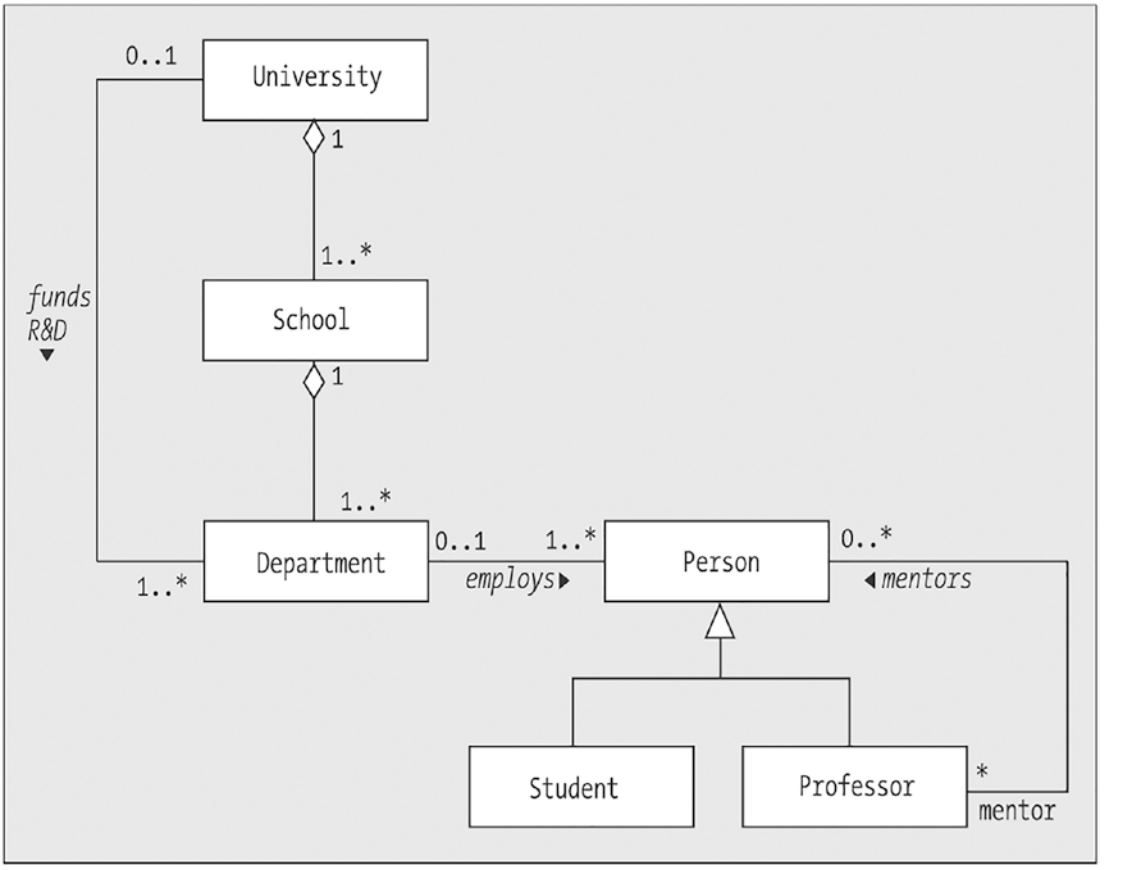


Figure 10-48. *A sample UML model*

First of all, we see some familiar uses of aggregation and inheritance:

- The use of aggregation in the upper-left corner of the diagram—a two-tier aggregation—communicates the facts that a `University` is comprised of one or more `Schools` and that a `School` is comprised of one or more `Departments`, but that any one `Department` is associated with only a single `School` and any one `School` is associated with only a single `University`.
- The use of inheritance in the lower-right corner of the diagram indicates that `Person` is the common base class for both `Student` and `Professor`. Stated another way, a `Student` *is a* `Person`, and a `Professor` *is a* `Person`.

The first "interesting" use of the notation that we observe is that an association can be used to relate classes at differing levels in an aggregation, as in the use of the *funds R&D (research and development)* association used to relate the `University` and `Department` classes. This indicates that the `University` funds one or more `Departments` for research and development purposes, but that a ***given*** `Department` may or may not be funded for R&D.

Next, we note the use of the *employs* association to relate the `Department` and `Person` classes, indicating that a `Department` *employs* one or more `Persons`, but that a given `Person` *may work for* only one `Department`, if indeed that `Person` works for ***any*** `Department` at all.

Because `Person` is the superclass of both the `Student` and `Professor` subclasses, then by virtue of the "is a" relationship, anything we can say about a `Person` must also be true of its derived classes. Therefore

- ***Associations/aggregations that a base class participates in are inherited by its derived classes.*** (This makes sense, because we now know that associations are rendered as reference variable attributes.) Thus, a given `Student` may optionally *work for* one `Department`, perhaps as a teaching assistant, and a given `Professor` may optionally *work for* one `Department`, because `Student` and `Professor` are derived from `Person`.
- Also, because we can deduce (via the aggregation relationship) which `School` and `University` a given `Department` belongs to, the fact that a `Person` *works for* a given `Department` also implies which `School` and `University` the `Person` *works for*.

Finally, we note that an association can be used to relate classes at differing levels in an inheritance hierarchy, as in the use of the *mentors* association to relate the `Person` and `Professor` classes. Here, we're stating that a `Professor` optionally *mentors* many `Persons`—`Students` and/or `Professors`—and conversely that a `Person`—either a `Student` or a `Professor`—is mentored by optionally many `Professors` specifically. We label the end of the association line closest to the `Professor` class with the role designation "mentor" to emphasize that `Professors` are mentors at the `University`, but that `Persons` in general (i.e., `Students`) are not.

What if we instead wanted to reflect the fact that ***both*** `Students` and `Professors` may serve in the capacity of a mentor? We could substitute a reflexive association on the `Person` class, as shown in Figure 10-49, which, by virtue of inheritance, actually implies four relationship possibilities:

- A `Professor` mentoring a `Student`
- A `Professor` mentoring another `Professor`
- A `Student` mentoring another `Student`
- A `Student` mentoring a `Professor` (which is not very likely)

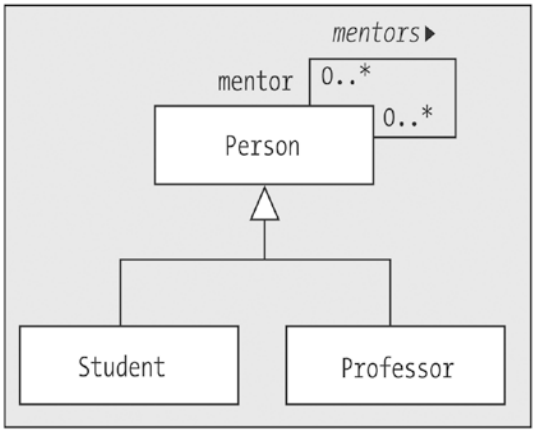


Figure 10-49. *Various possible "mentorship" associations are implied*

If we wanted to reflect that only the first three of these are possible, we'd have to resort to the rather more complex version shown in Figure 10-50, where the three relationships of interest are all reflected as separate association lines (two reflexive, one binary).

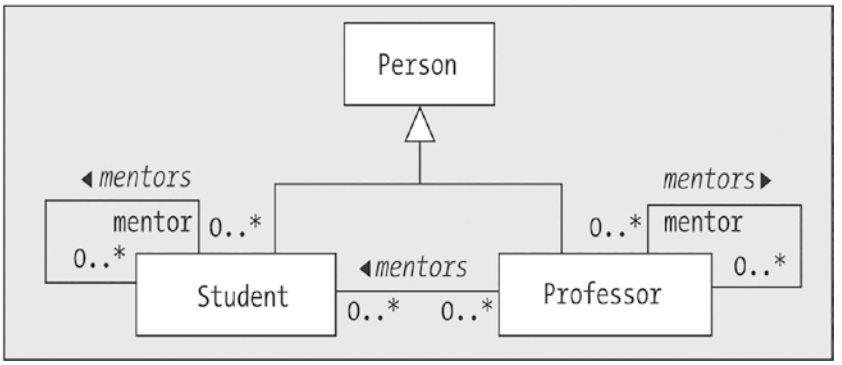


Figure 10-50. *Specific mentor associations made explicit*

As cumbersome as it is to change the diagram to reflect these refinements in our understanding, it would be orders of magnitude more painful to change the software once the application has been coded.

Association Classes

We sometimes find ourselves in a situation where we identify an attribute that is critical to our model, but which doesn't seem to fit nicely into any one class. As an example, let's revisit the (many-to-many) association "a `Student` *attends* a `Section`," as shown in Figure 10-51. (Note that we're using the "generic" ***many*** multiplicity symbol this time, a single asterisk (*), at each end of the association line.)

Figure 10-51. *A many-to-many association between* `Student` *and* `Section`

At the end of every semester, a student receives a letter grade for every section that they attended during that semester. We decide that the grade should be represented as a `String` attribute (e.g., "A-", "C+"). However, where does the "grade" attribute belong?

- It's not an attribute of the `Student` class, because a student doesn't get a single overall grade for all of their coursework, but rather an individual grade for each section attended.

- It's not an attribute of the `Section` class, either, because not all students attending a section typically receive the same letter grade.

If we think about this situation for a moment, we realize that the grade is actually an attribute of the ***pairing*** of a given `Student` object with a given `Section`; that is, it's an attribute of the ***link*** that exists between these two objects.

With UML, we create a separate class, known as an **association class**, to house attribute(s) belonging to the link between objects and attach it with a ***dashed*** line to the association line, as shown in Figure 10-52.

Because association classes represent attributes of a link between two objects, these are sometimes informally referred to as "**link attributes**."

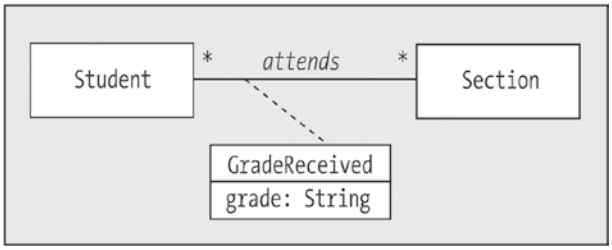


Figure 10-52. *Placing an association class on a many-to-many association*

Any time you see an association class in a class diagram, realize that there is an alternative equivalent way to represent the same situation ***without*** using an association class:

- In the case of a ***many-to-many association*** involving an association class, we can split the many-to-many association into two one-to-many associations, inserting what was formerly the association class as a "standard" class between the other two classes. Doing this for the preceding *attends* association, we wind up with the alternative equivalent representation in Figure 10-53.

  One important point to note is that the "many" ends of these two new associations reside with the newly inserted class, because a `Student` *receives* many `Grades` and a `Section` *issues* many `Grades`.

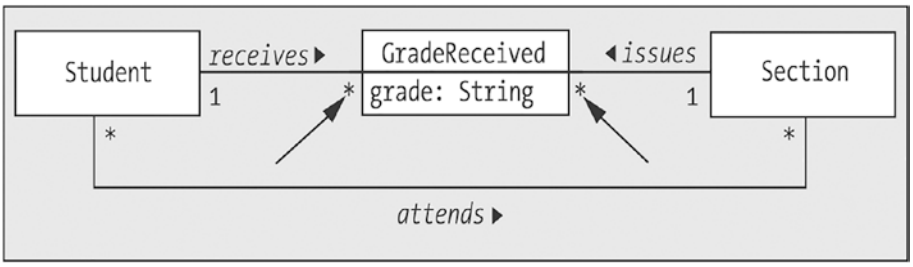


Figure 10-53. *An alternative representation for Figure 10-52*

- If we happen to have an association class for a ***one-to-many association***, as in the *works for* association between `Professor` and `Department` in Figure 10-54, then the association class's attribute(s) can, in theory, be "folded into" the class at the "many" end of the association instead, and we can do away with the association class completely as shown in Figure 10-55.

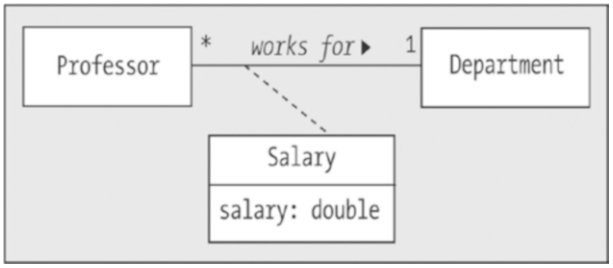


Figure 10-54. *Placing an association class on a one-to-many association*

Figure 10-55. *An alternative representation for Figure 10-54*

- With a ***one-to-one association***, we can fold the association class's attributes into ***either*** of the two classes.

That being said, this practice of folding association class attributes into one end of a one-to-many or one-to-one association is discouraged, however, because it reduces the amount of information communicated by the model. In the preceding example, the only reason that a `Professor` has a `salary` attribute is because he or she *works for* a `Department`; knowledge of this "cause-and-effect" connection between employment and salary is lost if the association class is eliminated as such from the model.

Note that association classes may themselves participate in relationships with other classes. The diagram in Figure 10-56, for example, shows the association class `Role` participating in a one-to-many association with the class `USPresident`; an example illustrating this model would be "Film Star Anthony Hopkins starred in the movie *Nixon* in the role of Richard M. Nixon, thus portraying the ***real*** former U.S. President Richard M. Nixon."

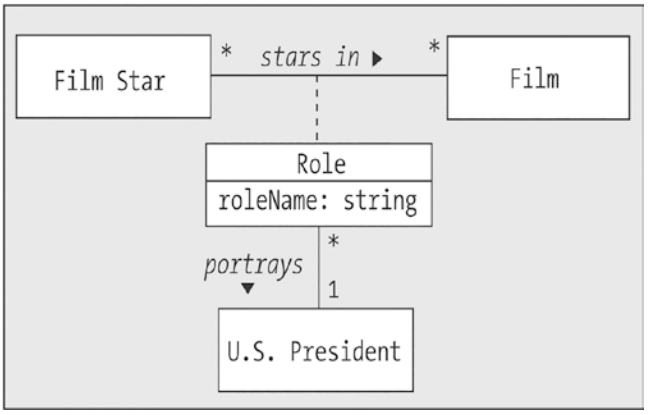


Figure 10-56. *Association classes themselves can participate in associations with other classes*

Our "Completed" Student Registration System Class Diagram

By applying all that we've covered in this chapter about static modeling, we can produce the UML class diagram for the SRS shown in Figure 10-57. Of course, as I've said repeatedly, this isn't the only correct way to model the requirements, nor is it necessarily the "best" model that we could have produced; but it is an accurate, concise, and correct model of the static aspects of the problem to be automated.

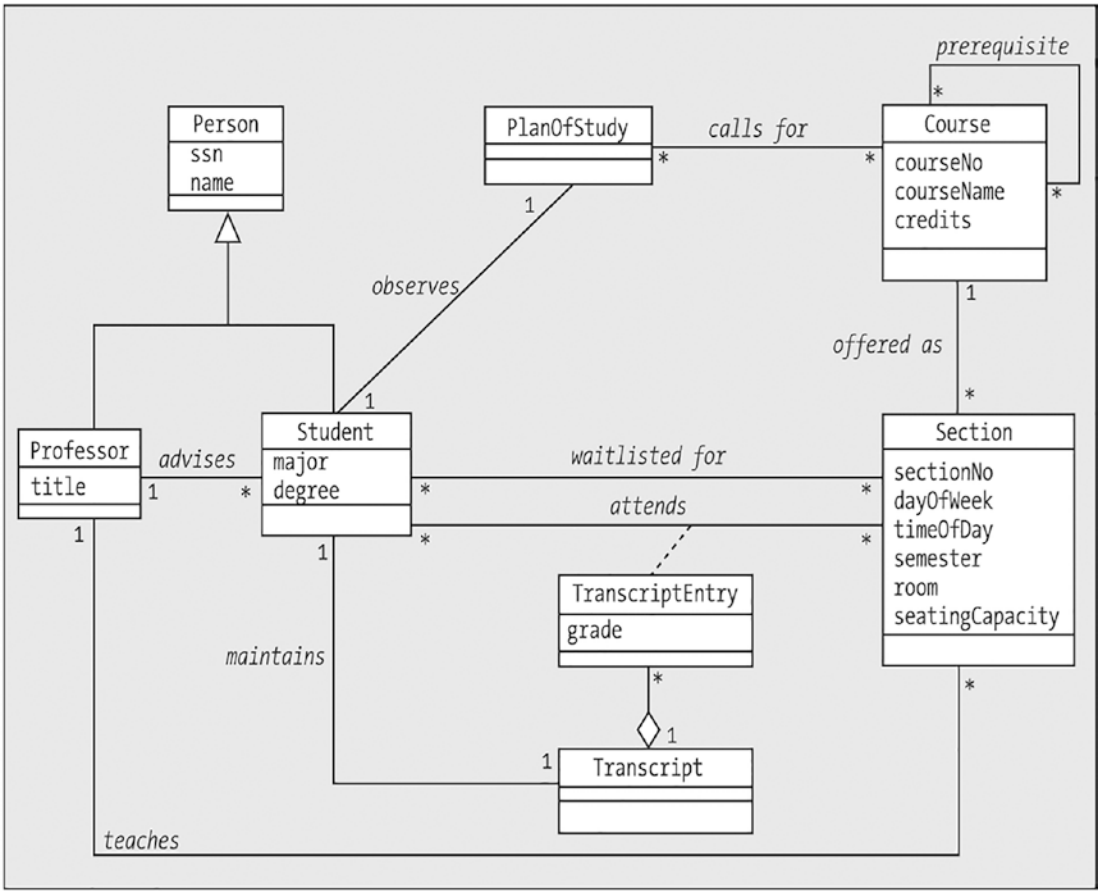


Figure 10-57. *Our "completed" SRS class diagram*

There are a few things worth noting.

We opted to use the "generic" ***many*** notation (* for the UML) rather than specifying `0..*` or `1..*`; this is often adequate during the initial modeling stages of a project.

Note that we've reflected two separate many-to-many associations between the `Student` and `Section` classes: *waitlisted for* and *attends*. A given `Student` may be waitlisted for many different `Sections`, and they may be registered for/attending many ***other*** `Sections`. What this model doesn't reflect is the fact that a `Student` is ***not*** permitted to simultaneously be attending and waitlisted for the ***same*** `Section`. Constraints such as these can be reflected as textual notes on the diagram, enclosed in curly braces, or omitted from the diagram but spelled out in the data dictionary.

In the diagram excerpt in Figure 10-58, we use the annotation { xor } to represent an "exclusive or" situation between the two associations: a Student can either be *waitlisted* for or *attending* a given Section, but not both.

As mentioned earlier in this chapter, we're able to get by with a single *attends* association to handle both the Sections that a Student is currently attending and those that they have attended in the past. The date of attendance—past or present—is reflected by the "semester" attribute of the Section class; also, for any courses that are currently in progress, the value of the "grade" attribute of the TranscriptEntry association class would be as of yet undetermined.

We could have also reflected an association class on the *waitlisted for* association representing a given Student's position in the waitlist for a particular Section, and then we could have gone on to model the notion of a WaitList as an aggregation of WaitListEntry objects (see Figure 10-58).

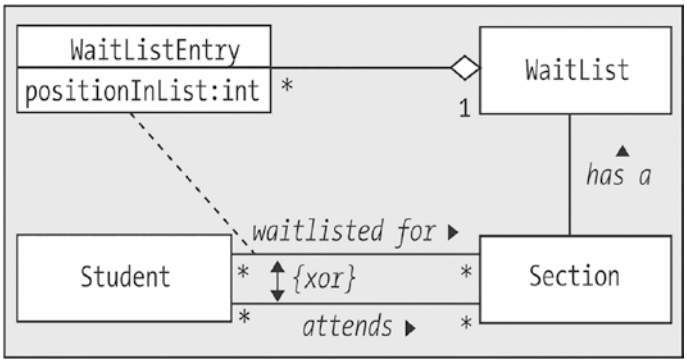


Figure 10-58. *A* WaitList *can be modeled as an aggregation of* WaitListEntry *objects*

Since we're going to want to use the object model to gain user confirmation that we understand their primary requirements, we needn't clutter the diagram with such behind-the-scenes implementation details just yet, however.

We renamed the association class for the *attends* relationship; it was introduced earlier in this chapter as GradeReceived, but is now called TranscriptEntry. We've also introduced an aggregation relationship between the TranscriptEntry class and another new class called Transcript (see Figure 10-59).

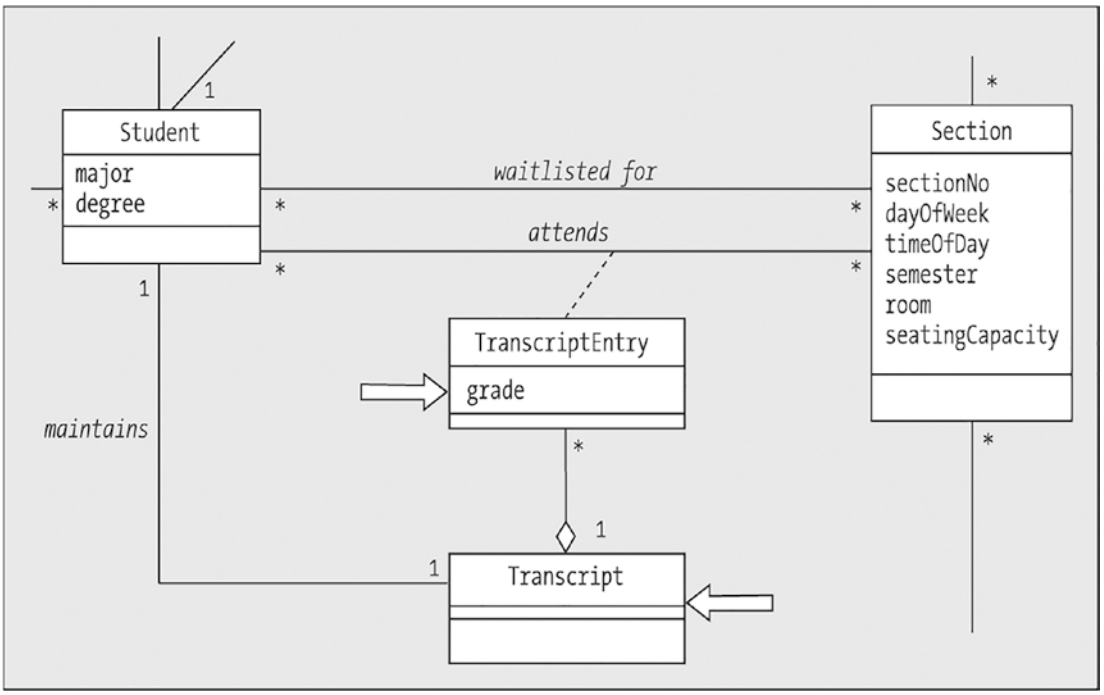


Figure 10-59. *A* `Transcript` *is an aggregation of* `TranscriptEntry` *objects*

Let's explore how all of this evolved.

When the *attends* association was first introduced earlier in this chapter, we portrayed it as shown in Figure 10-60.

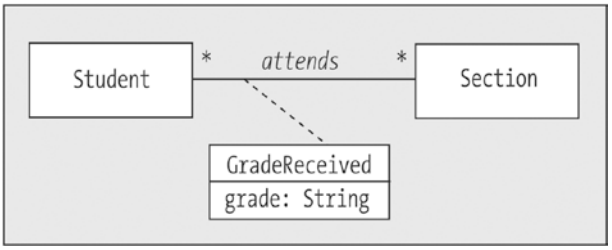


Figure 10-60. *Initial portrayal of the attends association*

We then realized that it could equivalently be represented as a ***pair*** of one-to-many associations *issues* and *receives* (see Figure 10-61).

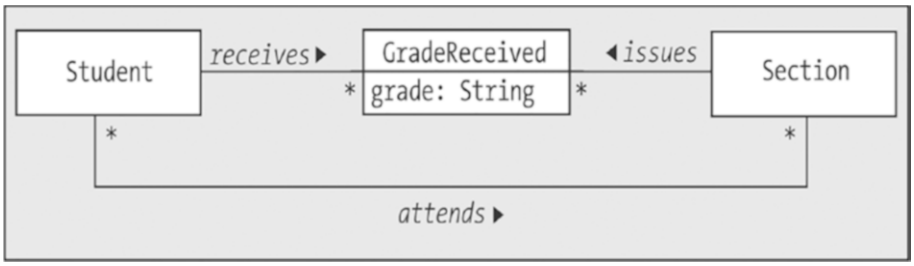


Figure 10-61. *The attends association may be portrayed alternatively as issues and receives*

In this alternative form, it's clear that any individual `GradeReceived` object maintains one handle on a `Student` object and another handle on a `Section` object and can ask either of them for information whenever necessary. The `Section` object, in turn, maintains a handle on the `Course` object that it represents by virtue of the *offered as* association. It's a trivial matter, therefore, for the `GradeReceived` object to request the values of attributes `semester`, `courseNo`, `courseName`, and `credits` from the `Section` object (which would in turn have to ask its associated `Course` object for the last three of these four values); this is illustrated conceptually in Figure 10-62.

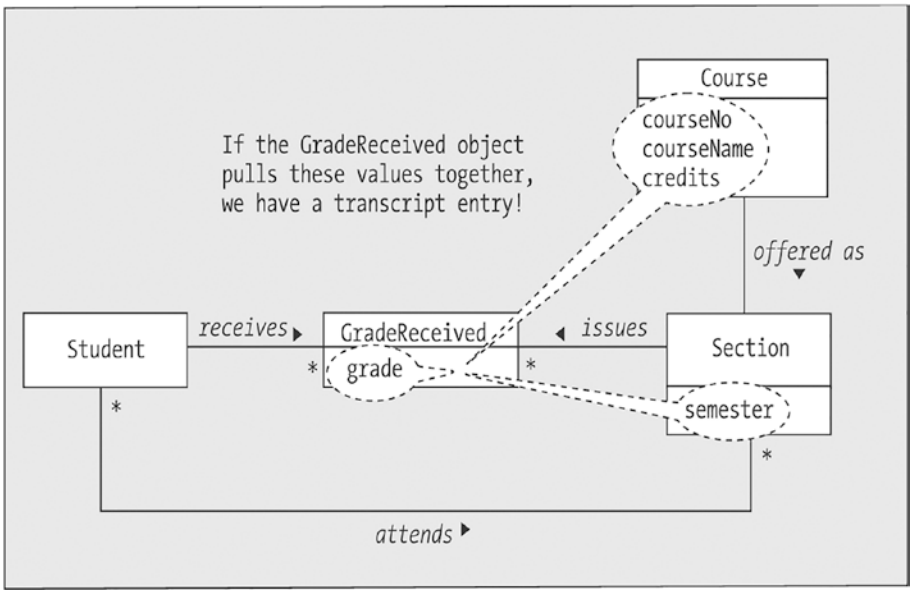
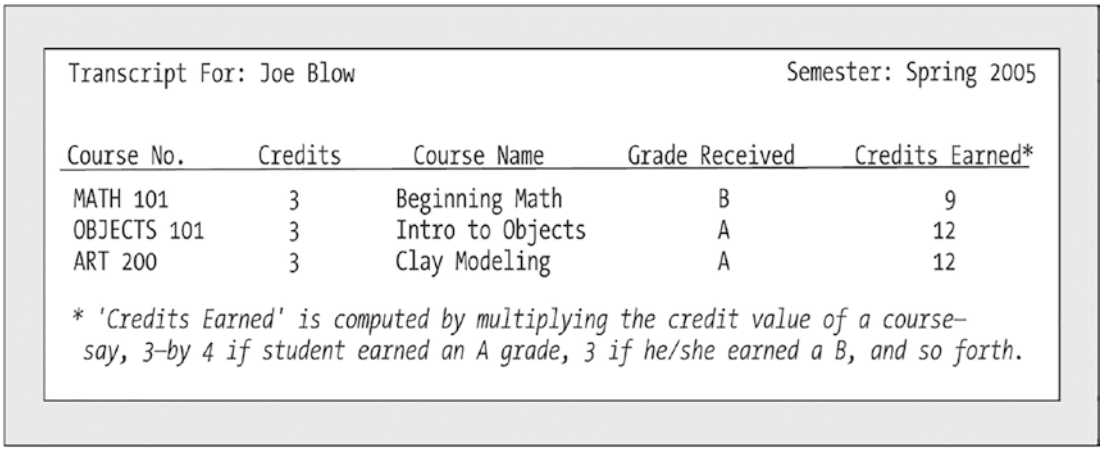


Figure 10-62. *`GradeReceived` has access to all of the makings of a `TranscriptEntry`*

If the GradeReceived object pulls these values together, we have everything that we need for a line item entry on a student's transcript, as shown in Figure 10-63.



Transcript For: Joe Blow — Semester: Spring 2005

Course No.	Credits	Course Name	Grade Received	Credits Earned*
MATH 101	3	Beginning Math	B	9
OBJECTS 101	3	Intro to Objects	A	12
ART 200	3	Clay Modeling	A	12

* 'Credits Earned' is computed by multiplying the credit value of a course—say, 3—by 4 if student earned an A grade, 3 if he/she earned a B, and so forth.

Figure 10-63. *A sample transcript report*

Therefore, we see that renaming the association class from GradeReceived to TranscriptEntry makes good sense. It was then a natural step to aggregate these into a Transcript class.

Our SRS diagram is a little "light" in terms of attributes; we've reflected only those that we'll minimally need when we code the model layer of the SRS in Part 3.

Of course, we now need to go back to the data dictionary to capture definitions of all of the new attributes, relationships, and classes that we've identified in putting together this model. The following sidebar shows our revised SRS data dictionary.

THE REVISED SRS DATA DICTIONARY

Classes

- ***Course:*** A semester-long series of lectures, assignments, exams, etc. that all relate to a particular subject area and that are typically associated with a particular number of credit hours, a unit of study toward a degree. For example, Beginning Objects is a required **course** for the Master of Science degree in information systems technology.
- ***Person:*** A human being associated with the university.

- ***PlanOfStudy:*** A list of the **courses** that a student intends to take to fulfill the **course** requirements for a particular degree.
- ***Professor:*** A member of the faculty who teaches **sections** or advises **students**.
- ***Section:*** The offering of a particular **course** during a particular semester on a particular day of the week and at a particular time of day. (For example, **course** Beginning Objects is taught in the Spring 2025 semester on Mondays from 1:00 to 3:00 p.m.)
- ***Student:*** A person who is currently enrolled at the university and who is eligible to register for one or more **sections**.
- ***Transcript:*** A record of all of the **courses** taken to date by a particular **student** at this university, including which semester each **course** was taken in, the grade received, and the credits granted for the **course**, as well as reflecting an overall total number of credits earned and the **student's** grade point average (GPA).
- ***TranscriptEntry:*** A single line item from a **transcript**, reflecting the **course** number and name, semester taken, value in credit hours, and grade received.

Relationships

- ***Advises—a professor advises a student:*** A professor is assigned to oversee a student's academic pursuits for the student's entire academic career, leading up to their attainment of a degree. An advisor counsels their advisees regarding course selection, professional opportunities, and any academic problems the student might be having.
- ***Attends—a student attends a section:*** A student registers for a section, attends class meetings for a semester, and participates in all assignments and examinations, culminating in the award of a letter grade representing the student's mastery of the subject matter.
- ***Calls for—a plan of study calls for a course:*** A student may take a course only if it's called out by their plan of study. The plan of study may be amended, with a student's advisor's approval.

- ***Maintains—a student maintains a transcript:*** Each time a student completes a course, a record of the course and the grade received is added to the student's transcript.
- ***Observes—a student observes a plan of study:*** See notes for the ***calls for*** association.
- ***Offered as—a course is offered as a section:*** The same course can be taught numerous times in a given semester and of course over numerous semesters for the "lifetime" of a course—that is, until such time as the subject matter is no longer considered to be of value to the student body or there is no qualified faculty to teach the course.
- ***Prerequisite—a course is a prerequisite for another course:*** If it's determined that the subject matter of a course A is necessary background to understanding the subject matter of a course B, then course A is said to be a prerequisite of course B. A student typically may not take course B unless they have either successfully completed course A or can otherwise demonstrate mastery of the subject matter of course A.
- ***Teaches—a professor teaches a section:*** A professor is responsible for delivering lectures, assigning thoughtful homework assignments, examining students, and otherwise ensuring that a quality treatment of the subject matter of a course is made available to students.
- ***Waitlisted for—a student is waitlisted for a section:*** If a section is "full"—for example, the maximum number of students signed up for the course based on either the classroom capacity or the student group size deemed effective for teaching—then interested students may be placed on a waitlist, to be given consideration should seats in the course subsequently become available.

(aggregation between Transcript and TranscriptEntry)

(specialization of Person as Professor)

(specialization of Person as Student)

Attributes

- ***Person.ssn:*** The unique Social Security number (SSN) assigned to an individual.
- ***Person.name:*** The person's name, in "last name, first name" order.

- ***Professor.title:*** The rank attained by the professor (e.g., "Adjunct Professor").
- ***Student.major:*** A reflection of the department in which a student's primary studies lie (e.g., Mathematics). (We assume that a student may designate only a single major.)
- ***Student.degree:*** The degree that a student is pursuing (e.g., Master of Science degree).
- ***TranscriptEntry.grade:*** A letter grade of A, B, C, D, or F, with an optional +/– suffix, such as A+ or C–.
- ***Course.courseNo:*** A unique ID assigned to a course, consisting of the department designation plus a unique numeric ID within the department (e.g., MATH 101).
- ***Course.courseName:*** A full name describing the subject matter of a course (e.g., Beginning Objects).
- ***Course.credits:*** The number of units or credit hours a course is worth, roughly equating to the number of hours spent in the classroom in a single week (typically, three credits for a full-semester lecture course).
- ***Section.sectionNo:*** A unique number assigned to distinguish one section/offering of a particular course from another offering of the same course in the same semester (e.g., MATH 101, section no. 1).
- ***Section.dayOfWeek:*** The day of the week on which the lecture course meets.
- ***Section.timeOfDay:*** The time (range) during which the course meets (e.g., 2:00 to 4:00 p.m.).
- ***Section.semester:*** An indication of the scholastic semester in which a section is offered (e.g., Spring 2025).
- ***Section.room:*** The building and room number where the section will be meeting (e.g., Innovation Hall, Room 333).
- ***Section.seatingCapacity:*** The maximum number of students permitted to register for a section.

Metadata

One question that is often raised by beginning modelers is why we don't use an inheritance relationship to relate the `Course` and `Section` classes, rather than using a simple association as we've chosen to do. On the surface, it does indeed seem tempting to want `Section` to be a derived class of `Course`, because all of the attributes listed for a `Course`—`courseNo`, `courseName`, and `creditValue`—also pertain to a `Section`; so why wouldn't we want `Section` to ***inherit*** these, in the same way that `Student` and `Professor` inherit all of the attributes of `Person`? A simple example should quickly illustrate why inheritance isn't appropriate.

Let's say that because "Beginning Object Concepts" is such a popular course, the university is offering three sections of the course for the Spring 2025 semester. We therefore instantiate one `Course` object and three `Section` objects. If `Section` were a derived class of `Course`, then all ***four*** objects would carry `courseNo`, `courseName`, and `creditValue` attributes. Filling in the attribute values for these four objects, as shown in Table 10-3, we see that there is quite a bit of repetition in the attribute values across these four objects: we've ***repeated*** the same `courseNo`, `courseName`, and `creditValue` attribute values four times. That's because the information contained within a `Course` object is common to, and hence describes, ***numerous*** `Section` objects.

Table 10-3. *Duplication of Data Across Four Object Instances*

Attribute Name	Attribute Values for the Course Object
courseName	Beginning Object Concepts
courseNo	OBJECTS 101
creditValue	3

Attribute Name	Attr. Values for Section Object #1	Attr. Values for Section Object #2	Attr. Values for Section Object #3
courseName	Beginning Object Concepts	Beginning Object Concepts	Beginning Object Concepts
courseNo	OBJECTS 101	OBJECTS 101	OBJECTS 101
creditValue	3	3	3
studentsRegistered	(To be determined)	(To be determined)	(To be determined)
instructor	Reference to professor X	Reference to professor Y	Reference to professor Z
semesterOffered	Spring 2025	Spring 2025	Spring 2025
dayOfWeek	Monday	Tuesday	Thursday
timeOfDay	7:00 p.m.	4:00 p.m.	6:00 p.m.
classroom	Hall A, Room 123	Hall B, Room 234	Hall A, Room 345

To reduce redundancy and to promote encapsulation, we should eliminate ***inheritance*** of these attributes and instead create only one instance of a `Course` object for *n* instances of its related `Section` objects. We can then have each `Section` object maintain a handle on the common `Course` object so as to retrieve these shared values whenever necessary. This is precisely what we've modeled via the one-to-many *offered as* association.

Whenever an instance of some class `A` encapsulates information that describes numerous instances of some other class `B` (such as `Course` does for `Section`), we refer to the information contained by the `A` object (`Course`) as **metadata** relative to the `B` objects (`Sections`).

Summary

Our object model has started to take shape! We have a good idea of what the static structure needs to be for the SRS—the classes, their attributes, and their relationships with one another—and are able to communicate this knowledge in a concise, graphical form. There are many more embellishments to the UML notation that we haven't covered in this chapter, but we've examined the core concepts that will suffice for most "industrial-strength" modeling projects.

There is an obvious "hole" in our class diagram, however: all of our classes have empty operations compartments. We'll address this deficiency by learning some complementary modeling techniques for determining the ***dynamic behavior*** of our objects in Chapter 11.

In this chapter, you learned

- The noun phrase analysis technique for identifying candidate domain classes
- The verb phrase analysis technique for determining potential relationships among these classes
- That coming up with candidate classes is a bit subjective and hence that we must remain flexible, revisiting our model numerous times until we—and our users—are satisfied with the outcome
- The importance of producing a data dictionary as part of a project's documentation set
- How to graphically portray the static structure of a model as a class diagram using UML notation
- The importance of having an experienced object modeling mentor available to a project team

EXERCISES

1. Come up with a list of candidate classes for the Prescription Tracking System (PTS) case study presented in the Appendix, as well as an association matrix.
2. Develop a class diagram for the PTS case study, using UML notation. Reflect all significant attributes and relationships among classes, including the appropriate multiplicity. Ideally, you should use an object modeling software tool if you have one available to you.
3. Prepare a data dictionary for the PTS, to include definitions of all classes, attributes, and associations.
4. Devise a list of candidate classes for the problem area whose requirements you defined for Exercise 3 in Chapter 1, as well as an association matrix.
5. Develop a class diagram for the problem area whose requirements you defined for Exercise 3 in Chapter 1, using UML notation. Reflect all significant attributes and relationships among classes, including the appropriate multiplicity. Ideally, you should use an object modeling software tool if you have one available to you.
6. Prepare a data dictionary for the problem area whose requirements you defined for Exercise 3 in Chapter 1 to include definitions of all classes, attributes, and associations.

CHAPTER 11

Modeling the Dynamic/ Behavioral Aspects of the System

Thus far, we've been focused on the **static structure** of the problem being modeled—the floor plan for our custom home, as it were. As you learned in Chapter 10, this static structure is communicated via a class diagram plus supporting documentation. The building blocks of a class diagram are

- Classes.
- Associations/aggregations.
- Attributes.
- Generalization/specialization hierarchies (also known as inheritance relationships).
- Operations/methods. ***These are conspicuously absent from our class diagram***. Why? Because they aren't part of the static structure, so we haven't discussed how to determine these yet; this topic is the focus of this chapter.

As I've said many times already, an OO software system is a set of collaborating objects, each with a "life" of its own. If each object went about its own business without regard to what any other object needed it to do, however, utter chaos would reign! The only way that objects can collaborate to perform some overall system mission, such as registering a student for a course, is if each class defines the appropriate methods—***services***—that will enable its instances to fulfill their respective roles in the collaboration.

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_11

In order to determine what these methods/services must be, we must complement our knowledge of the static structure of the system to be built by also modeling the **dynamic** aspects of the situation: that is, the ways in which concurrently active objects interact over time and how these interactions affect each object's state. Producing a dynamic model to complement the static model will not only enable us to determine the methods required for each class but also give us new insights into ways to improve upon the static structure.

In this chapter, you'll learn about the building blocks of a **dynamic model**

- Events
- Scenarios
- Sequence diagrams
- Communication diagrams

and how to use the knowledge gleaned from these modeling artifacts to identify the operations/methods that are needed to complete our class diagram.

How Behavior Affects State

Back in Chapter 3, we defined the ***state*** of an object as the collective set of all of the object's attribute values at a given point in time, including

- The values of all of the "simple" attributes for that object—in other words, attributes that don't represent other domain objects
- The values of all of the reference variable attributes representing links to other domain objects

Table 11-1 repeats our list of `Student` attributes from Chapter 6, with an additional column to indicate which category each attribute falls into.

Table 11-1. *Student Class Attributes*

Attribute Name	Data Type	Represents Link(s) to an SRS Domain Object?
Name	String	No
studentID	String	No
birthDate	Date	No
Address	String	No
Major	String	No
Gpa	double	No
Advisor	Professor	Yes
courseLoad	Collection of Course objects	Yes
Transcript	Collection of TranscriptEntry objects, or Transcript	Yes

In Chapter 10, you learned about UML object diagrams as a way of portraying a "snapshot" of the links between specific individual objects. Let's use an object diagram to reflect the state of a few hypothetical objects within the SRS domain.

Figure 11-1 shows that Dr. Smith (a Professor) works for the Math Department; Dr. Green (another Professor) works for the Science Department; and Bill and Mary, both Students, are majoring in Math and Science, respectively.

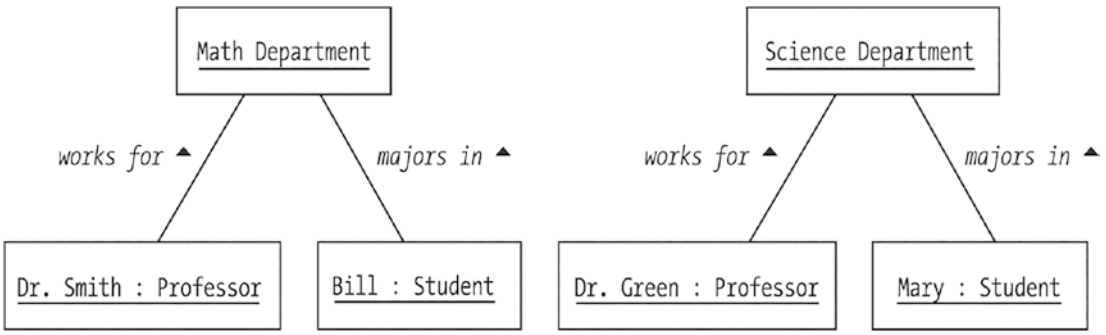


Figure 11-1. *The state of an object includes the links it maintains with other objects*

Bill is dissatisfied with his choice of major and calls Dr. Green, a professor whom he admires, to make an appointment. Bill wants to discuss the possibility of transferring to the Science Department. After meeting with Dr. Green and discussing his situation, Bill indeed decides to switch majors. I've informally reflected these object interactions using arrows on the object diagram, shown in Figure 11-2; as this chapter progresses, you'll learn the official way to portray object interactions in UML notation.

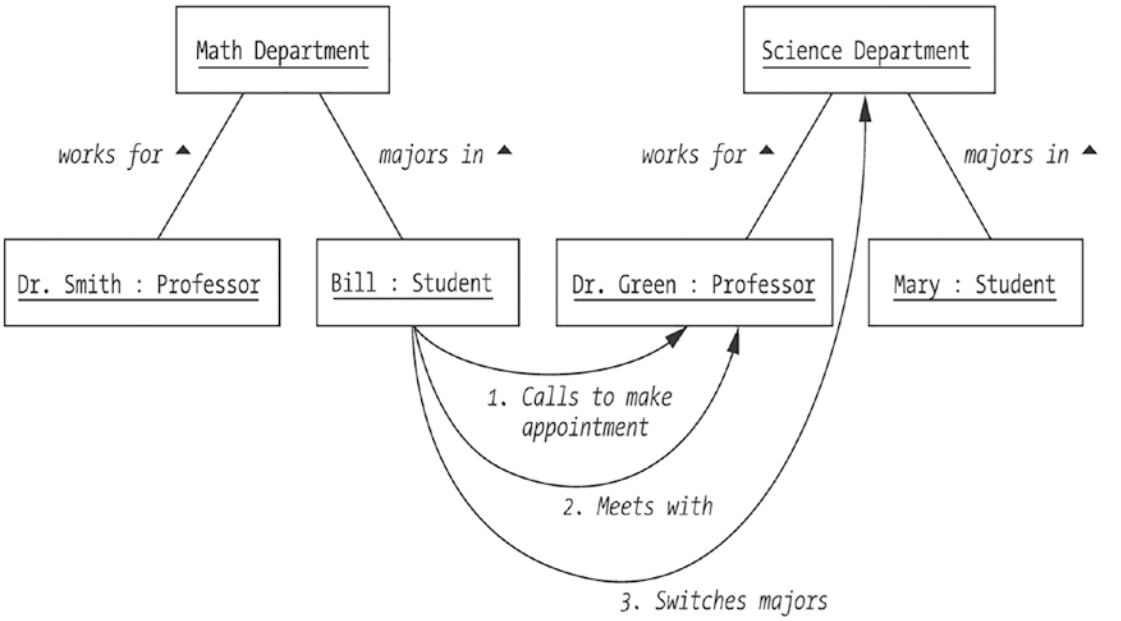


Figure 11-2. *Objects' interactions can affect their state*

When the dust settles from all of this activity, we see that the resultant state of the system has changed, as reflected in the revised object diagram in Figure 11-3. In particular

- Bill's state has changed, because his link to the Math Department object has been replaced with a link to the Science Department object.
- The Math Department object's state has changed, because it no longer has a link to Bill.
- The Science Department's state has changed, because it now has an additional link (to Bill) that wasn't previously there.

Note, however, that although Dr. Green collaborated with Bill in helping him make his decision to switch majors, the state of the "Dr. Green" (`Professor`) object ***has not changed*** as a result of the collaboration.

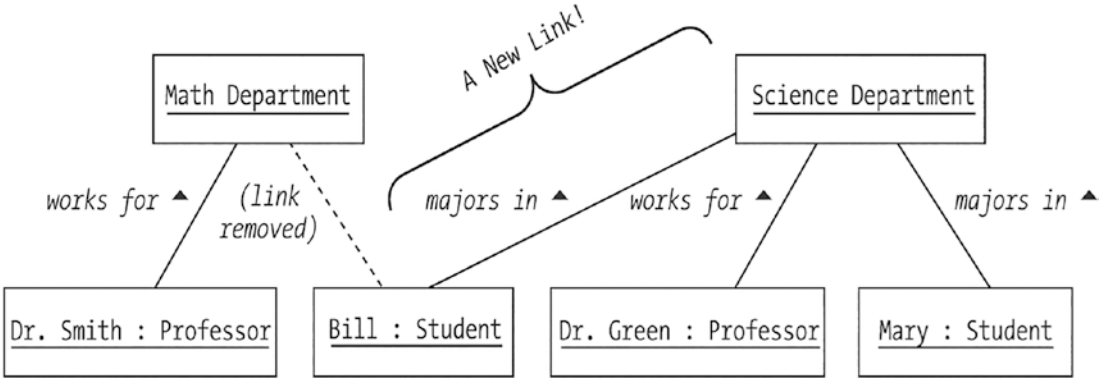


Figure 11-3. *Some interacting objects* ***don't*** *experience a change of state*

So we see that

- Objects' dynamic activities can result in changes to the **static structure of a system**—that is, the states of all of its objects taken collectively.
- However, such activities needn't affect the state of ***all*** objects involved in a collaboration.

Events

You saw in Chapter 4 that object collaborations are triggered by events. By way of review, an ***event*** is an external stimulus to an object, signaled to the object in the form of a ***message*** (method call). An event can be

- ***User initiated*** (e.g., the result of clicking a button or link on a GUI)
- ***Initiated by another computer system*** (e.g., the arrival of information being transferred from the Student Billing System to the Student Registration System)
- ***Initiated by another object within the same system*** (e.g., a `Course` object requesting some service of a `Transcript` object)

When an object receives notification of an event via a message, it may react in one or more of the following ways:

- An object may change its state.
- An object may direct an event (message) toward another object.
- An object may return a value.
- An object may react with the external boundaries of its system.
- An object may seemingly ignore an event.

Let's discuss these five types of reaction in detail, one by one.

An Object May Change Its State

An object may change its state (the values of its "simple" attributes and/or links to other objects), as in the case of a `Professor` object receiving a message to take on a new `Student` advisee, illustrated by the following code snippet:

```
Professor p = new Professor();
Student s = new Student();
// Details omitted.
p.addAdvisee(s);
```

Let's look at the code for the `Professor` class's `addAdvisee` method to see how the `Professor` will respond to this message. We see that the `Professor` object is inserting the reference to `Student` object `s` that it is being handed as an argument into a `Collection` of `Student` object references called `advisees`:

```
public class Professor {
  // Attributes.
  Collection<Student> advisees;  // Holds Student object references.

  // Other details omitted.

  public void addAdvisee(Student s) {
    // Insert s into the advisees collection.
    advisees.add(s);
  }
}
```

In so doing, `Professor` object `p` will have formed a new link of type *advises* with `Student` object `s` (see Figure 11-4).

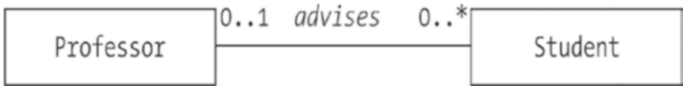


Figure 11-4. *Revisiting the UML diagram for the advises association*

Typical "set" methods fall into this category of event response.

An Object May Direct an Event (Message) Toward Another Object

An object may direct an event (message) toward another object (including, perhaps, the sender of the original message), as in the case of a `Section` object receiving a message to register a `Student`, illustrated by the following code snippet:

```
Section x = new Section();
Student s = new Student();
// Details omitted.
x.register(s);
```

If we next look at the method code for the `Section` class's `register` method to see how it will respond to this message, we see that the `Section` object in turn sends a message to the `Student` to be enrolled, to verify that the `Student` has completed a necessary prerequisite course:

```
public class Section {
  // Details omitted.

  public boolean register(Student s) {
    // Verify that the student has completed a necessary
    // prerequisite course. (We are delegating part
    // of the work to another object, Student s.)
    // Pseudocode.
    boolean completed = s.successfullyCompleted(some prerequisite);
    if (completed) {
      // Pseudocode.
      register the student and return a value of true;
    }
```

```
    else {
      // Pseudocode.
      return a value of false to signal that the registration
        request has been rejected;
    }
  }
}
```

This happens to be an example of ***delegation***, which we discussed in Chapter 4: namely, another object (a `Student`, in this case) helping to fulfill a service request originally made of the `Section` object.

An Object May Return a Value

An object may return a value. The returned value may be one of the following:

- The value of one of the object's attributes
- Some computed value (i.e., a "pseudoattribute," as we discussed in Chapter 4)
- A value that was obtained from some ***other*** object through delegation
- A status code (as in `true`/`false` responses, signaling success or failure of `boolean` methods)

Typical "get" methods fall into this category of event response.

An Object May Interact with the External Boundaries of Its System

An object may interact with the external boundaries of its system; that is, it may display some information on a GUI or cause information to be passed to another application. As you'll learn in Chapters 15 and 16, however, what appears to be an external system boundary is often implemented in Java as yet another object.

An Object May Seemingly Ignore an Event

Finally, an object may seemingly ignore an event, as would be the case if a `Professor` object received the message to add an advisee, but determined that the `Student` whom it was being asked to take on as an advisee was ***already*** an advisee:

```
Student s = new Student();
Professor p = new Professor();
// Details omitted.
// Professor p will seemingly "ignore" this next message.
p.addAdvisee(s);
```

Let's look a slightly different version of the `addAdvisee` method than we saw previously:

```
public class Professor {
  Collection<Student> advisees;  // Holds Student object references.
  // Details omitted.

  public void addAdvisee(Student s) {
    // ONLY insert s into the 'advisees' collection IF IT
    // ISN'T ALREADY IN THERE.
    // Pseudocode.
    if (s is already in collection) return;  // do nothing
    else advisees.add(s);
  }
}
```

Actually, to say that the `Professor` object is doing nothing is an oversimplification: at a minimum, the object is executing the appropriate method code, which is performing some internal state checks ("Is this student already one of my advisees?"). It's just that, when the dust settles, the `Professor` object has neither changed state nor fired off any messages to other objects, so it ***appears*** as if nothing has happened.

Scenarios

Events originating externally to an application occur randomly: we can't predict, for example, when a user is going to click a button on a GUI. In order for an application to perform useful functions, however, the ***internal*** events that arise in ***response*** to these external events—in other words, the messages that objects exchange in carrying out some system function—***can't*** be left to occur randomly. Rather, they must be orchestrated in such a way as to lead, in cause-and-effect fashion, to some desired result. In the same way that a musical score indicates which notes must be played by various instruments to produce a melody, a **scenario** prescribes the sequence of internal messages (events) that must occur in carrying out some system function from beginning to end.

I introduced use cases in Chapter 9 as a way to specify all of the goals for an application from the standpoint of external actors—users or other computer systems. *Merriam-Webster's Collegiate Dictionary, Eleventh Edition*, defines the term ***scenario*** as

A sequence of events esp. when imagined; esp : an account or synopsis of a possible course of action or events.

which is precisely how the term is used in the object modeling sense.

A scenario is one hypothetical instance of how a particular use case might play out. Just as an object is an instance of a class and a link is an instance of an association, ***a scenario may be thought of as an instance of a use case***. Or, stated another way, just as a class is a template for creating objects and an association is a template for creating links, ***a use case is a template for creating scenarios***. A single use case thus inspires many different scenarios, in the same way that planning a driving trip from one city to another can involve many different routes.

We describe scenarios in narrative fashion, as a series of steps observed from the standpoint of a hypothetical observer who is able to see not only what is happening outwardly as the system carries out a particular request but also what is going on behind the scenes, internally to the system. (Note, however, that even though we're now concerned with internal system processes, we're still only interested in ***functional*** requirements as defined in Chapter 9, not in the "bits and bytes" of how the computer works.)

The following is a sample scenario representing the "Register for a course" use case, one of several use cases that we identified for the SRS in Chapter 9.

Scenario #1 for the "Register for a Course" Use Case

In this first scenario, a student by the name of Fred successfully registers for a course. The specific sequence of events is as follows:

1. Fred, a student, logs on to the SRS.
2. He views the schedule of classes for the current semester to determine which section(s) he wishes to register for.
3. Fred requests a seat in a particular section of a course entitled "Beginning Object Concepts," course number OBJ101, section 1.
4. Fred's plan of study is checked to ensure that the requested course is appropriate for his overall degree goals. (We assume that students are not permitted to take courses outside of their plans of study.)
5. His transcript is checked to ensure that he has satisfied all of the prerequisites for the requested course, if there are any.
6. Seating availability in the section is confirmed.
7. The section is added to Fred's current course load.

From Fred's vantage point (sitting in front of a computer screen!), here's what he perceives to be occurring: after logging on to the SRS, he indicates that he wishes to register for OBJ101, section 1, by choosing it from the available course list and then clicks the Add button (see Figure 11-5).

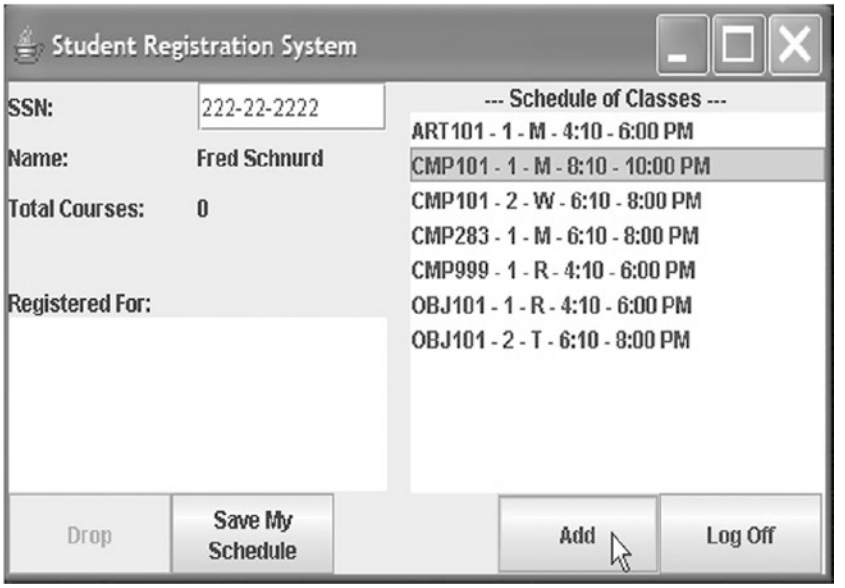


Figure 11-5. *Fred's view of things, part 1*

A few moments later, Fred receives a confirmation message, as shown in Figure 11-6.

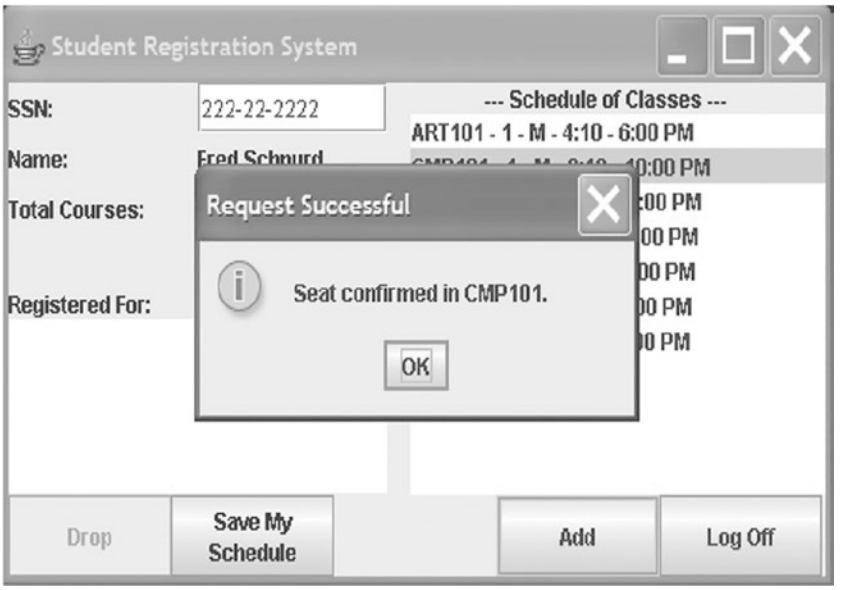


Figure 11-6. *Fred's view of things, part 2*

Fred is unaware (for the most part) of all of the "behind-the-scenes" processing steps that are taking place on his behalf.

The preceding scenario represents a "best-case" scenario, where everything goes smoothly and Fred ends up being successfully registered for the requested course. But, as we know all too well, things don't always work out this smoothly, as evidenced by the following alternative scenario for the ***same*** use case. Everything is the same between Scenario #1 and Scenario #2 except for the steps that are shown in ***bold***.

Scenario #2 for the "Register for a Course" Use Case

In this scenario, Fred once again attempts to register for a course. While he meets all of the requirements, the requested section is unfortunately full. The SRS offers Fred the option of putting his name on a waitlist. The specific sequence of events is as follows:

1. Fred, a student, logs on to the SRS.
2. Fred views the schedule of classes for the current semester to determine which section(s) he wishes to register for.
3. Fred requests a seat in a particular section of a course entitled "Beginning Object Concepts," course number OBJ101, section 1.
4. Fred's plan of study is checked to ensure that the requested course is appropriate for his overall degree goals.
5. His transcript is checked to ensure that he has satisfied all of the prerequisites for the requested course, if any.
6. ***Seating availability in the section is checked, but the section is found to be full.***
7. ***Fred is asked if he wishes to be put on a first-come, first-served waitlist.***
8. ***Fred elects to be placed on the waitlist.***

With a little imagination, you can undoubtedly think of numerous other scenarios for this use case, involving such circumstances as Fred having requested a course that isn't called for by his plan of study or a course for which he hasn't met the prerequisites. And there are many other ***use cases*** to be considered, as well, as were discussed in Chapter 9.

Are there practical limits to the number of alternative scenarios that we should consider for a given use case? As with all requirements analysis, the criteria for when to stop are somewhat subjective. We stop when it appears that we can no longer generate ***significantly different*** scenarios; trivial variations are to be avoided.

When devising scenarios, it's often helpful to observe the future users of the system that we're modeling as they go about performing the same business functions today. In the case of student registration, for example, what manual or automated steps does a student have to go through presently to register for a course? What steps does the university take before deeming a student eligible to register? Whether the registration process is 100 percent manual at present or is based on an automated system that you're going to be replacing or augmenting, observing the steps that are involved today in carrying out a particular business goal can serve as the basis for one or more useful scenarios.

Scenarios, once written, should be added to our project's use case documentation; generally, we pair all scenarios with their associated use cases in that document.

Why are scenarios so important? Because they're the means by which we start to gain insight into the ***behaviors*** that will be required of our objects. We'll need a way to formalize these scenarios so that the actual methods needed for each of our classes become apparent; UML **sequence diagrams** are the means by which we do so, so let's now discuss how to prepare these.

Sequence Diagrams

Sequence diagrams are one of two types of UML **interaction diagrams** (we'll explore the second type, **communication diagrams**, a bit later in this chapter). Sequence diagrams are a way of graphically portraying how messages should flow from one object to another in carrying out a given scenario.

We'll illustrate the process of creating a sequence diagram by creating one for Scenario #1 of the "Register for a course" use case.

Determining Objects and External Actors for Scenario #1

To prepare a sequence diagram, we must first determine

- Which classes of objects (from among those that we specified in our static model [class diagram] in Chapter 10) are involved in carrying out a particular scenario
- Which external actors are involved

Looking back at Scenario #1 for the "Register for a course" use case, we determine that the following objects are involved:

- One `Student` object (representing Fred)
- One `Section` object (representing the course entitled "Beginning Object Concepts," course number OBJ101, section number 1)
- One `PlanOfStudy` object, belonging to Fred
- One `Transcript` object, also belonging to Fred

The scenario also mentions that the student "views the schedule of classes for the current semester to determine which section(s) he wishes to register for." You may recall that when we were determining what our candidate classes should be back in Chapter 10, we debated whether or not to add `ScheduleOfClasses` as a candidate class to our model, and we elected to leave it out at that time. In order to fully represent the details of Scenario #1, we're going to reverse that decision and retrofit `ScheduleOfClasses` into our UML class diagram now as follows:

- We'll show `ScheduleOfClasses` participating in a one-to-many aggregation with the `Section` class because one `ScheduleOfClasses` object will be instantiated per semester to represent all of the sections that are being taught that semester. (It's an abstraction of the paper booklet or online schedule that students look at in choosing which classes they wish to register for in a given semester.)
- We'll also transfer the `semester` attribute from the `Section` class to `ScheduleOfClasses`. Since each `Section` object will now be maintaining a handle on its associated `ScheduleOfClasses` object by virtue of the aggregation relationship between them, a `Section` object will be able to request semester information whenever it is needed.

The results of these changes to our class diagram are highlighted in Figure 11-7.

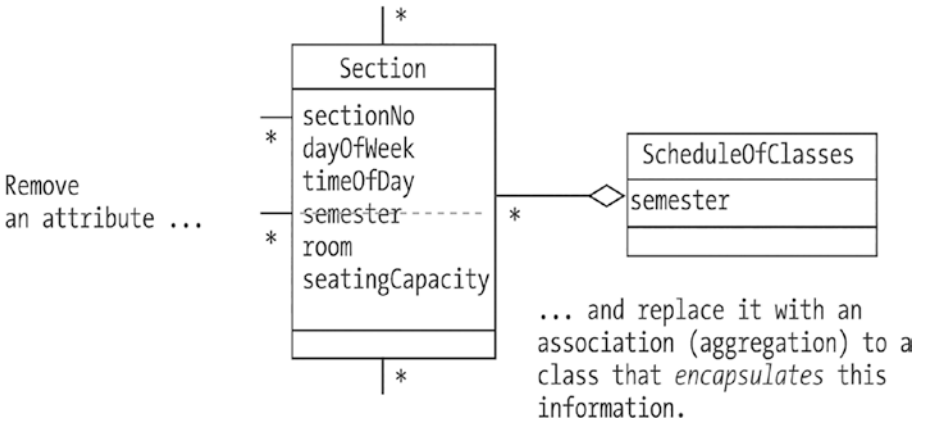


Figure 11-7. *Fine-tuning the UML diagram*

Acknowledging `ScheduleOfClasses` as a class in our model allows us to now reference a `ScheduleOfClasses` object in our sequence diagram, as we'll see in a moment. Scenarios often unearth new classes, attributes, and relationships, thus contributing to our structural "picture" of the system; this is a common occurrence and is a ***desirable*** side effect of dynamic modeling.

Of course, we must also remember to add a definition of `ScheduleOfClasses` to our data dictionary.

ScheduleOfClasses A list of all classes/**sections** that are being offered for a particular semester; **students** review the **schedule of classes** to determine which **sections** they wish to register for.

Finally, since the scenario explicitly mentions interactions between the student user and the system, we'll reflect Fred the ***actor*** separately from Fred the ***object***. Doing so will allow us to represent the SRS interacting externally with the user, as well as showing the system's internal object-to-object interactions. We refer to an object that represents an abstraction of an actor as an instance of a **boundary class**.

Our adjusted list of object/actor participants is now as follows:

- One `Student` object (representing Fred)
- One `Section` object (representing the course entitled "Beginning Objects," course number OBJ101, section number 1)
- One `PlanOfStudy` object, belonging to Fred
- One `Transcript` object, also belonging to Fred
- One `ScheduleOfClasses` object
- One `Student` actor (Fred again!)

Preparing the Sequence Diagram

To prepare a sequence diagram for Scenario #1, we do the following:

- We draw vertical dashed lines, one per object or actor that participates in the scenario; these are referred to as the objects' **lifelines**. Note that the objects/actors can be listed in any order from left to right in a diagram, although it's common practice to place the external user/actor at the far left.
- At the top of each lifeline, as appropriate, we place either an **instance icon**—that is, a box containing the (optional) name and class of an object participant—or a stick figure symbol to designate an actor. (For rules governing how an instance icon is to be formed, please refer back to the section on creating object diagrams in Chapter 10.)
- Then, for each event called out by our scenario, we reflect its corresponding message as a horizontal ***solid-line*** arrow drawn from the lifeline of the sender to the lifeline of the receiver.
- Responses back from messages (in other words, return values from methods or simple `return;` statements in the case of methods declared to have a `void` return type) are shown as horizontal ***dashed-line*** arrows drawn from the lifeline of the **receiver** of the original message ***back to*** the lifeline for the **sender** of the message.
- Message arrows appear in chronological order from top to bottom in the diagram.

The completed sequence diagram for Scenario #1 is shown in Figure 11-8.

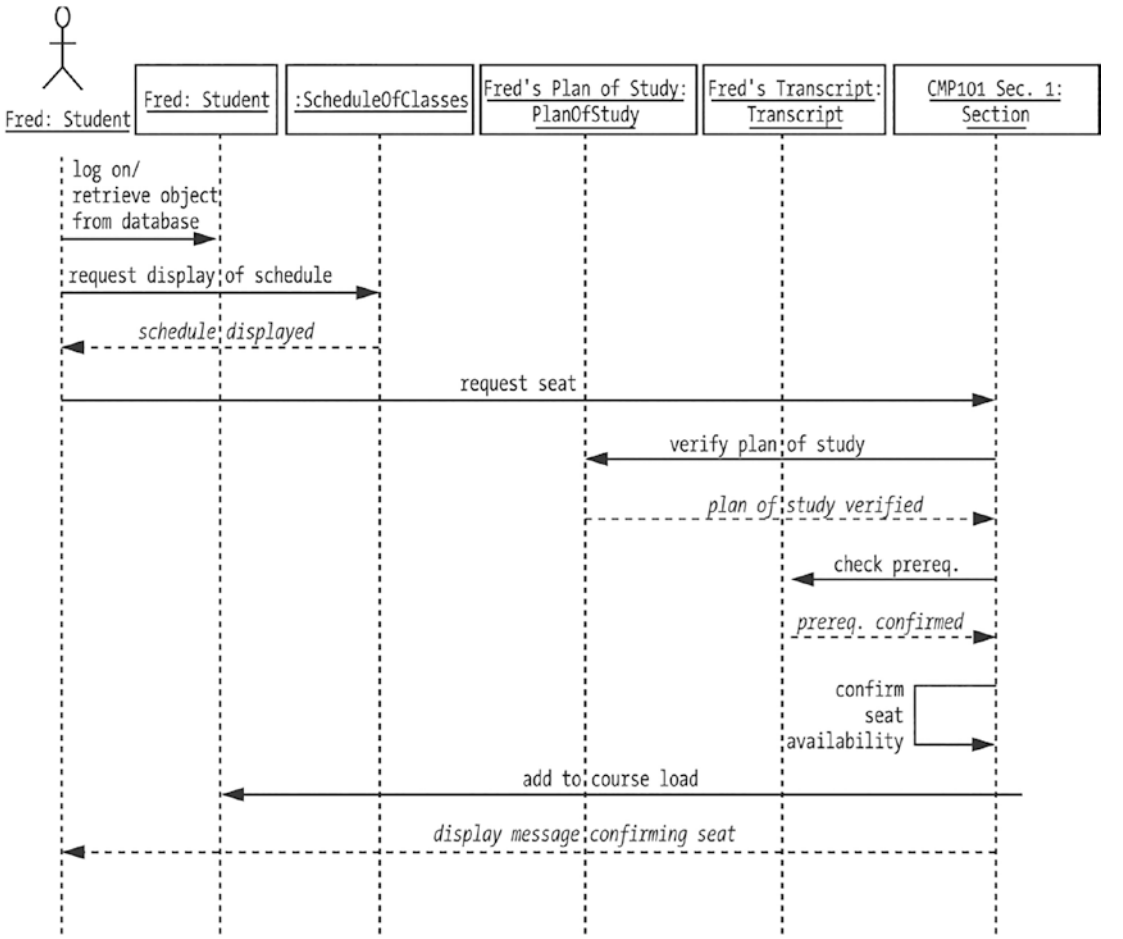


Figure 11-8. *Sequence diagram for Scenario #1*

Let's step through the diagram to make sure you understand all of the activities that are reflected in the diagram:

1. When Fred logs on to the system, his "alter ego" as an object is activated (see Figure 11-9).

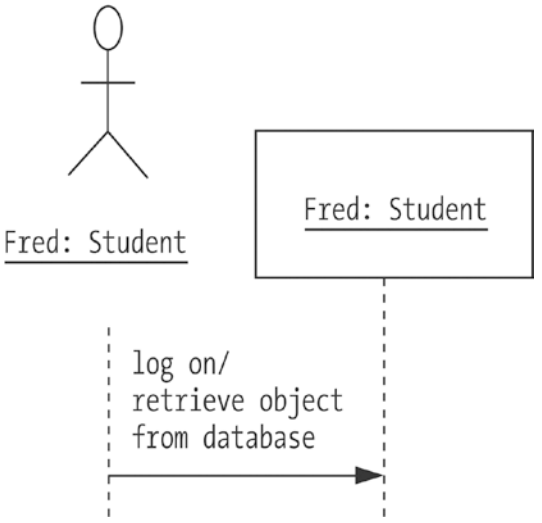


Figure 11-9. *When Fred logs on, a* `Student` *object is instantiated..*

Presumably, information representing each `Student`—in other words, the `Student` object's attribute values—is maintained offline in persistent storage, such as a DBMS or file, until such time as the student logs on, at which time the information is used to instantiate a `Student` object in memory, mirroring the user who has just logged on. We'll talk about reconstituting objects from persistent storage in Chapter 15.

---

2. When Fred the user/actor requests that the semester class schedule be displayed, we reflect the message "request display of schedule" being sent to an anonymous `ScheduleOfClasses` object. The dashed-line-arrow response from the `ScheduleOfClasses` object indicates that the schedule is being displayed to the user, strictly speaking, via a GUI (see Figure 11-10).

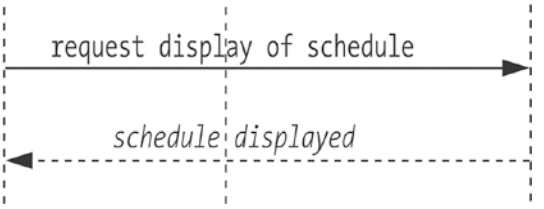


Figure 11-10. *As requested, a schedule of classes is displayed*

I've chosen to label the response arrows with *italic* instead of regular font, a slight departure from official UML notation.

---

3. The next message shown in our diagram is a message from the user to the `Section` object, requesting a seat in the class.

This message is shown originating from the user. In reality, it originates from a GUI component object of the SRS GUI, but we aren't worrying about such implementation details at this stage in the analysis effort.

Note that there is no immediate reply to this message; that's because the `Section` object has a few other objects that it needs to consult with before it can grant a seat to this student, namely

- The `Section` sends a message to the object representing Fred's plan of study, asking that object to confirm that the course that Fred has requested is one of the courses required of Fred in completing his degree program.
- The `Section` next sends a message to the object representing Fred's transcript, asking that object to confirm that a prerequisite course—say COMP 001—has been satisfactorily completed by this student.

4. Assuming that both of these other objects respond favorably, as they are expected to do by virtue of how this scenario was written, the `Section` object then performs some internal processing to verify that there is indeed room for Fred in this section. We reflect internal processing within a single object as an arrow that loops back to the same lifeline that it starts with, as shown in Figure 11-11.

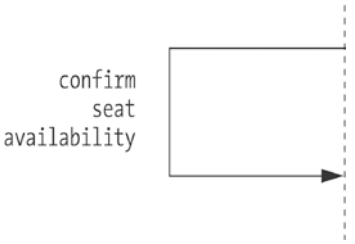


Figure 11-11. *Availability of the requested section is confirmed*

Of course, if we were to reflect ***all*** of the internal processing that is performed by every one of the objects in our sequence diagram, it would be ***flooded*** with such loops! The only reason that we've chosen to show this particular loop is because it's explicitly called out as a step in Scenario #1; if we had omitted it from our diagram, it might appear that we had accidentally overlooked this step.

5. Finally, with all checks having been satisfied, the `Section` object has two remaining responsibilities:

   - First, it sends a new message to the "Fred" `Student` object, requesting that the `Student` object add this `Section` to Fred's course load.
   - Next, the `Section` object sends a response back to Fred the user/actor (via the GUI) confirming his seat in the section. This is the response to the original "request seat" message that was sent by the user toward the beginning of the scenario! All of the extra "behind-the-scenes" processing necessary to fulfill the request—involving a `Section` object collaborating with a `PlanOfStudy` object, a `Transcript` object, and a `Student` object—is transparent

to the user. As we saw earlier in the chapter, Fred merely selected a section from the schedule of classes that was displayed on the SRS GUI, clicked the Add button, and, a few moments later, saw a confirmation message appear on his screen.

Of course, as with all modeling, this particular sequence diagram isn't necessarily the best, or only, way to portray the selected scenario. And, for that matter, we can argue the relative merits of one scenario as compared with another. It's important to keep in mind that preparing sequence diagrams is but a means to an end: namely, discovering the dynamic aspects of the system to be built—that is, the methods—to complement our static/structural knowledge of the system. Recall that our ***ultimate*** goal for Part 2 of the book is to produce an OO blueprint that we can use as the basis for coding the SRS model layer in Part 3. But, as already pointed out, the class diagram that we created in Chapter 10 had a noticeable deficiency: all of its classes' operations compartments were empty. Fortunately, sequence diagrams provide us with the missing pieces of information, as we'll next discuss.

Using Sequence Diagrams to Determine Methods

Now that we've prepared a sequence diagram, how do we put the information that it contains to good use? In particular, how do we "harvest" information from such diagrams concerning the methods that the various classes need to implement?

The process is actually quite simple. We step through the diagram, one lifeline at a time, and study all arrows pointing to that line:

- Arrows representing a new request being made of an object—solid-line arrows—signal methods that the receiving object must be able to perform. For example, we see a solid-line arrow labeled "check prerequisite" pointing to the lifeline representing a `Transcript` object. This tells us that the `Transcript` class needs to define a method that will allow some client object to pass in a particular course object reference and receive back a response indicating whether or not the `Transcript` contains evidence that the course was successfully completed.

- We're free to name our methods in whatever intuitive way makes the most sense, consistent with the method naming conventions discussed in Chapter 4. We're using the method in this particular scenario to check completion of a prerequisite course, so we could declare the method as follows:

  ```
  boolean checkPrerequisite(Course c)
  ```

  But this name is unnecessarily restrictive; what we're ***really*** doing with this method is checking the successful completion of some `Course c`. The fact that it happens to be a prerequisite of some other course is immaterial to how this method will perform. So by naming the method

  ```
  boolean verifyCompletion(Course c)
  ```

  instead, we'll be able to use it anywhere in our application that we need to verify successful completion of a course—for example, when we check whether a student has met all of the course requirements necessary to graduate. (Of course, we could have still used the method in this fashion even if it had been named `checkPrerequisite`, but then our code would be less accurately self-documenting.)

- Arrows representing responses from an operation that some other object has performed—dashed-line arrows—don't get modeled as methods/operations. These do, however, hint at the return type of the method from which this response is being issued. For example, since the response to the "verify plan of study" message is "plan of study verified," this would imply that the method is returning a `boolean` result; hence, we'd declare a method header as follows:

  ```
  boolean verifyPlan(Course c)
  ```

- Loops also represent method calls, performed by an object on itself; these may either represent private "housekeeping" methods or public methods that other client objects may avail themselves of.

Table 11-2 summarizes all of the arrows reflected by the sequence diagram for Scenario #1 from a few pages back.

Table 11-2. *Determining the Methods Implied by Scenario #1*

Arrow Labeled	Drawn Pointing to Class X	A New Request or a Response to a Previous Request?	Method to be Added to Class X
log on	`Student`	Request	(A method to reconstitute this object from persistent storage, such as a file or database; perhaps a special form of constructor—we'll discuss this in Part 3 of the book)
request display of schedule	`ScheduleOfClasses`	Request	`void Display()`
schedule displayed	`Student`	*Response*	*N/A*
request seat	`Section`	Request	`boolean enroll(Student s)`
verify plan of study	`PlanOfStudy`	Request	`boolean verifyPlan (Course c)`
plan of study verified	`Section`	*Response*	*N/A*
check prerequisite	`Transcript`	Request	`boolean verifyCompletion(Course c)`
prerequisite confirmed	`Section`	*Response*	*N/A*
confirm seat availability	`Section`	Request	`boolean confirmSeatAvailability()` (perhaps a private housekeeping method)
add to course load	`Student`	Request	`void addSection(Section s)`
display message confirming seat	*(actor/user)*	*Response*	*N/A (will eventually involve calling upon some method of a user interface object—we'll worry about this in Part 3 of the book)*

Thus we have identified six new "standard" methods plus one constructor that will need to be added to our class diagram, which we'll do shortly.

Repeating this process of sequence diagram production and analysis for various other use case/scenario combinations will flush out most of the methods that we'll need to implement for the SRS. Despite our best efforts, however, a few methods may not surface until we've begun to program our classes—this is to be expected.

Communication Diagrams

The UML notation introduced a second type of interaction diagram, called a **communication diagram**, as an alternative to sequence diagrams. Both types of diagram present essentially the same information, but portray it in a different manner.

In a communication diagram, we eliminate the lifelines used to portray objects and actors. Rather, we lay out instance icons representing objects and stick figures representing actors in whatever configuration is most visually appealing. We then use lines and arrows to represent the flow of messages and responses back and forth between these objects/actors. Because we lose the top-to-bottom chronological sense of message flow that we had with the sequence diagrams, we compensate by numbering the arrows in the order that they would occur during execution of a particular scenario.

The communication diagram in Figure 11-12 is equivalent to the sequence diagram that we produced for Scenario #1.

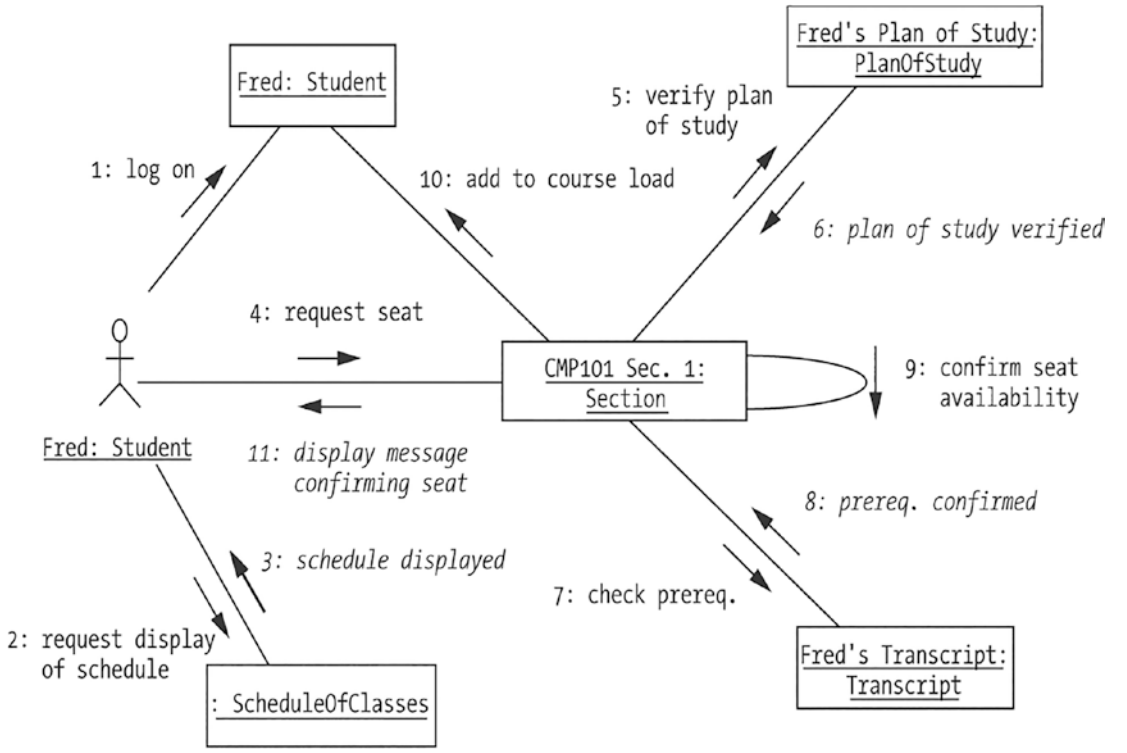


Figure 11-12. *Communication diagram for Scenario #1*

Again, from Fred's vantage point, he observes only a few of these actions, as shown in Figure 11-13.

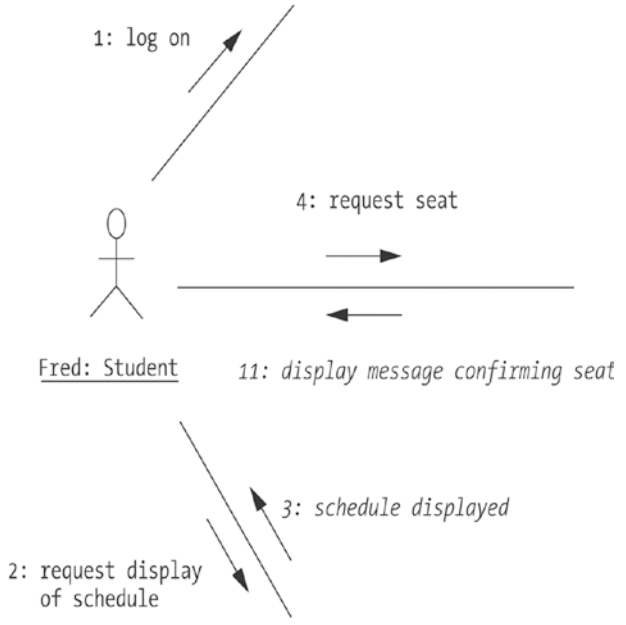


Figure 11-13. *Fred sees only a small subset of the SRS collaborations*

Because sequence and communication diagrams reflect essentially the same information, many object modeling software tools automatically enable us to produce one diagram from the other with the push of a button.

Revised SRS Class Diagram

Going back to the SRS class diagram that we produced in Chapter 10, let's reflect all of the new insights—some behavioral, some structural—that we've gained from analyzing one scenario/sequence diagram (see Figure 11-14).

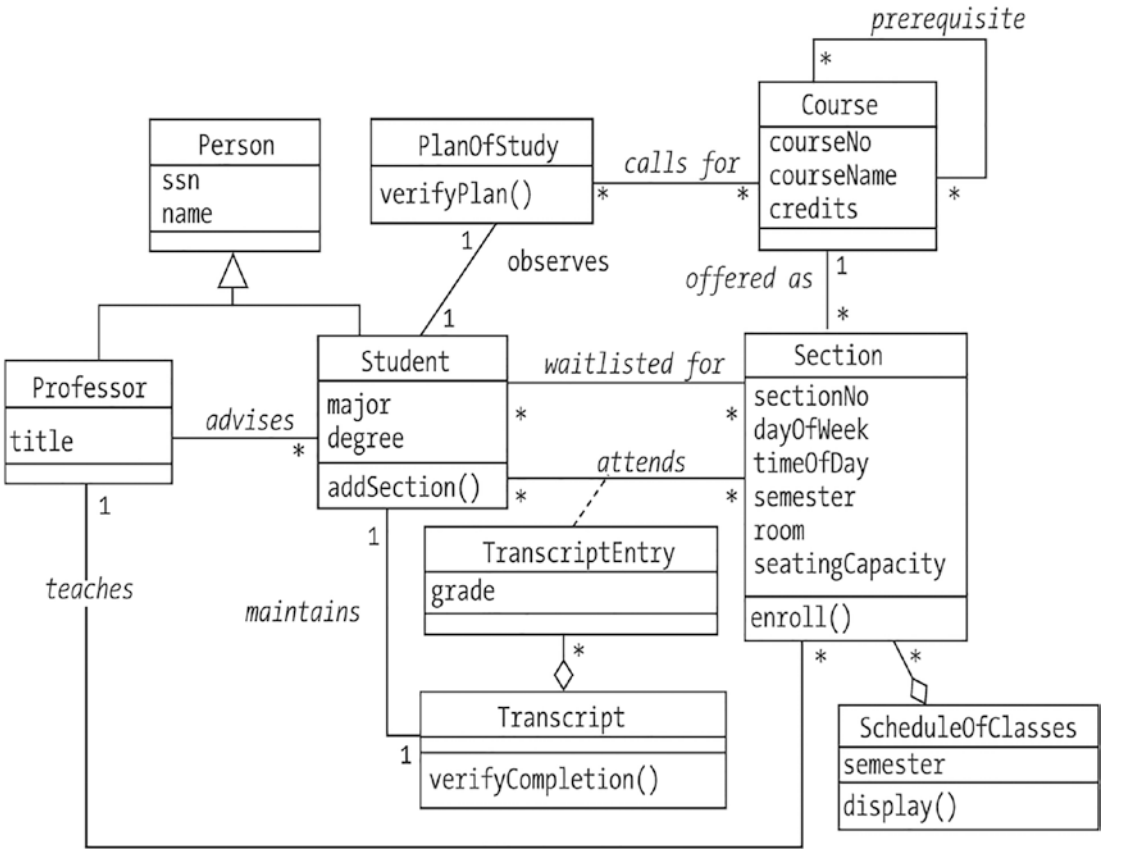


Figure 11-14. *Revised SRS class diagram*

Note that we've decided not to reflect the `confirmSeatAvailability` "housekeeping" method at this time, as we suspect that it will be a private method and don't wish to clutter our diagram. The decision of whether to reflect private methods on a class diagram—or, for that matter, to reflect ***any*** feature of a class—is up to the modeler, because again, the purpose of the diagram is to communicate, and too much detail can actually lessen a diagram's effectiveness in this regard.

We must remember to update the SRS data dictionary any time we add classes, attributes, relationships, or methods to our model. Here's a suggested format for how we might wish to describe a method in the dictionary:

Method: `enroll`

Defined for class: `Section`

Header: `boolean enroll(Student s)`

Description: This method enrolls the designated person in the section, unless (a) the section is already full, (b) the student's plan of study doesn't call for this course, or (c) the student hasn't met the prerequisites. It returns a `boolean` value to indicate success (`true`) or failure (`false`) of the enrollment.

Summary

In this chapter, you've seen how the process of dynamic modeling is a complementary technique to the static modeling that enriches our overall understanding of the problem to be automated, hence enabling us to improve our object "blueprint," also known as a class diagram. In particular, you've seen

- How events trigger state changes
- How to develop scenarios, based on use cases
- How to represent these as UML interaction diagrams: sequence diagrams or, alternatively, communication diagrams
- How to glean information from sequence diagrams concerning the behaviors expected of objects—that is, the methods that our classes will need to implement—so as to round out our class diagram
- How sequence diagrams can also yield additional knowledge about the structural aspects of a system

EXERCISES

1. Prepare a sequence diagram for Scenario #2 as presented earlier in this chapter.
2. Prepare a sequence diagram to represent the following scenario for the SRS case study:
   a. Mary, a student, logs on to the SRS.
   b. She indicates that she wishes to drop ART 222, section 1.

c. ART 222, section 1, is removed from Mary's course load.

d. The system determines that Joe, another student, is waitlisted for this section.

e. The section is added to Joe's current course load.

f. An email is sent to Joe notifying him that ART 222 has been added to his course load.

3. Provide a list of all of the method headers that you would add to each of your classes based on the sequence diagram that you prepared for Exercise 2. Also, note any new classes, attributes, or relationships that would be needed.

4. Prepare a second sequence diagram for the SRS case study, representing a scenario of your own choosing based upon any of the SRS use cases identified in Chapter 9. This scenario should be significantly different from those presented in this chapter and from the scenario in Exercise 2. You must also narrate the scenario as was done for Exercise 2.

5. Provide a list of all of the method headers that you would add to each of your classes based on the sequence diagram that you prepared for Exercise 4. Also, note any new classes, attributes, or relationships that would be needed.

6. Prepare a sequence diagram to represent the following scenario for the Prescription Tracking System (PTS) case study presented in the Appendix:

   a. Mary Jones, an existing customer of the pharmacy, brings in a prescription for eye drops to have it filled.

   b. The pharmacist checks to see if Ms. Jones has previously had a prescription filled for this item.

   c. The pharmacist discovers that she has and furthermore that the last time it was refilled was less than a month ago.

   d. Knowing that her insurance won't authorize payment for this same prescription so soon, the pharmacist informs Ms. Jones, and she decides to wait to have it filled at a later date.

7. Devise an "interesting" scenario, and prepare the corresponding sequence diagram, for the problem area whose requirements you defined for Exercise 3 in Chapter 2.

8. Provide a list of all of the method headers that you would add to each of your classes based on the sequence diagram that you prepared for Exercise 7. Also, note any new classes, attributes, or relationships that would be needed.

CHAPTER 12

Wrapping Up Our Modeling Efforts

Having used the techniques for static and dynamic modeling presented in Chapters 10 and 11, respectively, we've arrived at a fairly thorough object model of the SRS—or so it seems! Before we embark upon implementing our class diagram as Java code in Part 3 of the book, however, we need to make sure that our model is as accurate and representative of the goal system as possible.

In this chapter, we'll

- Explore some simple techniques for testing our model.
- Talk about the notion of reusing models.

Testing the Model

Testing a model doesn't involve "rocket science"; rather, it calls for some commonsense measures designed to identify errors and/or omissions.

- First of all, revisit all requirements-related project documentation—the original problem statement and the supporting use cases—to ensure that no requirements were overlooked. We'll do so for our SRS model in a moment.
- Conduct a minimum of two separate formal walk-throughs of the model: one with the development team members and the other with the future users of the system. Prior to each walk-through, make sure to distribute copies of the following documentation to each of the

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_12

participants far enough in advance to allow them adequate time to review these, if they so desire (but be prepared to discuss significant aspects of these at the meeting in case the participants haven't reviewed them):

- Executive summary version of the problem statement narrative
- Class diagram
- Data dictionary
- Use case documentation
- Significant scenarios and corresponding message trace diagrams

By this stage in the project, you'll have hopefully already educated your users on how to read UML diagrams, and they'll have informally seen numerous iterations of the evolving models. If any of the participants in the upcoming walk-throughs aren't familiar with any of the notation, however, take time in advance to tutor them in this regard. (The information contained in Chapters 10 and 11 of this book should be more than adequate as the basis for such a tutorial.)

When conducting the walk-through, designate someone to be the narrator and discussion leader and a different person to be responsible for recording significant discussion content, particularly changes that need to be made. Having one person trying to do both is too distracting, and important notes may be missed as a result. If appropriate, you may even arrange to tape-record the discussion.

Remain open-minded throughout the review process. It's human nature to want to defend something that we've worked hard on putting together, but remember that it's far better to find and correct shortcomings now, when the SRS is still a paper skeleton, than after it has been rendered into code.

Revisiting Requirements

In revisiting the SRS case study problem statement, we find that we've indeed ***missed*** one requirement, namely

The SRS will verify whether or not the proposed plan of study satisfies the requirements of the degree that the student is seeking.

We didn't model `Degree` as a class—recall that we debated whether or not to do so back in Chapter 10 and ultimately decided against it. Nor, for that matter, do we reflect the requirements of a particular degree program in our model. Let's look at what it would take to do so properly at this time.

Researching the way in which our university specifies degree program requirements, we learn the following:

- Every degree program specifies five "core" courses—that is, courses that a student ***must*** take. For example, for the degree of Master of Science in information technology (MSIT), students are required to complete the following five core courses:
  - Analysis of Algorithms
  - Application Programming Design
  - Computer Systems Architecture
  - Data Structures
  - Information Systems Project Management
- Students are expected to select an area of specialization, known as a **concentration**, within their degree program. For the MSIT degree, our university offers three different concentrations:
  - Object Technology
  - Database Management Systems
  - Networking and Communications
- Each concentration in turn specifies three mandatory, concentration-specific courses. For the MSIT degree with a concentration in Object Technology, the required concentration-specific courses are
  - Object Methods for Software Development
  - Advanced Java Programming
  - Object Database Management Systems
- Finally, the student must take two additional electives to bring their course total to ten.

Phew! To model all of these interdependencies would require a fairly complex class diagram structure, as shown in Figure 12-1.

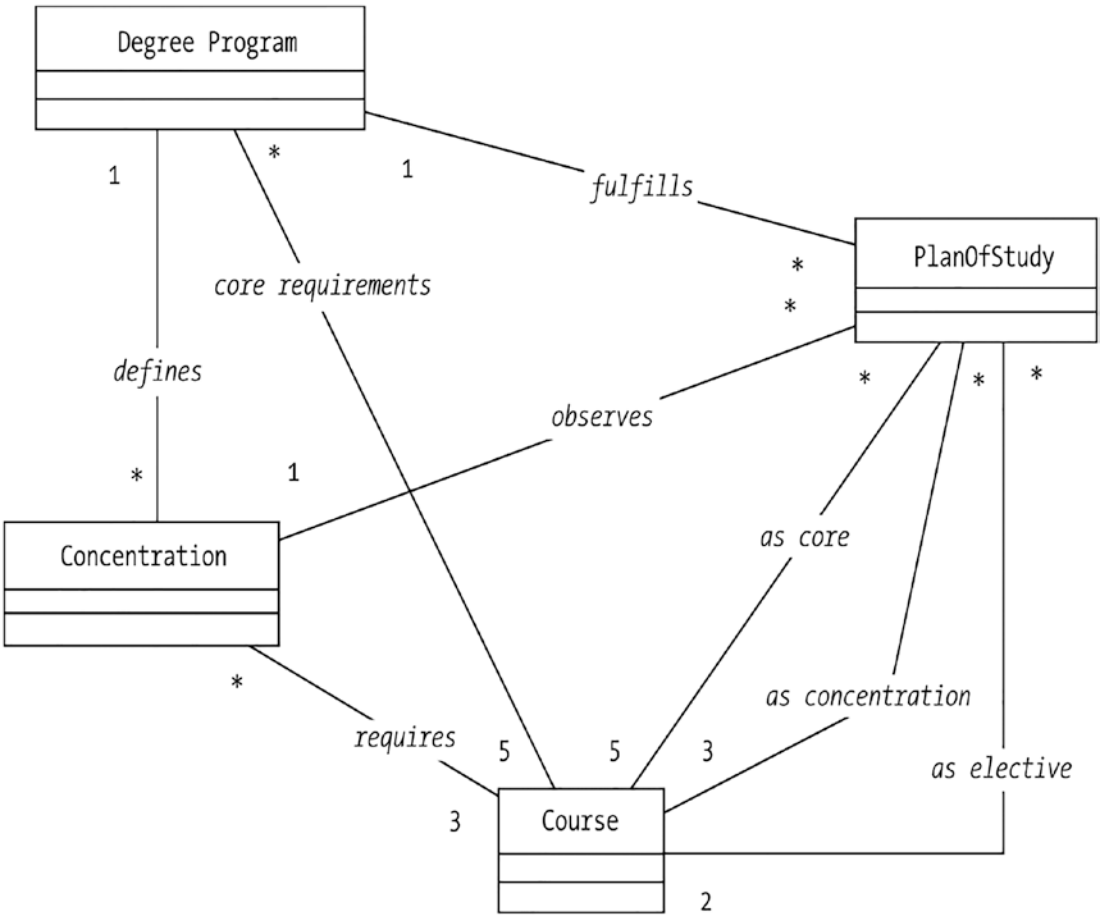


Figure 12-1. *Modeling degree program requirements proves to be rather complicated*

We go back to our project sponsors—the future users of the SRS—and break the news to them that we've just uncovered a previously missed requirement that is going to significantly increase the complexity and cost of our automation effort. The sponsors decide that having the SRS verify the correctness of a student's plan of study is too ambitious a goal; they instead decide that a student will use the SRS to submit a ***proposed*** plan of study, but that their advisor will then be responsible for ***subsequently*** verifying and approving it. So all we wind up having to do to correct our SRS class

diagram as last presented is to add one attribute to the `PlanOfStudy` class, reflecting the date on which it was approved, and a new *approves* association connecting the `Professor` class to the `PlanOfStudy` class, and we're good to go!

Note that we don't need to add an `approvePlan` method to the `PlanOfStudy` class, because as discussed in Chapter 10 we may assume the presence of "set" methods for all attributes; the `setDateApproved` method would suffice for marking a plan as approved. And the *approves* association between the `PlanOfStudy` and `Professor` classes (see the diagram excerpt in Figure 12-2) ensures us that each `PlanOfStudy` object will maintain a handle on the `Professor` object who actually approved the plan on the date indicated.

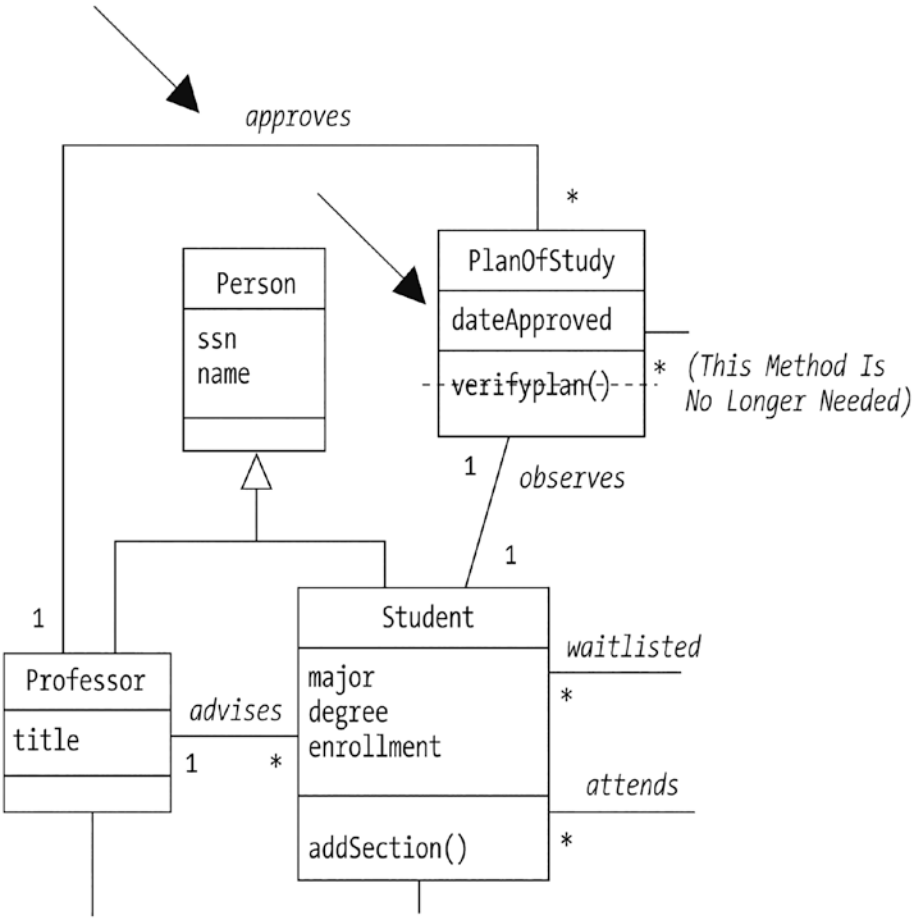


Figure 12-2. *Making minor adjustments to the SRS class diagram*

Reusing Models: A Word About Design Patterns

As we discussed in Chapter 2, when learning about something new, we automatically search our "mental archive" for other abstractions/models that we've previously built and mastered, to look for similarities that we can build upon. This technique of comparing features to find an abstraction that is similar enough to be reused effectively is known as **pattern reuse**. As it turns out, pattern reuse is an important technique for object-oriented software development.

Let's say that after we finish up our SRS class diagram, we're called upon to model a system for a small travel agency, Wild Blue Yonder (WBY). As a brand-new travel agency, WBY wishes to offer a level of customer service above and beyond their well-established competitors, and so they decide to enable their customers to make travel reservations online via the Web (most of WBY's competitors take such requests over the phone).

For any given travel package—let's say a ten-day trip to Ireland—WBY offers numerous trips throughout the year. Each trip has a maximum client capacity, so if a client can't get a confirmed seat for one of the trips, they may request a position on a first-come, first-served waitlist.

In order to keep track of each client's overall experience with WBY, the travel agency plans on following up with each client after a trip to conduct a satisfaction survey and will ask the client to rate their experience for that trip on a scale of 1 to 10, with 10 being outstanding. By doing so, WBY can determine which trips are the most successful, so as to offer them more frequently in the future, as well as perhaps eliminating those that are less popular. WBY will also be able to make more informed recommendations for future trips that a given client is likely to enjoy by studying that client's travel satisfaction history.

In reflecting on the requirements for this system, we experience déjà vu! We recognize that many aspects of the WBY system requirements are similar to those of the SRS. In fact, we're able to reuse the overall structure, or ***pattern***, of the SRS object model by making the following class substitutions:

- Substitute `TravelPackage` for `Course`.
- Substitute `Trip` for `Section`.
- Substitute `Client` for `Student`.
- Substitute `TripRecord` for `TranscriptEntry`.
- Substitute `TravelHistory` for `Transcript`.

Note that all of the relationships among these classes—their names, types, and even their multiplicities—remain unchanged from the SRS class diagram (see Figure 12-3).

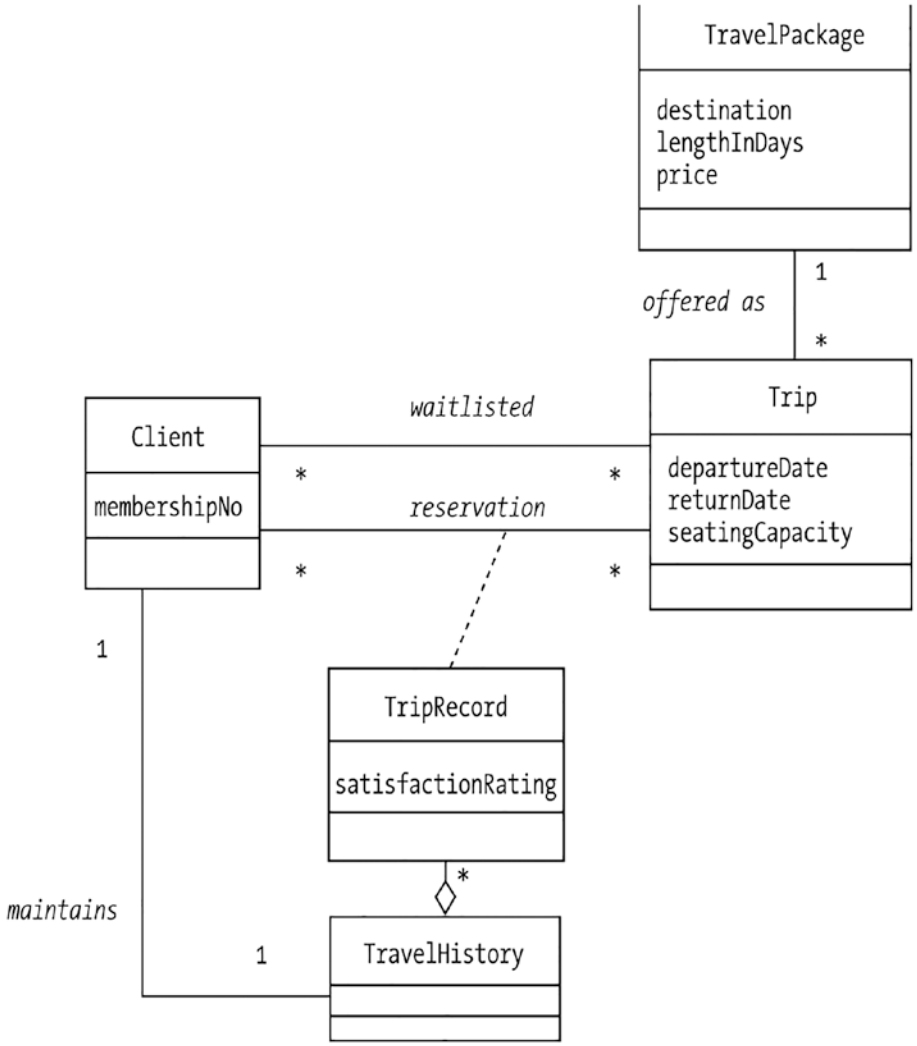


Figure 12-3. *Reusing the SRS design pattern for WBY*

Such an exact match is exceptionally rare when reusing design patterns; don't be afraid to change some things (eliminate classes or associations, change multiplicities, etc.) in order to facilitate reuse of a similar, but not identical, pattern.

Having recognized the similarities between these two designs, we're poised to take advantage of quite a bit of reuse with regard to the code of these two systems, as well. In fact, had we anticipated the need for developing these two systems prior to developing either one, we could have taken steps up front to develop a ***generic*** pattern that could have been used as the basis for both systems, as well as any future reservation systems we might be called upon to model, as illustrated in Figure 12-4.

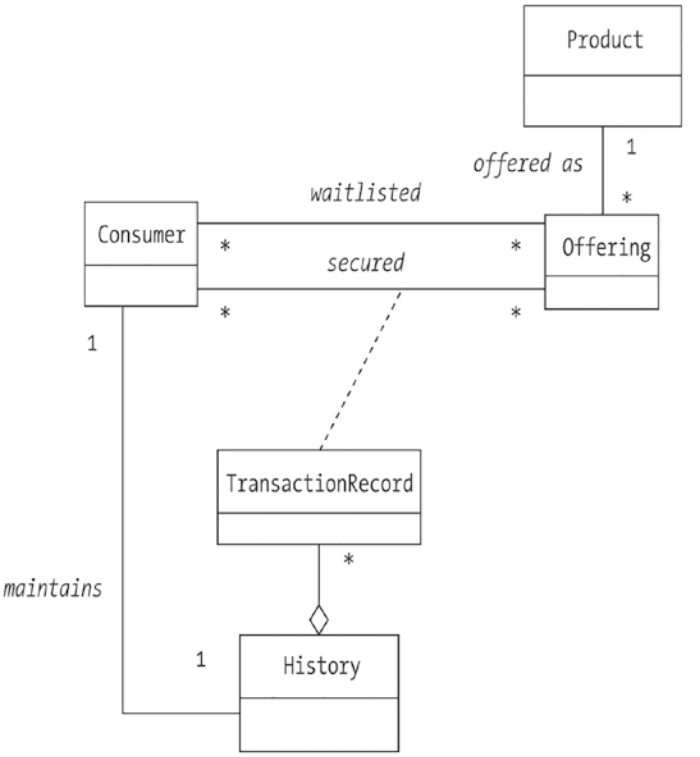


Figure 12-4. *A general-purpose class diagram for reservation systems*

Many useful, reusable patterns have been studied and documented; before embarking on a new object modeling project, it's worth exploring whether any of these may be a suitable starting point.

Summary

Learning to model a problem from the perspective of objects is a bit like learning to ride a bicycle. You can read all the books ever published on the subject of successful bicycle riding, but until you actually sit on the seat, grab the handlebars, and start pedaling, you won't get a real sense of what it means to ride. You'll probably wobble at first, but with a bit of a boost from training wheels or a friendly hand to steady you, you'll be riding off on your own with time. The same is true of object modeling: with practice, you'll get an intuitive feel for what makes a good candidate class, a useful scenario, and so on.

In this chapter, we

- Discussed techniques for verifying the accuracy and completeness of a class diagram
- Looked at how object models can be reused/adapted to other problems with similar requirements

EXERCISES

1. Conduct a walk-through of one of the class diagrams that you prepared as an exercise for Chapter 10—either the Prescription Tracking System (PTS) case study presented in the Appendix or the problem area whose requirements you defined for Exercise 3 in Chapter 1—with a classmate or coworker. Report on any insights that you gained as a result of doing so.
2. Think of two other problem areas where the reservation pattern that we identified for the Wild Blue Yonder travel agency might also apply. What adjustments, if any, would you need to make to the reservation pattern in order to use it in those situations?

CHAPTER 13

A Few More Java Details

You received a solid introduction to Java syntax in Part 1 of this book, particularly as it pertains to illustrating fundamental object concepts. Before we dive into the specifics of coding the model layer of the Student Registration System (SRS), however, we'd like to introduce you to a few more details about the Java language that we'll subsequently take advantage of when coding the SRS model layer.

In this chapter, you'll learn about

- The nature and purpose of Java archive ("jar") files
- The mechanics of Java documentation comments
- The object nature of `Strings`
- A special type of class called an `enum` that can be used to enumerate the explicit values that a particular variable is allowed to assume
- How we can form highly complex expressions by chaining messages
- How objects refer to themselves from within their own methods
- The nature of Java run-time exceptions and how to gracefully handle them, including defining and using our own custom exception types
- Several approaches for providing input to command-line-driven "GUI-less" programs
- Some important features of the Java `Object` class

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_13

Java-Specific Terminology

As mentioned previously, the purpose of this book is to educate you on object principles that are, for the most part, language neutral, and so I've favored generic (and sometimes informal) OO terminology over Java-specific terminology as used by Oracle. That being said, I'd now like to expose you to ***Java-specific terminology*** for a number of basic object concepts as used in the formal Java Language Specification (JLS) maintained by Oracle Corporation.

Table 13-1. *Comparing Generic OO vs. Java Terminology*

Generic OO Terminology Used in This Book	Formal Java-Specific Terminology as Used by Sun Microsystems	Used to Describe the Following Notion
Attribute	field, instance variable	A variable that is created once per object, that is, per each instance of a class. Each object has its own separate set of instance variables.
static variable (informal: static attribute)	static field, class variable	A variable that exists only once per class.
Method	instance method	A function that is invoked on an object.
static method	class method	A function that can be called on a class as a whole, without reference to a specific object. Class methods can neither call instance methods nor access instance variables.
Feature	member	Those components of a class that can potentially be inherited: for example, instance/class variables and instance/class methods, but ***not*** constructors.

To round out this terminology, the term **local variable** refers to a variable that is declared inside of a method and hence is locally scoped relative to that method. (Method parameters are also local to the method, but Java distinguishes them from "local variables"). Local variables are neither static nor instance variables.

Here is a code snippet that illustrates all three types of variable:

```
public class Student {
  // Attributes.
  private String name;  // <== name is an instance variable
  private static int totalStudents;  // <== totalStudents is a
                                                  //  class variable

  // Methods.
  public void foo(int y) { // <== parameter y is local to the method
    int x;  // <== x is a local variable
    // etc.
  }

  // etc.
}
```

Java Archive (jar) Files

The Java bytecode comprising an application is commonly bundled and delivered in the form of a **Java archive ("jar") file**. Let's use a simple application as an example, consisting of

- Three user-defined types–classes `Person`, `Student`, and `Professor`
- A `main` method wrapper class called `MyApp`

to illustrate the use of jar files. The code for our simple example is shown in the following

```
public class Person {
    private String name;

    public void setName(String n) {
        name = n;
    }
}

public class Student extends Person {
    private Professor advisor;
```

```
    public void setAdvisor(Professor p) {
        advisor = p;
    }
}

public class Professor extends Person {
    private String title;

    public void setTitle(String t) {
        title = t;
    }
}
```

to support the following program:

```
public class MyApp {
    public static void main(String[] args) {
        Professor p = new Professor();
        Student s = new Student();
        s.setAdvisor(p);
    }
}
```

Creating Jar Files

We start by compiling our code:

```
javac *.java
```

Then, to create a jar file, we type

```
jar cvf jarfilename.jar list_of_files_to_be_included_in_jar_file
```

where the command-line argument `cvf` indicates that

- We wish to c***reate*** a jar file.
- We wish the command to be v***erbose***–that is, we want the command to display everything that is going on as the jar file is created.
- We are designating the (f***ile)name*** of the jar file that is to be created.

For example, to place our simple application's bytecode into a jar file named `MyJar.jar`, we'd type

```
jar cvf MyJar.jar Person.class Student.class Professor.class MyApp.class
```

(Note that the bytecode files can be listed in any order). Alternatively, we could use a wildcard character to include multiple files at once:

```
jar cvf MyJar.jar *.class
```

Because of our use of the v(erbose) command option, the following output would be displayed by the `jar` utility:

```
added manifest
adding: Person.class(in = 222) (out= 178)(deflated 19%)
adding: Student.class(in = 419) (out= 287)(deflated 31%)
adding: Professor.class(in = 219) (out= 176)(deflated 19%)
adding: MyApp.class(in = 1751) (out= 1021)(deflated 41%)
```

Note that we can include any type of file that we wish in a jar file, not only bytecode files. For example, if we want to archive our source code along with our bytecode, we can type the command

```
jar cvf MyJar.jar *.class *.java
```

A jar file is a ZIP file in disguise. While we'll create, inspect, and (occasionally) extract individual bytecode files from a jar file via the command-line `jar` utility that comes with the Java Development Kit (JDK), many standard ***ZIP*** utilities are also able to read/extract files from jar files. Thus, as with ZIP files, we can store literally any file type within a jar: source code, bytecode, image files, even other jar files.

Inspecting the Contents of a Jar File

To inspect/list the contents of a jar file without "unjarring" (extracting) files, we use the command

```
jar tvf jarfilename.jar
```

where the `t` command-line argument indicates that we wish to see a t***able of contents*** for the named jar file, for example:

```
jar tvf MyJar.jar
```

Output:

```
     0 Sun Feb 20 13:55:34 EST 2005 META-INF/
    71 Sun Feb 20 13:55:34 EST 2005 META-INF/MANIFEST.MF
   222 Mon Feb 07 16:07:16 EST 2005 Person.class
   419 Fri Feb 18 10:06:02 EST 2005 Student.class
   219 Mon Feb 07 16:07:16 EST 2005 Professor.class
  1751 Wed Feb 06 07:36:44 EST 2002 MyApp.class
```

Once our jar file is created, it is a simple matter to share that file with a user or another developer, perhaps by sending it to them as an email attachment or by storing it in a shared location on a network file system.

Using the Bytecode Contained Within a Jar File

To inform the JVM that we want to ***use*** the bytecode within a jar file, we use a command-line option to set an environment variable called the **classpath** as follows:

```
java -cp path_to_jar_file class_containing_main_method
```

For example, if we store our `MyJar.jar` file in a shared directory by the name of `S:\applications`, then to execute the `MyApp` program that is stored ***within*** that jar file, we would type

```
java -cp S:\applications\MyJar.jar MyApp
```

If more than one jar file is to be referenced at the same time, the references are separated by semicolons under DOS and colons under UNIX, for example, under DOS:

```
java -cp S:\applications\MyJar.jar;T:\stuff\AnotherJar.jar SomeApp
```

Extracting Contents from Jar Files

Note that we need ***not*** extract bytecode from a jar file in order to use it; the JVM is able to retrieve individual bytecode files from ***within*** jar files as needed. However, should we ever wish to extract selected files from a jar file–say that Java source files were included and we'd like to work with individual source files–we'd type the command

`jar xvf` *`jarfilename`*`.jar` *`space_separated_list_of_files_to_be_extracted`*

For example:

```
jar xvf MyApp.jar Student.java Professor.java
```

where the `x` command option indicates that we wish to ***extract*** files; to extract ***all*** source code in a jar file, we'd type

```
jar xvf MyApp.jar *.java
```

And to extract ***everything*** from a jar file, we'd type

```
jar xvf MyApp.jar
```

Note that if you extract the contents of a jar file into a directory that contains files by the same names as those you are extracting, the extracted files will automatically ***overwrite*** similarly named files; you will ***not*** be warned beforehand! So it's a good idea to make a backup copy of the files in a directory before "unjarring" a jar file there or, alternatively, to always "unjar" a file in a separate empty directory.

"Jarring" Entire Directory Hierarchies

It's possible to incorporate the contents of an entire directory hierarchy (all subfolders) into a single jar file via the command:

`jar cvf` *`jarFileName topLevelDirectoryName`*

For example, the SRS code examples that accompany this book are stored within a hierarchy of directories on my computer under a parent directory named `C:\My Documents\BJO Second Edition\Code`. To create a jar file containing all of the code, I'd type the command

```
jar cvf BJOcode.jar "C:\My Documents\BJO Second Edition\Code"
```

(note that we use double quotes to surround a path if it contains blank spaces) or, alternatively,

```
cd "C:\My Documents\BJO Second Edition"
jar cvf BJOcode.jar Code
```

The resultant output, excerpted in the following, illustrates how all of the subdirectories are traversed so as to include their contents in the jar file:

```
added manifest
adding: Code/(in = 0) (out= 0)(stored 0%)
adding: Code/Ch14/(in = 0) (out= 0)(stored 0%)
adding: Code/Ch14/SRS/(in = 0) (out= 0)(stored 0%)
adding: Code/Ch14/SRS/Course.java(in = 2784) (out= 953)(deflated 65%)
adding: Code/Ch14/SRS/EnrollmentStatus.java(in = 872) (out= 438)
(deflated 49%)
adding: Code/Ch14/SRS/Person.java(in = 1223) (out= 513)(deflated 58%)
adding: Code/Ch14/SRS/Professor.java(in = 2967) (out= 1122)(deflated 62%)
etc.
adding: Code/Ch15/(in = 0) (out= 0)(stored 0%)
adding: Code/Ch15/SRS/(in = 0) (out= 0)(stored 0%)
adding: Code/Ch15/SRS/Course.java(in = 2784) (out= 953)(deflated 65%)
adding: Code/Ch15/SRS/CourseCatalog.java(in = 1640) (out= 713)
(deflated 56%)
etc.
adding: Code/Ch16/(in = 0) (out= 0)(stored 0%)
adding: Code/Ch16/BeanExample.java(in = 668) (out= 378)(deflated 43%)
adding: Code/Ch16/BorderLayoutLayout.java(in = 1272) (out= 470)
(deflated 63%)
```

```
adding: Code/Ch16/BorderLayoutLayout2.java(in = 1279) (out= 476)
(deflated 62%)
adding: Code/Ch16/Calculator1.java(in = 2449) (out= 966)(deflated 60%)
adding: Code/Ch16/Calculator2.java(in = 2462) (out= 921)(deflated 62%)
adding: Code/Ch16/Calculator3.java(in = 3514) (out= 1229)(deflated 65%)
etc.
adding: Code/Ch16/SRS/(in = 0) (out= 0)(stored 0%)
adding: Code/Ch16/SRS/Course.java(in = 2784) (out= 953)(deflated 65%)
adding: Code/Ch16/SRS/CourseCatalog.java(in = 1640) (out= 713)
(deflated 56%)
etc.
```

Javadoc Comments

In Chapter 2, we briefly mentioned the notion of ***Java documentation comments*** (a.k.a. "***javadoc*" *comments***), a special type of comment from which we can automatically generate HTML documentation for an application. Let's explore how this is accomplished.

Java documentation comments, like traditional comments, can span multiple lines of code. However, javadoc comments start with a slash followed by ***two*** asterisks /** and end with an asterisk slash */. Within the body of a javadoc comment, we are able to use a number of predefined **javadoc tags** (whose names start with "@") to control how the resultant HTML will look.

Here's a simple `Person` class that incorporates javadoc comments (**bolded**):

```
// Person.java

/**
 * A person is a human being. We might use a Person to represent a student
 * or a professor in an academic setting.
 */
public class Person {
    //------------
    // Attributes.
```

```
    //------------

    /**
     * A person's legal name. Typically represented as
     * "FirstName I. LastName".
     */
    public String name;

    /**
     * A person's age in years. No matter how imminent a person's next
     birthday
     * is, their age will always reflect how old they were at their most
     * recent birthday.
     */
    private int age;

    //-------------
    // Constructor.
    //-------------
    /**
     * This constructor initializes attributes name and age.
     * @param n the Person's name, in first name - middle initial -
     * last name order.
     * @param a the Person's age.
     */
    public Person(String n, int a) {
        name = n;
        age = a;
    }

    /**
     * This method is used to determine a person's age in dog years.
     */
    public double dogYears() {
        return age/7.0;
    }
}
```

Here are some observations about the preceding example:

- `public` features automatically appear in javadoc-generated documentation; `private` features, by default, do not. Thus, despite the fact that we've documented the `private age` attribute in javadoc style, age will not be reflected in the resultant HTML documentation, as we'll see in a moment.
- `@param` is a javadoc-specific tag used to define the purpose of a particular parameter to a method; the general syntax for its use is `@param` *parameterName description*.
- Intervening blank lines and/or non-javadoc comments, if present, are ignored by the `javadoc` utility:

```
/**
 * A person's legal name. Typically represented as "FirstName
 I. LastName".
 */
// Having a non-Javadoc comment here won't hurt, nor will intervening
// blank lines.

public String name;
```

To generate HTML documentation for this class, we use the command-line `javadoc` utility that comes standard with the Java Development Kit (JDK). We can either type the command

```
javadoc Person.java
```

to generate documentation for a single class, or we may type

```
javadoc *.java
```

if documentation is to be generated for more than one `.java` file at the same time.

A number of files are automatically generated as a result of typing a `javadoc` command, as illustrated by the output shown in the following:

```
C:\> javadoc Person.java
Loading source file Person.java...
Constructing Javadoc information...
```

```
Standard Doclet version 1.5.0-beta2
Building tree for all the packages and classes...
Generating Person.html...
Generating package-frame.html...
Generating package-summary.html...
Generating package-tree.html...
Generating constant-values.html...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating allclasses-noframe.html...
Generating index.html...
Generating help-doc.html...
Generating stylesheet.css...
```

To view the resultant documentation, we use a web browser to load the `index.html` file, which will bring up the "home page" for our documentation, as shown in Figure 13-1. (Note that the exact layout of this page may change from one Java version to another.)

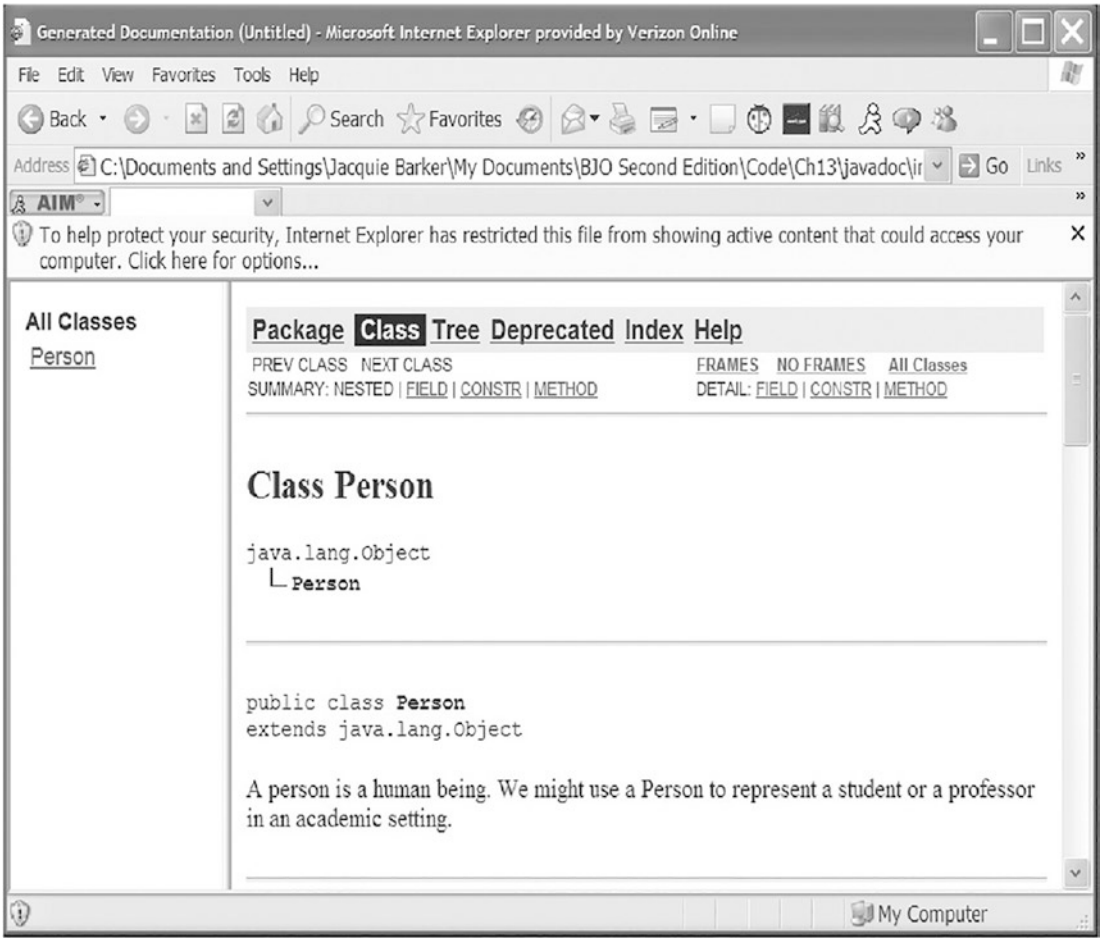


Figure 13-1. *Viewing the* `index.html` *page for our* `Person` *class*

Let's explore this page:

- At the top of the page, we see the inheritance hierarchy to which the `Person` class belongs (in our case, `Person` is shown as being directly derived from the `Object` class of the `java.lang` package).
- Next, we see the narrative description of our class from the javadoc comment that preceded the `public class Person { ...` declaration.

Scrolling down a bit further on the page as illustrated in Figure 13-2, we see lists of all `public` attributes, constructors, and methods belonging to this class under the headings **Field Summary**, **Constructor Summary**, and **Method Summary**, respectively. Recall

that since age was declared to be a `private` attribute, it is omitted by default. To include ***all*** features in the generated documentation whether `public` or not, simply include the `-private` flag on the `javadoc` command, for example:

```
javadoc -private Person.java
```

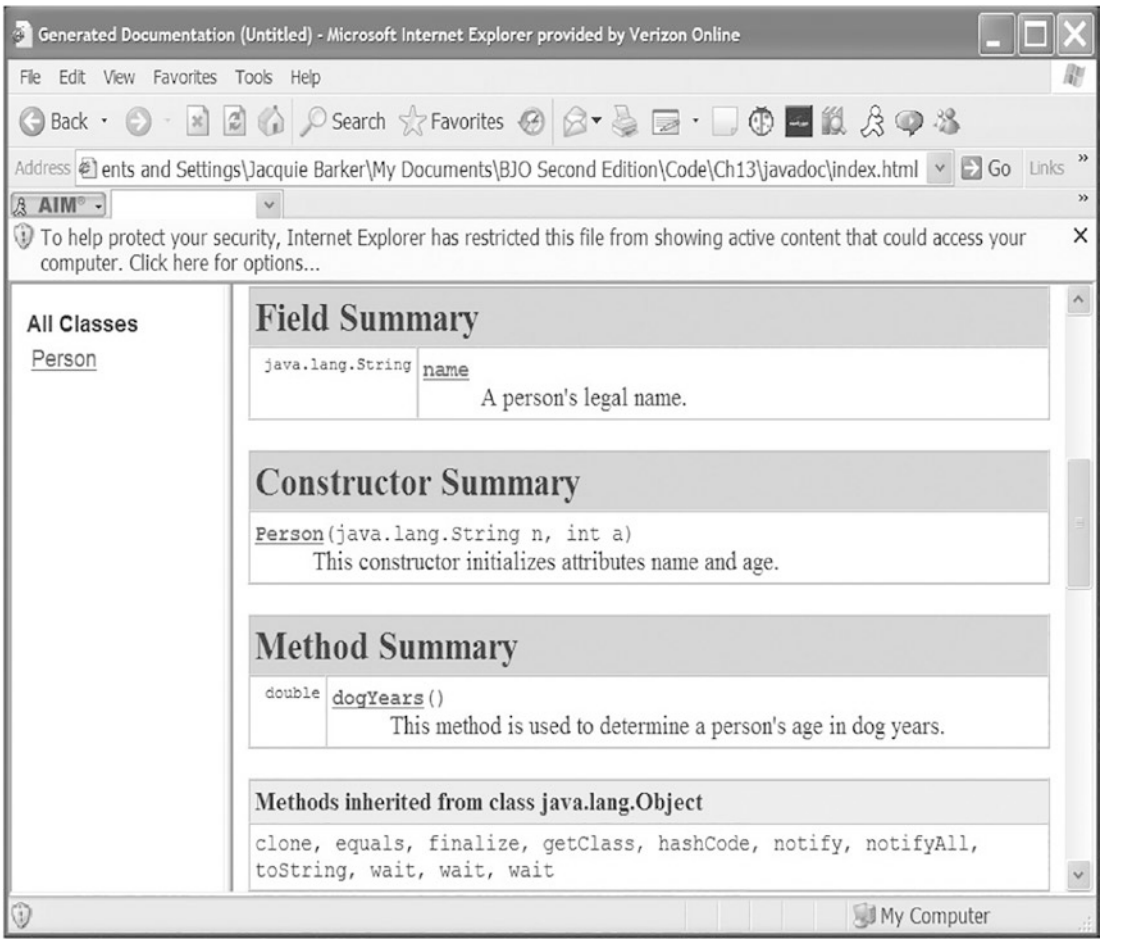


Figure 13-2. *Viewing the Field, Constructor, and Method Summaries for the Person class*

Scrolling down a bit further yet on the page as illustrated in Figures 13-3 and 13-4, we see additional details about the fields (attributes), constructors, and methods. Note in Figure 13-3 that our use of the `@param` tag in our constructor documentation has paid off: we see an explanation of each parameter under the **Parameters:** heading.

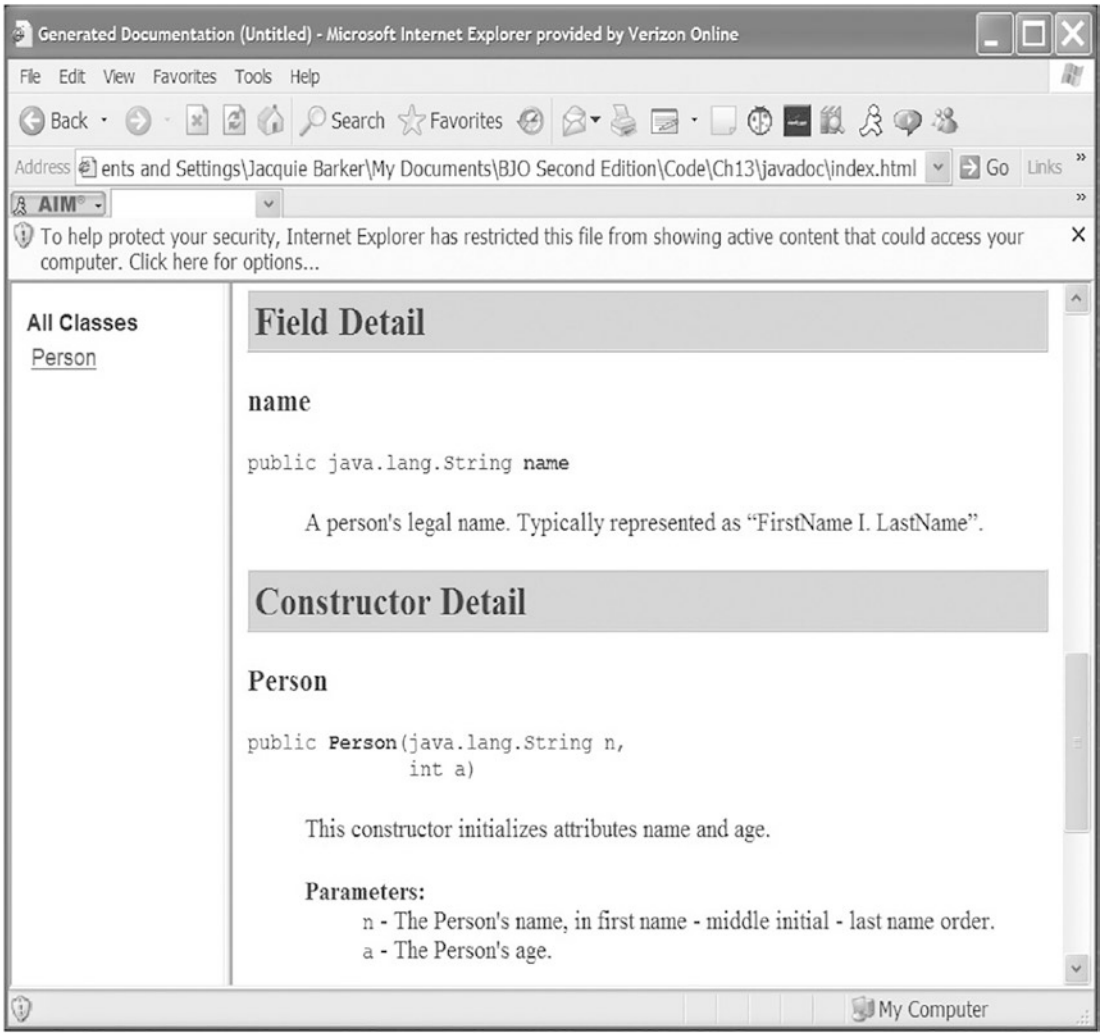


Figure 13-3. *Additional details concerning the Person class*

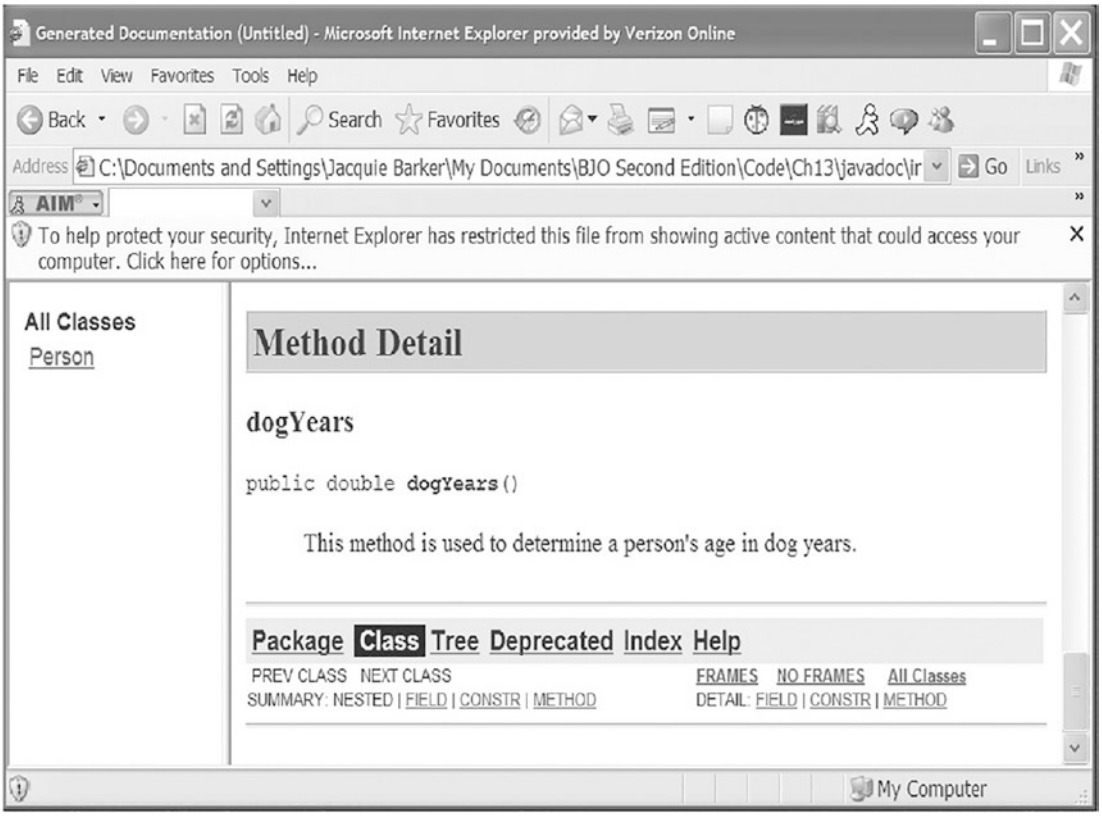


Figure 13-4. *Additional details concerning the Person class, continued*

Clicking the **Index** link at the top of the page, as illustrated in Figure 13-5, brings up an alternative view of our documentation as shown in Figure 13-6. Here, we are able to navigate through an alphabetical list of all class, attribute, constructor, and method names. Had we generated javadoc documentation for multiple classes–say, for all of the classes comprising the SRS–then ***all*** of these classes' ***combined*** features would be navigable via this ***consolidated*** index view.

Figure 13-5. *Clicking the Index link…*

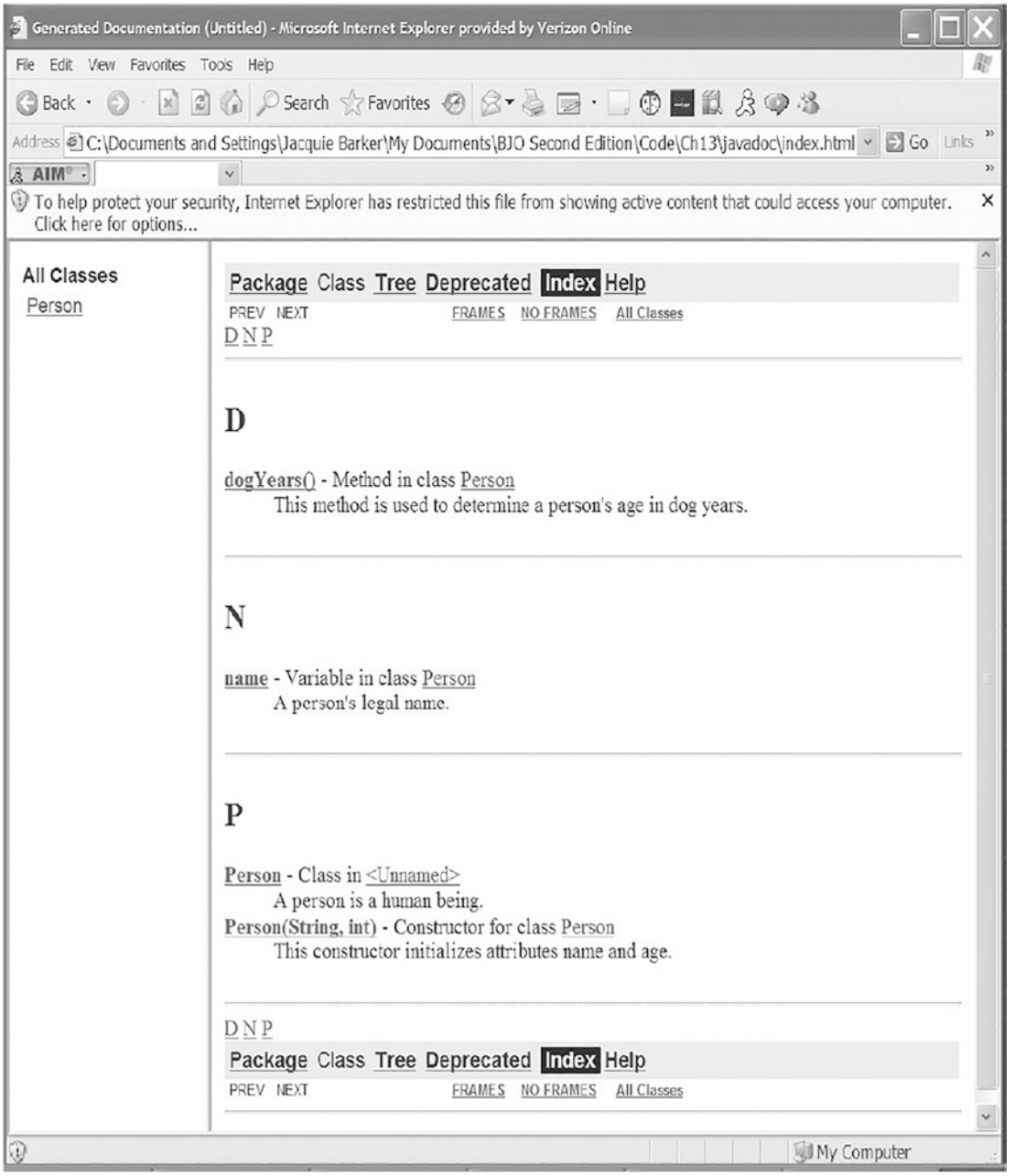


Figure 13-6. *... displays an alphabetical listing of all symbols*

The Object Nature of Strings

In Chapter 2, we introduced the `String` type along with eight other primitive Java types: `int`, `double`, `char`, `boolean`, `float`, `byte`, `short`, and `long`. At that time, we emphasized the fact that the symbol "`String`" must be capitalized, whereas the other eight type names are expressed in all lowercase. What we ***didn't*** make clear at the time is that `String` is a ***reference type***–that is, variables declared to be of type `String` refer to ***objects***. (Recall from our discussion in Chapter 3 that variables declared to be one of the primitive types in Java do ***not*** refer to objects.)

```
// s refers to an OBJECT of type String.
String s = "Java";
```

Thus, `String` is said to be a reference type whose structural and behavioral characteristics are defined by the `String` class, one of the classes defined within the core `java.lang` package.

Operations on Strings

As we learned in Chapter 2, the plus sign (+) operator is used to concatenate `String` values:

```
String x = "foo";
String y = "bar";
String z = x + y + "!";  // z assumes the value "foobar!"
```

But now that we appreciate the object nature of `String`s, we can also take advantage of the numerous methods that are declared by the `String` class for manipulating `String`s:

- `int length()`: This method, when applied to a `String` reference, returns the length of the `String` as an integer:

  ```
  // Continuing the previous example, where z equals "foobar!":
  int len = z.length();  // len now equals 7
  ```

- `boolean startsWith(String s)`: Returns `true` if the `String` to which this method is applied starts with the `String` provided as an argument to the method and `false` otherwise:

```
String s = "foobar";

// This will evaluate to true.
if (s.startsWith("foo")) ...
```

- `boolean endsWith(String s)`: Returns `true` if the `String` to which this method is applied ends with the `String` provided as an argument to the method and `false` otherwise:

```
String s = "foobar";

// This will evaluate to true.
if (s.endsWith("bar")) ...
```

- `boolean contains(String s)`: Returns `true` if the `String` to which this method is applied contains the `String` provided as an argument to the method and `false` otherwise:

```
String s = "foobar";

// This will evaluate to true.
if (s.contains("oo")) ...
```

- `int indexOf(String s)`: Returns a non-negative integer value indicating the starting character position (counting from 0) at which the `String` provided as an argument is found within the `String` to which this method is applied or a negative value (typically –1) if the `String` argument is not found:

```
String s = "foobar";
int i = s.indexOf("bar");  // i will equal 3
int j = s.indexOf("cat");  // j will equal -1
int k = s.indexOf("f");    // k will equal 0
```

- `String replace(char old, char new)`: Creates a brand-new `String` object in which all instances of the `old` character are replaced with the `new` character–the original `String`'s value is unaffected:

```
String s = "o1o2o3o4";
// Note use of single quotes around characters
```

```
// vs. double quotes around Strings.
String p = s.replace('o', 'x');   // p assumes the value
"x1x2x3x4", while
                                  // s retains the value
                                  "o1o2o3o4"
```

- `String substring(int i)`: Creates a brand-new `String` object by copying a substring of an existing `String` object starting at the `i`th position through the end of the existing `String`:

```
String s = "foobar";
String p = s.substring(3); // p assumes the value "bar"
```

- `String substring(int i, int j)`: Creates a brand-new `String` object by copying a substring of an existing `String` object starting at the `i`th and stopping just ***before*** the `j`th position:

```
String s = "foobar";
String p = s.substring(1, 5);  // p assumes the value "ooba";
```

- `char charAt(int index)`: Returns the `char`(acter) value located at the `i`th position within the `String`:

```
String s = "foobar";

// Iterate through a String character by character.
for (int i = 0; i < s.length(); i++) {
    System.out.println(s.charAt(i));
}
```

Output:

```
f
o
o
b
a
r
```

- `boolean equals(String)`: Compares the value of the `String` object to which this method is applied with the value of the `String` object whose reference is passed in as an argument; returns `true` if the values are the same and `false` if they are not:

```
String s = "dog";
String t = "cat";
String u = "dog";

// This will evaluate to true ...
if (s.equals(u)) { ...

// ... and this, to false.
if (s.equals(t)) { ...
```

Note that we generally should ***avoid*** using the double equal sign (`==`) operator to test the equality of two `String` objects' values; that is, the following can yield seemingly inconsistent results, depending on how we've instantiated `String` objects `s1` and `s2`:

```
// We generally want to AVOID doing this ...
if (s1 == s2) { ...
```

This is because the `==` operator, when used to compare reference types such as `String`s or `Person`s or generic `Object`s, is actually comparing their ***addresses in memory*** to see if the two variables are referring to the ***same exact object***, as illustrated in Figure 13-7.

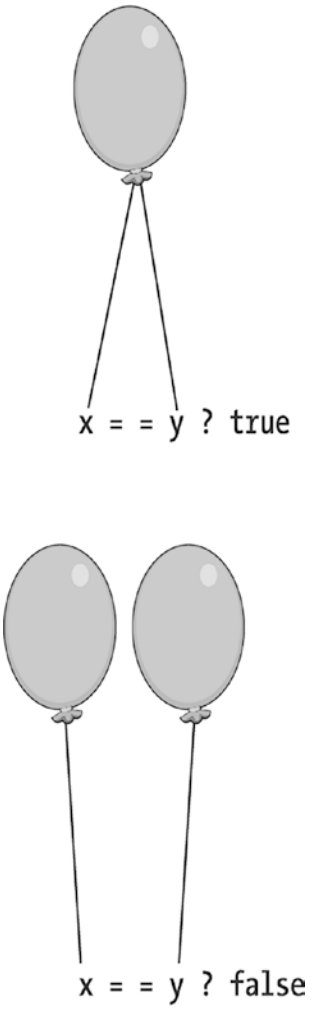


Figure 13-7. *The result of evaluating* `x == y` *can vary depending on how many String instances are involved*

We'll revisit this notion for objects in general, and for `String`s specifically, a bit later in the chapter.

Strings Are Immutable

Strings are said to be **immutable**: that is, the value of a particular `String` object cannot be changed once it has first been assigned at the time of instantiation. When we ***seem*** to be programmatically modifying an existing `String` object's value, we are actually creating a ***new*** `String` object with the desired value.

Let's look at an example of how this works. The following line of code would result in a `String` object with the value `"Foo"` being created somewhere in memory, as illustrated in Figure 13-8:

```
String x = "Foo";
```

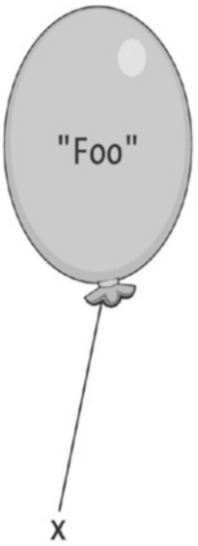


Figure 13-8. *A `String` object has been instantiated with the literal value `"Foo"`*

Continuing our example, this next line of code would result in a ***second*** `String` object with the value "`Foobar!`" being created somewhere else in memory:

```
// We're not really CHANGING the value of the specific String object
// originally referenced by x; rather, we're creating a new String
// object with the desired value for x to reference.
x = x + "bar!";
```

The original "`Foo`" `String` would still exist for some period of time, but is no longer directly accessible to us by reference and will soon be garbage collected–see Figure 13-9.

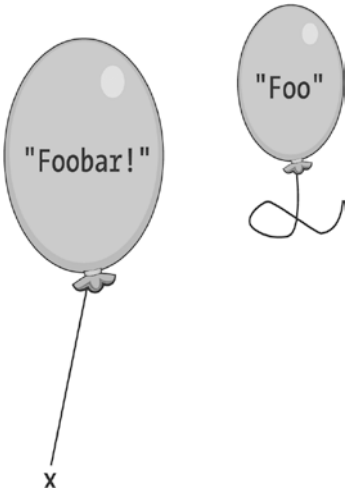


Figure 13-9. *When we assign a new value to* `String` *reference* `x`*, we're actually instantiating a new* `String` *object*

While the net result is the same from the programmer's perspective–namely, that the value of `x` will now be "`Foobar!`" as far as we're concerned

```
System.out.println(x);
```

Output:

```
Foobar!
```

the implication of this behind-the-scenes phenomenon is that building up long `String` values through iterative `String` concatenation can be quite inefficient. As an example, consider the following code:

```
// Initialize s to an empty String.
String s = "";

for (int x = 0; x < 10; x++) {
    // Append another digit to s.
    s = s + x;
}

System.out.println(s);
```

Output:

```
0123456789
```

With each successive iteration of the preceding `for` loop, we "change" the value of `s` by creating yet another `String` object: `"0"` in the first iteration, `"01"` in the second iteration, and so forth. By the time the loop is finished, we'll have created ***ten separate* `String` *objects*, *nine*** of which are waiting around to be garbage collected! This is illustrated in Figure 13-10.

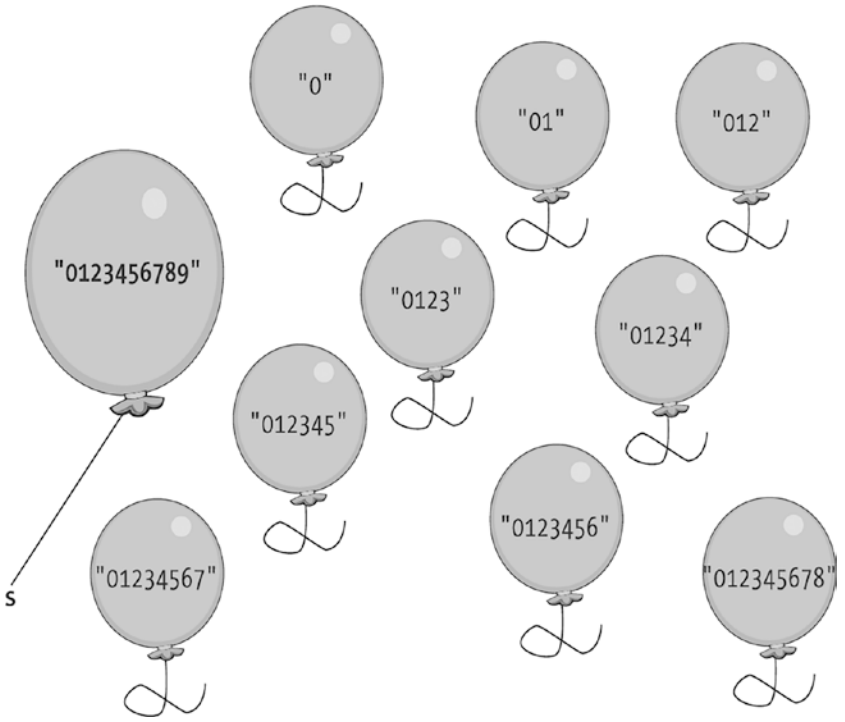


Figure 13-10. *Using* `String` *concatenation iteratively can be inefficient*

For this reason, the `java.util` package provides a special-purpose class for iteratively building up the value of a single `String` instance: the `StringBuffer` class.

The StringBuffer Class

Let's rewrite our previous example of concatenating digits using the `StringBuffer` class instead; remember that we'd have to include the `import` directive

```
import java.util.StringBuffer;
```

in whatever class this code is found:

```
// Instantiate an empty StringBuffer.
StringBuffer sb = new StringBuffer();

for (int x = 0; x < 10; x++) {
    // Append another digit to sb.
    sb.append(x);
}

// Extract the new String value from the StringBuffer.
String result = sb.toString();

// Let's see what we got!
System.out.println(result);
```

Our output will be the same as with the previous example:

```
0123456789
```

However, by switching to the `StringBuffer` approach of incremental `String` concatenation, we've instantiated only one object–a single `StringBuffer`–rather than creating ten `String` objects, nine of which are effectively wasted, as was the case with the previous version of this code (recall Figure 13-10). When the number of iterations increases–say, from 10 to 10,000–the performance improvement in using a `StringBuffer` can be dramatic.

Note that once we've finished "assembling" the `String` value of interest in a `StringBuffer`, we use the `toString` method of the `StringBuffer` class to extract the value as a `String` object:

```
String result = sb.toString();
```

We may use the add method of the StringBuffer class to append expressions of literally any type to a StringBuffer instance, because the add method is overloaded: there's a version that accepts a String expression as an argument, a version that accepts an int expression, and so forth.

The StringTokenizer Class

Another handy String-related class provided by the java.util package is the StringTokenizer class. With this class, we're able to **parse** (break apart) a String into **tokens** (segments/substrings) based on arbitrary delimiters.

The easiest way to learn how to use this class is with an example; again, we'd need to include the directive

```
import java.util.StringTokenizer;
```

in whatever class this code is found. We'll present the example in its entirety first and then will narrate it step by step:

```
String s = "This is a test.";
StringTokenizer st = new StringTokenizer(s);

while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

Output:

```
This
is
a
test.
```

Let's narrate the preceding example:

- The default StringTokenizer constructor takes one argument, the String value to be parsed, and parses along **white space** boundaries–blank spaces, tab characters, and the like:

  ```
  String s = "This is a test.";
  StringTokenizer st = new StringTokenizer(s);
  ```

- We use the `boolean hasMoreTokens` method of the `StringTokenizer` class to ascertain whether or not we've reached the end of the particular `String` instance being parsed; this method returns `true` if there are more tokens remaining or `false` if the `String` being parsed has been exhausted:

  ```
  while (st.hasMoreTokens()) {
  ```

- The `String nextToken` method is used to pluck out the next token/segment of the `String`:

  ```
      System.out.println(st.nextToken());
  }
  ```

A second overloaded form of constructor, `StringTokenizer(String s, String delimiter)`, can be used if we want to specify a ***specific*** delimiter to be used when parsing a `String`. For example, to parse along slash (/) boundaries, as when perhaps parsing a date, we'd write code as follows (observe that we enclose the delimiter, which can be ***one or more*** characters long, within ***double*** quote marks):

```
String date = "11/17/1985";

// Note use of double quote marks below.
StringTokenizer st = new StringTokenizer(date, "/");

while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

Output:

```
11
17
1985
```

Note that the delimiter–"/", in this example–is stripped off each token.

As another example, to parse on the three-character delimiter "-#-", we'd write

```
String fruit = "apple-#-banana-#-cherry";
StringTokenizer st = new StringTokenizer(fruit, "-#-");

while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

Output:

```
apple
banana
cherry
```

As our final example, let's assume that we're reading records one by one from a file and want to parse on tab characters only vs. on all white space; we'd write code as follows:

```
// Pseudocode.
String record = read a record from a file;

// Parse on tabs only, not on all white space.
StringTokenizer st = new StringTokenizer(record, "\t");

while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

Assuming that `record` contains the following text (where <*tab*> indicates the presence of an invisible tab character)

```
Bill Jost<tab>123-45-6789<tab>Cleveland, Ohio
```

we'd observe the following output:

```
Bill Jost
123-45-6789
Cleveland, Ohio
```

Note that the blank spaces between words have been left intact; had we instead parsed this record with the default `StringTokenizer`, which parses on all white space

```
// Pseudocode.
String record = read a record from a file;

// Parse on any/all white space.
StringTokenizer st = new StringTokenizer(record);

while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

we'd have gotten the following as output instead:

```
Bill
Jost
123-45-6789
Cleveland,
Ohio
```

`StringTokenizers` are particularly useful when reading and parsing structured records from data files; we'll use this technique in building our SRS application, in Chapter 15.

Instantiating Strings and the String Literal Pool

There are two ways to instantiate `String` objects in Java. The first, which we've seen in use many times before, allows us to simply assign a literal value to a `String` variable:

```
String s = "I am a String!";
```

But, because `Strings` are objects, we may also use the `new` keyword to formally invoke an explicit `String` constructor, as follows:

```
String t = new String("I am a String, too!");
```

We'll refer to the first way of instantiating `Strings`–the way that ***doesn't*** use the `new` operator–as the ***"shortcut" method*** of `String` instantiation and the second way as the

formal method. There is a subtle difference in terms of what happens behind the scenes with these two methods of `String` instantiation, as follows.

When we use the shortcut method of `String` instantiation

```
String x = "Foo"; // shortcut method
```

the JVM checks its `String` **literal pool**–a special place in the JVM's memory that enables automatic shared access to `String` objects–to see if there is already a `String` object in the literal pool with that exact same value:

- If so, the JVM ***reuses*** that existing instance without creating a second.
- Conversely, if a matching instance is ***not*** found in the literal pool, the JVM creates one and places it in the literal pool. This is illustrated conceptually in Figure 13-11.

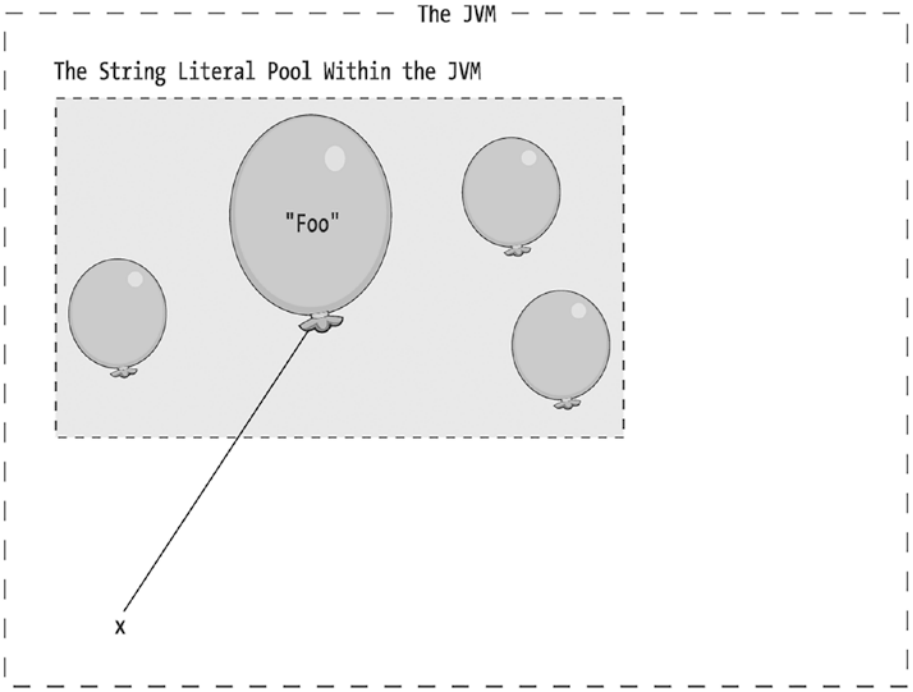


Figure 13-11. *Using the shortcut method of `String` instantiation inserts a newly created `String` object into the `String` literal pool*

Let's now use the shortcut method a second time in the same program, in an attempt to instantiate another `String` object `y` with the same value as `x`:

```
String x = "Foo";
String y = "Foo"; // shortcut method again
```

This time, since the JVM finds an instance of a `String` with the value `"Foo"` in the literal pool from when we instantiated `x`, the JVM assigns `y` as a ***second*** handle to the ***same*** `String` object. This is illustrated conceptually in Figure 13-12.

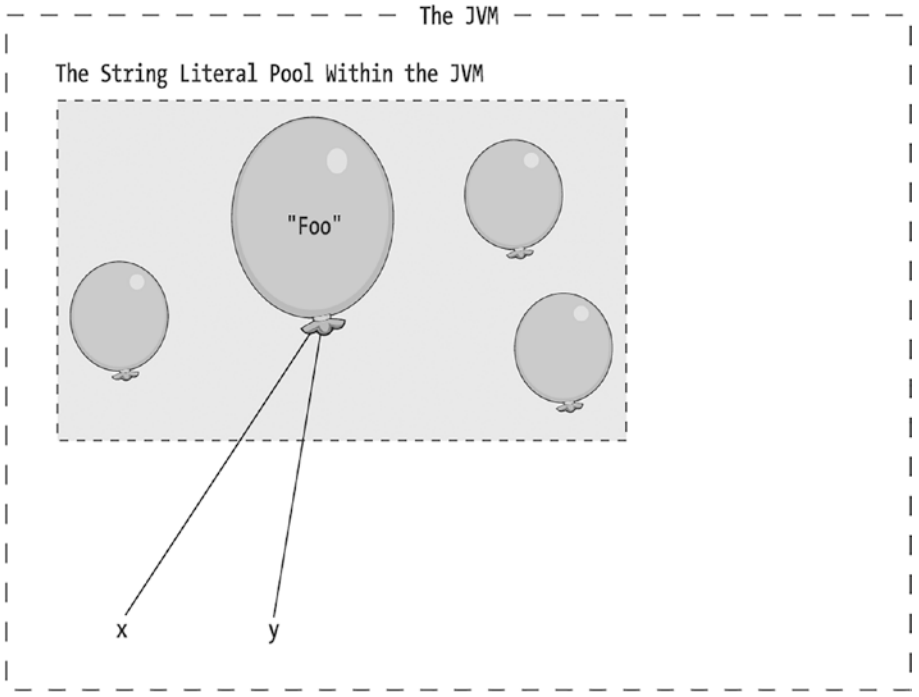


Figure 13-12. *Two shortcut instantiations of* `String`*s with the same value wind up sharing the same* `String` *object within the* `String` *literal pool*

Now, let's declare and instantiate a third `String` variable `z` in the same program, giving it the same value as `x` and `y`. However, rather than using the shortcut method for `String` instantiation with `z`, we'll use the `new` keyword to explicitly invoke a `String` constructor:

```
String x = "Foo";
String y = "Foo";
String z = new String("Foo");  // formal method this time
```

Our use of new instructs the JVM to ***bypass*** the literal pool: that is, a ***brand-new*** instance of a String object with the value "Foo" will be created ***outside*** of the literal pool, as illustrated in Figure 13-13.

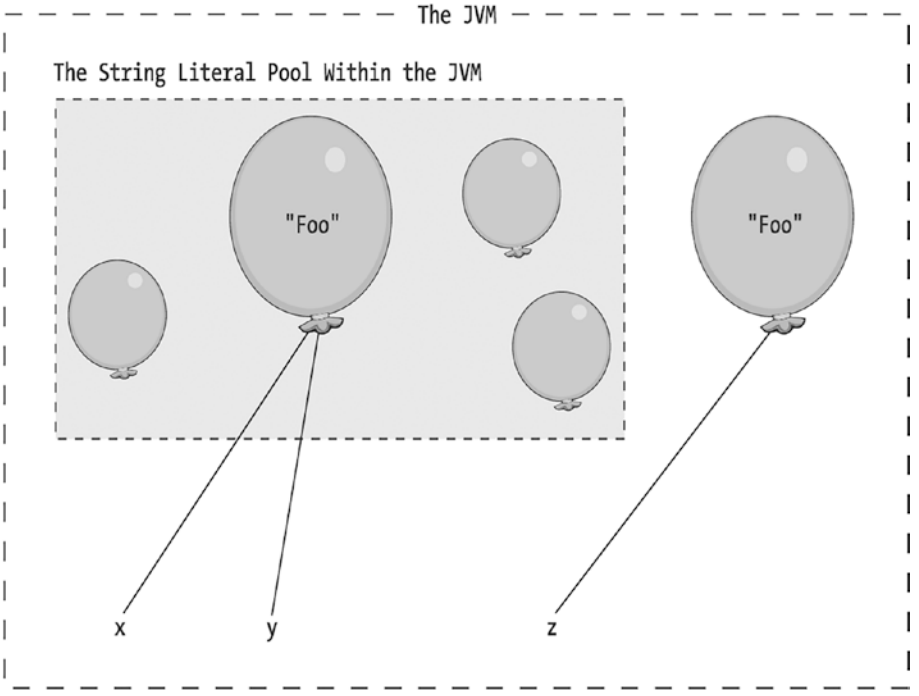


Figure 13-13. *Use of the new keyword to instantiate a String, on the other hand, explicitly creates a new String instance every time*

Figure 13-14 completes our example, wherein we create a fourth similarly valued String, w, again using the formal method:

```
String x = "Foo";
String y = "Foo";
String z = new String("Foo");
String w = new String("Foo");  // formal method again
```

As expected, our use of new once again circumvents the literal pool, and another distinct String object is created.

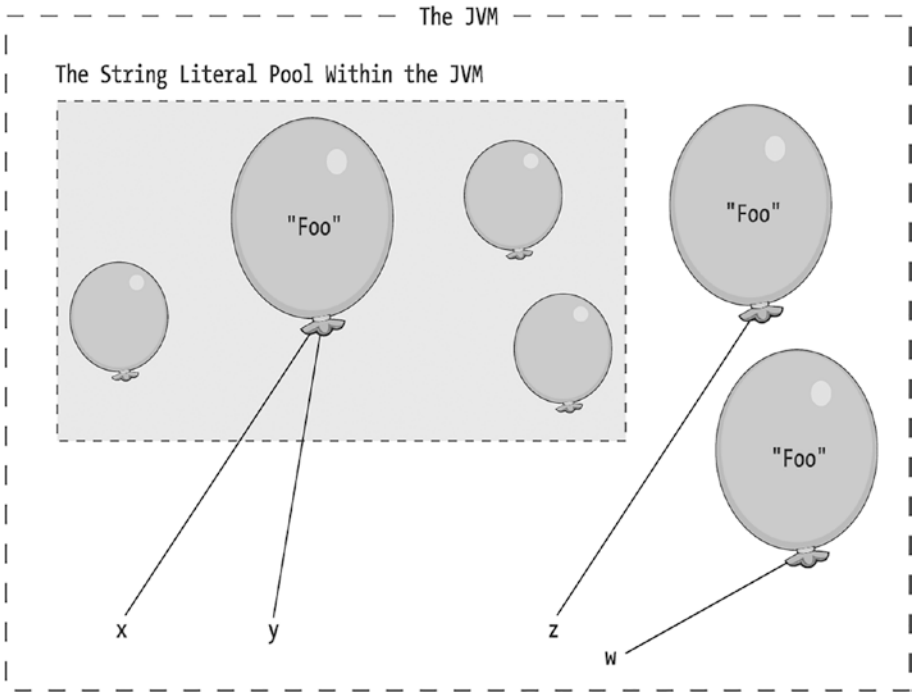


Figure 13-14. *Another distinct* `String` *object is created*

Because we used a combination of formal and shortcut `String` instantiation in our example, we wound up with three different `String` objects in memory–one in the literal pool and two in the general memory of the JVM, all with the same value `"Foo"`. Had we consistently used the shortcut approach when instantiating all four `String` objects, we'd have only created one such instance, and hence our JVM's memory would have been less cluttered. For this reason, the shortcut form of `String` instantiation is the preferred `String` construction technique in most situations.

Testing the Equality of Strings

We mentioned earlier that testing the equality of `String`s using the == operator

```
String s1;
String s2;
// Instantiation details omitted ...

if (s1 == s2) { ...
```

rather than using the `String` class's `equals` method

```
String s1;
String s2;
// Instantiation details omitted ...
```

`if (s1.equals(s2)) { ...`

can yield seemingly puzzling results, if one doesn't understand the difference between the ***formal*** and ***shortcut*** methods for instantiating `String`s. Using the four `String` references `w`, `x`, `y`, and `z` as represented in Figure 13-14, we'd get the following results from various tests of equality:

- Both of these first two tests, comparing `x` and `y`, evaluate to `true`

  ```
  // Are the VALUES of x and y the same?
  if (x.equals(y)) { ... // true

  // Are x and y referring to the SAME OBJECT?
  if (x == y) { ... // true
  ```

  because `x` and `y` not only have the same ***value*** `"Foo"` but they also refer to the ***same specific*** `String` ***object***.

- Where we see a difference is when we attempt to compare `w` and `z` in similar fashion:

  ```
  // Are the VALUES of w and z the same?
  if (w.equals(z)) { ... // true

  // Are w and z referring to the SAME OBJECT?
  if (w == z) { ... // FALSE!!!
  ```

  The first test evaluates to `true`, because the ***values*** of `w` and `z` are equivalent, but the second test evaluates to `false`, because `w` and `z` refer to two ***different*** objects.

In the vast majority of cases, when we say we wish to compare two `String`s, we typically mean that we want to compare their ***values***, ***not*** their unique identities as objects. Thus, we virtually always compare `String`s for equality using the `equals` method rather than the `==` operator.

We'll revisit this distinction between the == operator and the `equals` method once more, when we talk about generic `Objects` later in this chapter.

Message Chains

When we talked about Java expressions in Part 1 of the book, there was one form of expression that we omitted: namely, **message chains**. We've repeated our list of what constitutes Java expressions from Chapter 4 in the following, ***highlighting*** this latest addition:

- ***A constant***: 7, false
- ***A char(acter) literal***: `'A'`, `'&'`
- ***A String literal***: `"foo"`, `""`
- ***The name of any variable declared to be of one of the predefined types that we've seen so far***: `myString, x`
- ***Any one of the preceding that is modified by one of the Java unary operators***: `i++`
- ***A message/method invocation on an object reference***: `z.length()`
- ***Any two of the preceding that are combined with one of the Java binary operators***: `z.length() + 2`
- ***A "chain" of two or more messages concatenated by periods ("dots"):*** `p.getName().length()`
- ***Any of the preceding types of expression enclosed in parentheses***: `(p.getName().length() + 2)`

With Java (and other OOPLs), it is quite commonplace to form complex expressions by concatenating (and sometimes nesting) method calls. As with all expressions, we evaluate complex method expressions from innermost to outermost parentheses, left to right. So, based on the following code example

```
// Instantiate three objects.
Student s = new Student();
Professor p = new Professor();
Department d = new Department();
```

```
// Set selected attribute values.
d.setName("MATH");
p.setDepartment(d);
s.setAdvisor(p);
```

let's evaluate the following line of code:

```
s.setMajor(s.getAdvisor().getDepartment().getName());
```

- There are two levels of parenthesis nesting in this example, but the inner sets of parentheses represent the argument signatures of the various methods being invoked, and so we'll focus on evaluating the expression within the ***outer*** set of parentheses:

  ```
  s.getAdvisor().getDepartment().getName()
  ```

- Evaluating this expression from left to right, the first subexpression, `s.getAdvisor()`, returns a reference to a `Professor` object; it's as if we've simplified our original line of code as follows:

  ```
  // From:
  s.setMajor(s.getAdvisor().getDepartment().getName());

  // To:
  s.setMajor(p.getDepartment().getName());
  ```

- Next, we apply the `getDepartment` method to this `Professor`, which returns a reference to a `Department`; it's as if we've simplified the original line of code even further, as follows:

  ```
  // From:
  s.setMajor(p.getDepartment().getName());

  // To:
  s.setMajor(d.getName());
  ```

- Next, we apply the `getName` method to this `Department`, which returns a reference to a `String` object whose value is `"Math"`; it's as if we've simplified the original expression further still, as follows:

```
// From:
s.setMajor(d.getName());

// To:
s.setMajor("Math");
```

- Finally, we evaluate the statement as a whole, which assigns the `String` value `"Math"` as the major field for `Student s`.

We'll see many such concatenated method calls in the SRS code.

The ***type*** of a chained message expression is the type of the result that the ***last*** method in the chain returns. For example, if

- `s` is a `Student` reference
- `getAdvisor` is a `Student` method that returns a `Professor` reference
- `getName` is a `Professor` method that returns a `String` reference
- `length` is a `String` method that returns an `int` value

then `s.getAdvisor().getName().length()` is an `int`(eger) expression.

Object Self-Referencing with "this"

We've previously seen the `this` keyword used in two different ways:

- In Chapter 4, we learned that the syntax `this.`*featureName* can be used within any (non-static) method to emphasize the fact that we're accessing another feature of ***this same object***:

```
public class SomeClass {
    // Details omitted.

    public void foo() {
        // Call method bar (declared below) from within foo.
        this.bar();
        // etc.
    }

    public void bar() {
```

```
            // ...
        }
    }
```

- In that same chapter, we also learned that, from within any ***constructor*** of a class *X*, we can invoke any other constructor of the same class *X* via the syntax

  `this(`*`optional arguments`*`);`

so as to reuse code from one constructor to another.

We'll now explore a third context in which the `this` keyword may be used.

In client code, such as the `main` method of a program, we declare reference variables so as to assign symbolic names to objects:

```
Student s = new Student();  // s is a reference variable of type Student.
```

We can then conveniently manipulate the objects that these reference variables refer to by manipulating the reference variables themselves:

```
s.setName("Fred");
```

When we are executing the code that comprises the body of one of an object's own methods, we sometimes need the object to be able to refer to ***itself***–that is, to **self-reference**, as in this next bit of code (please read inline comments carefully):

```
public class Student {
  Professor facultyAdvisor;
  // Other details omitted.

  public void selectAdvisor(Professor p) {
    // We're down in the "bowels" of the selectAdvisor method,
    // executing this method for a particular Student object/instance.
    // We save the handle on our new advisor as one of our attributes ...
    this.setFacultyAdvisor(p);

    // ... and now we want to turn around and tell this Professor object to
    // reflect us as its student advisee. The Professor class has a
    // method with the header:  public void addAdvisee(Student s)
```

```
    // and so all we need to do is invoke this method on our
    advisor object,
    // passing in a reference to ourself as a Student; but, who the heck
    // are we? That is, how do we refer to ourself?
    facultyAdvisor.addAdvisee(???);
  }
}
```

Within the body of a method, when we need a way to refer to the object whose method we are executing, we use the reserved word `this` to "self-reference." So, with respect to our preceding example, the line of code **highlighted** in the following would do the trick nicely:

```
public class Student {
  Professor facultyAdvisor;
  // Other details omitted.

  public void selectAdvisor(Professor p) {
    this.setFacultyAdvisor(p);
    p.addAdvisee(this); // passing a reference to THIS Student
  }
}
```

Specifically, it would pass a reference to ***this*** `Student`–the `Student` object whose `selectAdvisor` method we are executing–as an argument to the `addAdvisee` method as invoked on `Professor facultyAdvisor`.

Java Exception Handling

Unexpected problems can arise as the JVM interprets/executes a Java program; for example

- A program may be unable to open a data file due to inappropriate permissions.
- A program may have trouble establishing a connection to a database management system because a user has supplied an invalid password.

- A user may supply inappropriate data via an application's user interface–for example, a non-numeric value where a numeric value is expected.
- The problem may be something as simple as a logic error that the compiler wasn't able to detect.

Suppose we were to write the following simple program:

```
public class Problem {
    public static void main(String[] args) {
        // Declare two Student object references and initialize them
        // to the value null, which is the "zero-equivalent" value
        // for object references; that is, null indicates that the variable
        // does not currently reference an object.
        Student s1 = null;
        Student s2 = null;

        // Details omitted ...

        // Later on, we instantiate an object for s1 to refer to,
        but forget
        // to do so for s2.
        s1 = new Student();

        // More details omitted ...

        // Still later in our program, we attempt to assign names to both
        Students.
        // This line of code is fine ...
        s1.setName("Fred");

        // ... but this next line of code causes a run-time problem,
        because we
        // are trying to invoke a method on -- i.e., "talk to" -- a
        // non-existent object.
        s2.setName("Mary");
    }
}
```

The preceding code would compile without error, but if we were to execute this program, the JVM would report the following run-time error:

```
Exception in thread "main" java.lang.NullPointerException
        at Problem.main(Problem.java:22)
```

We refer to (recoverable) Java run-time errors as **exceptions** and refer to the process whereby the JVM reports that a run-time error has arisen as **throwing an exception**. With respect to this example specifically, the JVM threw a `NullPointerException` on line 22 of the `Problem` class:

```
s2.setName("Mary");  // this is line 22
```

A `NullPointerException` arises whenever we try to invoke a method on an object reference (`s2`, in this example) whose value is `null` or, in plain English, whenever we try to "talk to" a nonexistent object.

When the JVM throws an exception, it's as if the JVM is shooting off a signal flare to notify an application that something has gone wrong, in order to give the application a chance to ***recover*** from the problem if we've equipped it to do so. Through a technique known as **exception handling,** we can design our applications so that they may anticipate and gracefully recover from such exceptions at run time.

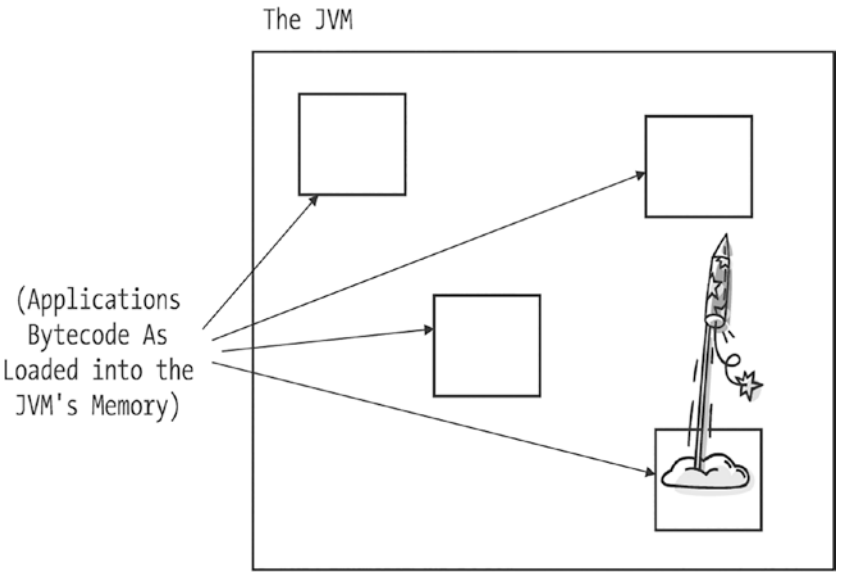


Figure 13-15. *When the JVM throws an exception, it is effectively shooting off a signal flare to notify an application of a problem that has arisen*

The Mechanics of Exception Handling

The basics of the exception handling mechanism are as follows.

The try Block

We enclose code that is likely to throw an exception inside of a pair of braces to form a code block and precede the opening brace with the keyword `try`. This indicates our intention to **catch**–that is, to detect and respond to–any exceptions that might by thrown by the JVM while executing the code within that **try block**.

Going back to our previous example, we might amend our code as **highlighted** in the following:

```
public class Problem2 {
    public static void main(String[] args) {
        Student s1 = null;
        Student s2 = null;

        // Details omitted ...

        // Later on, we instantiate an object for s1 to refer to, but
        // forget to do so for s2.
        s1 = new Student();

        // More details omitted ...

        // We've added a try statement to enclose the code likely to
        // generate an exception.
        try {
            s1.setName("Fred");
            s2.setName("Mary");
        }
        // There's more to follow ... stay tuned!
```

In order for the preceding code to compile, the `try` block must be immediately followed by at least one `catch` or `finally` block.

The catch Block

Each `catch` block begins with a declaration of the form

```
catch (ExceptionType variableName) { ...
```

where the keyword `catch` is followed by parentheses enclosing a specific type of exception that is to be caught; the braces that follow the closing parenthesis enclose the code to be used in recovering from the exception:

```
catch (ExceptionType variableName) {
    // Pseudocode.
    recovery code for an ExceptionType exception goes here ...
}
```

(For now, don't worry about the *variableName* that is also declared within these parentheses–we'll discuss its purpose shortly.)

One ***or more*** `catch` blocks can be associated with the same `try` block, as illustrated in the following:

```
try {
    code liable to throw exception(s) goes here ...
}
catch (ExceptionType1 variableName1) {
    recovery code for ExceptionType1 goes here ...
}
catch (ExceptionType2 variableName2) {
    recovery code for ExceptionType2 goes here ...
}
// etc.
```

There are many different exception types built into the Java language, each one defined by a different predefined Java class derived from a common ancestor class called `Exception`. Before we discuss some of the more important Java exception types, however, let's finish amending our previous example involving `Student` references by adding several `catch` clauses (**bolded** in the following) to the `try` statement. (Depending on what type of exceptions we're catching, we may need `import` directives; we do not need to import `NullPointerException`, however, as it is within the core `java.lang` package.)

```
public class Problem3 {
    public static void main(String[] args) {
        Student s1 = null;
        Student s2 = null;

        // Details omitted ...


        // Later on, we instantiate an object for s1 to refer to, but
        // forget to do so for s2.
        s1 = new Student();

        // More details omitted ...

        // We've added a try block to enclose the code likely to
        // generate an exception.
        try {
            s1.setName("Fred");
            s2.setName("Mary");
        } // end of try block
        // Here are our catch clauses (three in total).
        catch (ArithmeticException e) {
            // Pseudocode.
            recovery code for an ArithmeticException goes here ...
        }
        catch (NullPointerException e2) {
            // Here's where we write the code for what the program
            should do if
            // a NullPointerException occurred.
            System.out.println("Darn!  We forgot to initialize " +
                "all of the students!");
            // etc.
        }
        catch (ArrayIndexOutOfBoundsException e3) {
            // Pseudocode.
            recovery code for an ArrayIndexOutOfBoundsException goes
            here ...
```

```
        }

        // etc. ...
    }
}
```

Note that the code within the `try` block of the preceding example doesn't throw either `ArithmeticExceptions` or `ArrayIndexOutOfBoundsExceptions`; these are merely included for purposes of illustration.

With regard to exception handling, there are two possible paths through this `main` method's logic. As long as the code in the `try` block executes without error, all of the `catch` blocks are ***bypassed***, and execution of our program continues after the end of the last `catch` block, as illustrated in Figure 13-16.

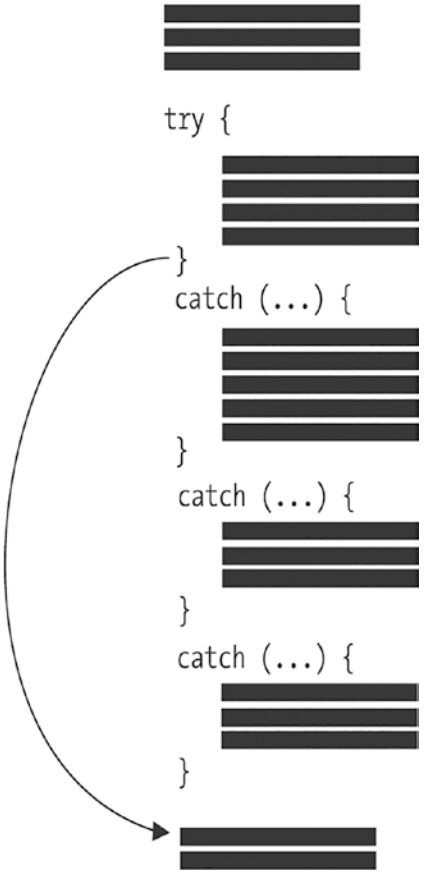


Figure 13-16. *If no exceptions are thrown in the* `try` *block, all* `catch` *blocks are bypassed*

If, on the other hand, an exception ***is*** thrown by the JVM while executing the `try` block, execution of the `try` block abruptly terminates as of the line on which the exception occurred. The JVM then examines the `catch` clauses in order from top to bottom, looking for a `catch` clause whose declared exception type addresses the type of exception that was thrown:

- If a match is found, the JVM executes the associated `catch` block; in the case of multiple matches, only the ***first*** such match is executed. Then, execution resumes after the ***end*** of the ***last*** catch block. This is illustrated in Figure 13-17.

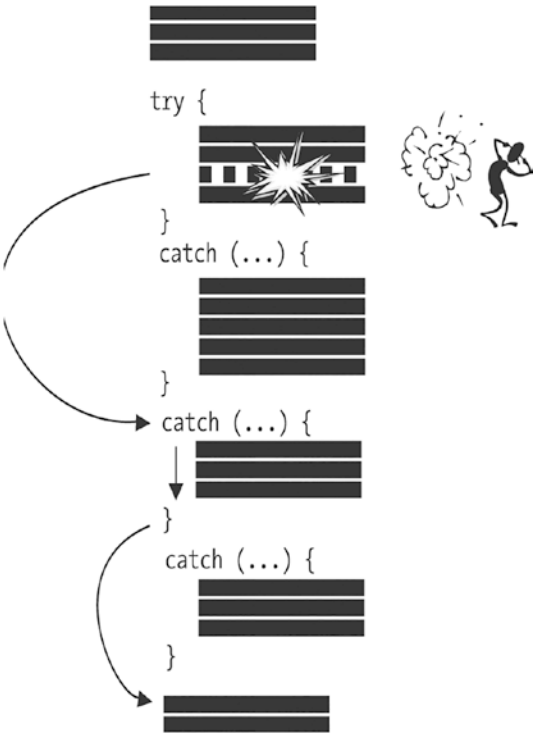


Figure 13-17. *If an exception arises, the first matching catch block, if any, is executed, and the rest are skipped*

- If no matching `catch` clause is found, the exception is said to be ***uncaught*** and, in the case of our preceding example, would be reported to the command window as we saw earlier:

```
Exception in thread "main" java.lang.NullPointerException
       at Problem.main(Problem.java: ...)
```

To help illustrate this flow of control, let's insert print statements into our previous `Problem`*x* example to produce a trace of our program's execution:

```
public class Problem4 {
    public static void main(String[] args) {
        Student s1 = null;
        Student s2 = null;
```

```
        // Details omitted ...

        // Later on, we instantiate an object for s1 to refer to, but
        // forget to do so for s2.
        s1 = new Student();

        // More details omitted ...

        System.out.println("We're about to enter the try block ...");
        try {
            System.out.println("We're about to call s1.setName(...)");
            s1.setName("Fred");
            System.out.println("We're about to call s2.setName(...)");
            s2.setName("Mary");
            System.out.println("We've reached the end of the try
            block ...");
        }
        // Here are our catch blocks (three in total).
        catch (ArithmeticException e) {
            System.out.println("Executing the first catch block ...");
        }
        catch (NullPointerException e2) {
            System.out.println("Executing the second catch block ...");
        }
        catch (ArrayIndexOutOfBoundsException e3) {
            System.out.println("Executing the third catch block ...");
        }

        System.out.println("We're past the last catch block ...");
    }
}
```

When executed, the following messages would be printed:

```
We're about to enter the try block ...
We're about to call s1.setName(...)
We're about to call s2.setName(...)
Executing the second catch block ...
We're past the last catch block ...
```

The finally Block

A `try` block may optionally have a `finally` block associated with it; if provided, a `finally` block follows any `catch` blocks associated with that same `try` block.

The code within a `finally` block is ***guaranteed*** to execute no matter what happens in the `try`/`catch` code that precedes it, that is, whether

- The `try` block executes to completion without throwing any exceptions whatsoever.
- The `try` block throws an exception that is handled by one of the `catch` blocks.
- The `try` block throws an exception that is ***not*** handled by ***any*** of the `catch` blocks (a situation that we'll explore a bit later in this chapter).

Let's add a `finally` block to our evolving `Problem`*x* program example; we'll use the version that included print statements, so that we may once again trace our code's execution:

```
public class Problem5 {
    public static void main(String[] args) {
        Student s1 = null;
        Student s2 = null;

        // Details omitted ...

        // Later on, we instantiate an object for s1 to refer to, but
        // forget to do so for s2.
        s1 = new Student();

        // More details omitted ...
```

```
        System.out.println("We're about to enter the try block ...");
        try {
            System.out.println("We're about to call s1.setName(...)");
            s1.setName("Fred");
            System.out.println("We're about to call s2.setName(...)");
            s2.setName("Mary");
            System.out.println("We've reached the end of the try
            block ...");
        }
        // Here are our catch blocks (three in total).
        catch (ArithmeticException e) {
            System.out.println("Executing the first catch block ...");
        }
        catch (NullPointerException e2) {
            System.out.println("Executing the second catch block ...");
        }
        catch (ArrayIndexOutOfBoundsException e3) {
            System.out.println("Executing the third catch block ...");
        }
        finally {
            System.out.println("Executing the finally block ...");
        }

        System.out.println("We're past the last catch block ...");
    }
}
```

When executed, this version of the program would produce the following output:

```
We're about to enter the try block ...
We're about to call s1.setName(...)
We're about to call s2.setName(...)
Executing the second catch block ...
Executing the finally block ...
We're past the last catch block ...
```

Let's now ***repair*** the code in our `try` block so that it doesn't throw any exceptions, to see what output our program will produce:

```
public class NoProblemo {
    public static void main(String[] args) {
        Student s1 = null;
        Student s2 = null;

        // Details omitted ...

        // Later on, we instantiate objects for both s1 and s2
        // to refer to, thus alleviating the NullPointerException
        // in the try block below.
        s1 = new Student();
        s2 = new Student();

        // More details omitted ...

        System.out.println("We're about to enter the try block ...");
        try {
            System.out.println("We're about to call s1.setName(...)");
            s1.setName("Fred");
            System.out.println("We're about to call s2.setName(...)");
            s2.setName("Mary");
            System.out.println("We've reached the end of the try
            block ...");
        }
        // Here are our catch blocks (three in total).
        catch (ArithmeticException e) {
            System.out.println("Executing first catch block ...");
        }
        catch (NullPointerException e2) {
            System.out.println("Executing second catch block ...");
        }
        catch (ArrayIndexOutOfBoundsException e3) {
            System.out.println("Executing third catch block ...");
        }
```

```
        finally {
            System.out.println("Executing finally block ...");
        }

        System.out.println("We're past the last catch block ...");
    }
}
```

When executed, this version of the program would produce the following output–note that all `catch` blocks are ***bypassed***:

```
We're about to enter the try block ...
We're about to call s1.setName(...)
We're about to call s2.setName(...)
We've reached the end of the try block ...
Executing the finally block ...
We're past the last catch block ...
```

A final variation on our `Problem`*x* program once again throws a `NullPointerException`, but does ***not*** catch the appropriate type of exception in any of the `catch` blocks:

```
public class Problem6 {
    public static void main(String[] args) {
        Student s1 = null;
        Student s2 = null;

        // Details omitted ...

        // Later on, we instantiate an object for s1 to refer to, but
        // forget to do so for s2.
        s1 = new Student();

        // More details omitted ...

        System.out.println("We're about to enter the try block ...");
        try {
            System.out.println("We're about to call s1.setName(...)");
            s1.setName("Fred");
```

```
            System.out.println("We're about to call s2.setName(...)");
            s2.setName("Mary");
            System.out.println("We've reached the end of the try
            block ...");
        }
        // Here are our catch blocks (two in total) - note that we
        // are *not* catching NullPointerException this time.
        catch (ArithmeticException e) {
            System.out.println("Executing first catch block ...");
        }
        catch (ArrayIndexOutOfBoundsException e2) {
            System.out.println("Executing second catch block ...");
        }
        finally {
            System.out.println("Executing finally block ...");
        }

        System.out.println("We're past the last catch block ...");
    }
}
```

When executed, this version of the program would produce the following output:

```
We're about to enter the try block ...
We're about to call s1.setName(...)
We're about to call s2.setName(...)
Executing the finally block ...
```

Note that we never make it to the final print statement

```
System.out.println("We're past the last catch block ...");
```

because an ***uncaught*** exception causes the overall code block in which it occurs–in our example, the `main` method–to terminate abnormally.

We thus see that no matter what happens in the `try`/`catch` code, a `finally` block, if present, is ***always*** executed.

Note that we may omit `catch` clauses for a `try` block if a `finally` block is present:

```
// Catch blocks are unnecessary if a finally block is present.
try { ... }
finally { ... }
```

Catching Exceptions

If the method in which an exception arises does not catch the exception, then the client code that called that method is given a chance to catch it.

Consider the following three-class example:

- In our `main` method, we invoke `methodX` on a `Student` object:

```
public class MainProgram {
    public static void main(String[] args) {
        Student s = new Student();
        s.methodX();
    }
}
```

- In the `Student`'s `methodX` code, we in turn invoke `methodY` on a `Professor` object:

```
public class Student {
    // Details omitted.
    public void methodX() {
        Professor p = new Professor();
        p.methodY();
    }
}
```

- Finally, here's the `Professor` class:

```
public class Professor {
    // Details omitted.
    public void methodY() {
        // Details omitted ...
    }
}
```

When the JVM executes our `MainProgram`, its `main` method is invoked, which in turn invokes `s.methodX()`, which in turn invokes `p.methodY();` this produces what is known as a **call stack** at run time, illustrated in Figure 13-18.

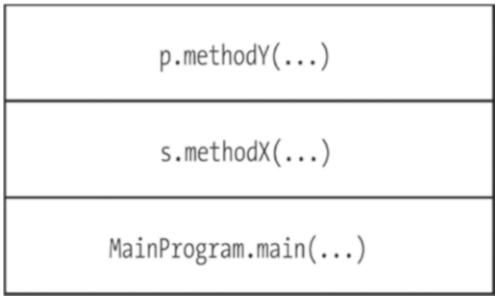


Figure 13-18. *The JVM keeps track of the order in which methods are called one from another by creating a call stack*

A **stack** is a last in, first out (LIFO) data structure; the most recent method call is **pushed** onto the top of the call stack, and when that method exits, it is removed from (**popped** off) the call stack. So, as an example, when `methodY` finishes executing and execution control is returned to `methodX`, `methodY` is popped off as illustrated conceptually in Figure 13-19.

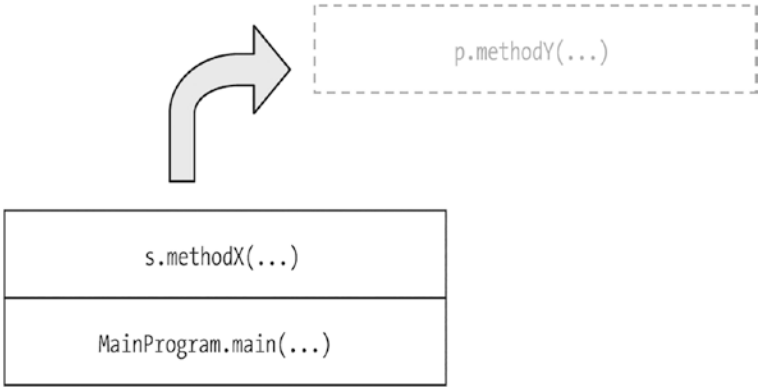


Figure 13-19. *When a method exits, it is removed from the call stack*

Let's now assume that a `NullPointerException` is thrown while executing `methodY`. If the appropriate `try/catch` logic is incorporated ***within the body of*** `methodY` to handle/resolve such a `NullPointerException`, as shown in the following

```
public class Professor {
    // Details omitted.
```

```
    public void methodY() {
        try { ... }
        catch (NullPointerException e) { ... }
    }
}
```

then neither the `Student` nor `MainProgram` class will be aware that such an exception was ever thrown. From the perspective of the call stack, awareness of the exception is contained within the current level of the stack, as illustrated in Figure 13-20.

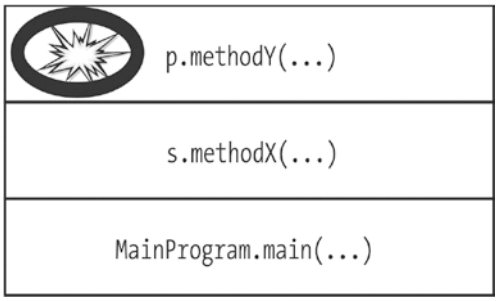


Figure 13-20. *The try/catch logic within* `methodY` *limits awareness of the exception to the current level in the call stack*

Let's assume instead that `methodY` does ***not*** catch/handle `NullPointerExceptions`:

```
public class Professor {
    // Details omitted.
    public void methodY() {
        // A NullPointerException is thrown here, but
        // is NOT caught/handled.
        // (Details omitted.)
    }
}
```

If a `NullPointerException` is thrown while executing ***this*** version of `methodY`, it will travel down the call stack one level to the `Student` class's `methodX`, which is where the call to `p.methodY()` originated. If the `Student` class's `methodX` code happens to include the necessary exception handling code, as shown in the following

```
public class Student {
    // Details omitted.
```

```
    public void methodX() {
        Professor p = new Professor();

        // Exception handling is performed here.
        try {
            p.methodY();
        }
        catch (NullPointerException e) { ... }
    }
}
```

then the exception is contained by `methodX` in the `Student` class, and the `MainProgram` will thus be unaware that such an exception was ever thrown. This is illustrated in terms of the call stack in Figure 13-21.

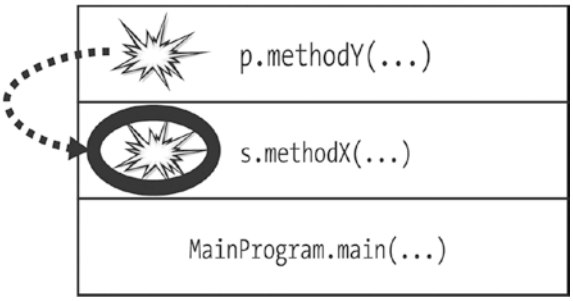


Figure 13-21. *The exception "escapes" the `methodY` level of the call stack, but is contained/handled by `methodX`*

Now, let's assume that ***neither*** `methodY` of `Professor` ***nor*** `methodX` of `Student` handles `NullPointerExceptions`, but that our `main` method is written to do so, as shown in the following:

```
public class Professor {
    // Details omitted.
    public void methodY() {
        // A NullPointerException is thrown here, but
        // is NOT caught/handled.
        // (Details omitted.)
    }
}
```

```
//-----------------------------

public class Student {
    // Details omitted.
    public void methodX() {
        Professor p = new Professor();

        // We're not doing any exception handling
        // here, either.
        p.methodY();
    }
}

//-----------------------------

public class MainProgram {
    public static void main(String[] args) {
        Student s = new Student();

        // Exception handling introduced here.
        try {
            s.methodX();
        }
        catch (NullPointerException e) { ... }
    }
}
```

In this case, a `NullPointerException` thrown in `methodY` would make its way through the call stack to the `main` method, where it would be contained as shown in Figure 13-22. In this case, the ***user*** of this application is unaware that an exception has occurred.

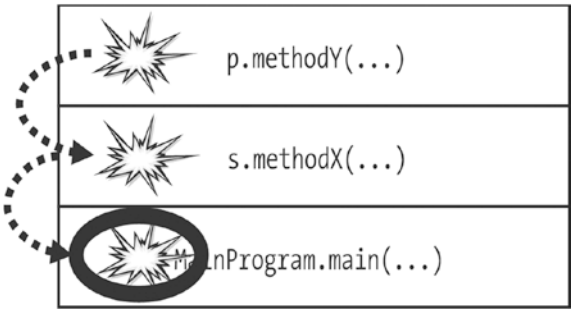


Figure 13-22. *The* `NullPointerException` *makes its way to the* `main` *method*

As a final variation, let's assume that even the `main` method omits explicit exception handling. If a `NullPointerException` were to arise in `methodY`, we'd see the following **stack trace** appear in the command window:

```
java PSExample
Exception in thread "main" java.lang.NullPointerException
    at Professor.methodY(Professor.java: ...)
    at Student.methodX(Student.java: ...)
    at MainProgram.main(MainProgram.java: ...)
```

From the perspective of the call stack, the situation is as shown in Figure 13-23.

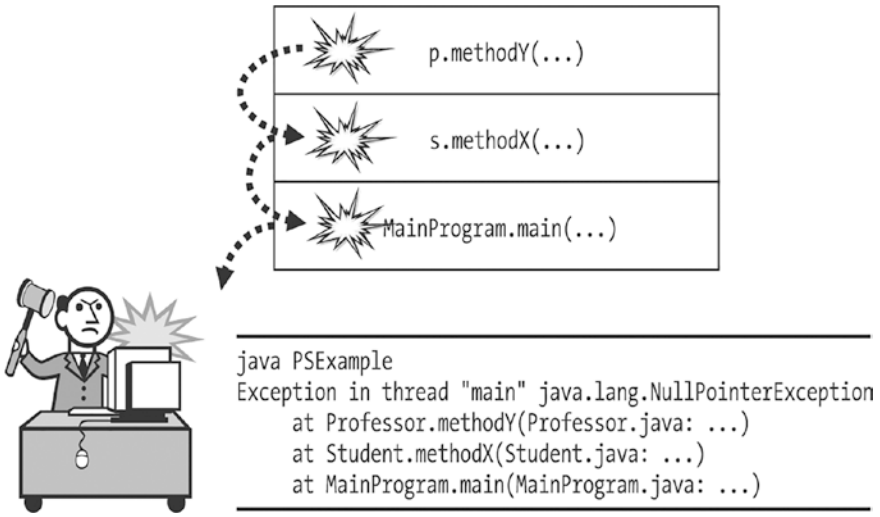


Figure 13-23. *If our application as a whole omits exception handling, the JVM terminates the application and reports the exception to the command window for the user to observe*

We'll learn a bit later in the chapter that the Java compiler will force us to catch selected types of exception, but not `NullPointerExceptions` in particular.

Interpreting Exception Stack Traces

As we saw previously, when an exception arises that we haven't properly handled, a **stack trace**–that is, a report by the JVM of where things went wrong and how we got there–is displayed in the command window. Interpreting an exception stack trace is quite easy; let's use a code example that would generate such a trace to illustrate how this is done.

Please read the inline comments carefully in the following three-class example to see where a potential `NullPointerException` problem is lurking!

```
public class Professor {
    // Our simplistic Professor class has only one attribute.
    private Student advisee;

    public void setAdvisee(Student s) {
        advisee = s;
```

```
    }

    public void printAdviseeInfo() {
        // A bit of delegation.
        advisee.print();
    }
}

// --------------------------------

public class Student {
    // Our Student class has only one attribute, and no way to set
    its value.
    // Thus, the name attribute of a Student will be initialized to
    the value
    // null, and will REMAIN null.
    private String name;

    public void print() {
        // Since the name attribute is guaranteed to be null here,
        // this line will generate a NullPointerException at run time.
        if (name.length() > 5) System.out.println(name);
    }
}

// --------------------------------

public class PSExample {
    public static void main(String[] args) {
        Student s = new Student();
        Professor p = new Professor();

        // Link the objects together.
        p.setAdvisee(s);

        // This next line of code will in turn invoke the print
        // method of p's Student advisee, which as we saw above is
        // going to generate a NullPointerException at run time.
```

```
        p.printAdviseeInfo();
    }
}
```

When we run this program, we'd see the following stack trace:

```
java PSExample
Exception in thread "main" java.lang.NullPointerException
    at Student.print(Student.java:10)
    at Professor.printAdviseeInfo(Professor.java:11)
    at PSExample.main(PSExample.java:12)
```

Reading the stack trace from top to bottom

- The actual `NullPointerException` arose on line 10 of the `Student` class:

  ```
  if (name.length() > 5) System.out.println(name); // line 10
  of Student
  ```

- ***That*** line of code is within the body of the `print` method of the `Student` class, which was invoked from line 11 of the `Professor` class:

  ```
  advisee.print(); // line 11 of Professor
  ```

- And ***that*** line of code is within the body of the `printAdviseeInfo` method of the `Professor` class, which was in turn invoked from the `PSExample` class's `main` method on line 12:

  ```
  p.printAdviseeInfo(); // line 12 of PSExample
  ```

Whenever a stack trace arises from an exception, the ***first*** place we should look to diagnose and repair the problem is the line of code that is reported as the ***first*** item in the stack trace: line 10 of the `Student` class, in this particular example. If in inspecting this line we cannot understand why the exception arises, we look to the second item in the stack trace, then the third, etc., until we have looked far enough back in the call history to determine why things went awry.

The Exception Class Hierarchy

As mentioned in passing earlier in the chapter, the generic `Exception` class, included in the `java.lang` package, is the superclass of all exception types in Java. As illustrated in Figure 13-24 (taken from Oracle's online Java documentation), there are ***many*** direct subclasses of the `Exception` class, and these continue to change with new versions of the Java language.

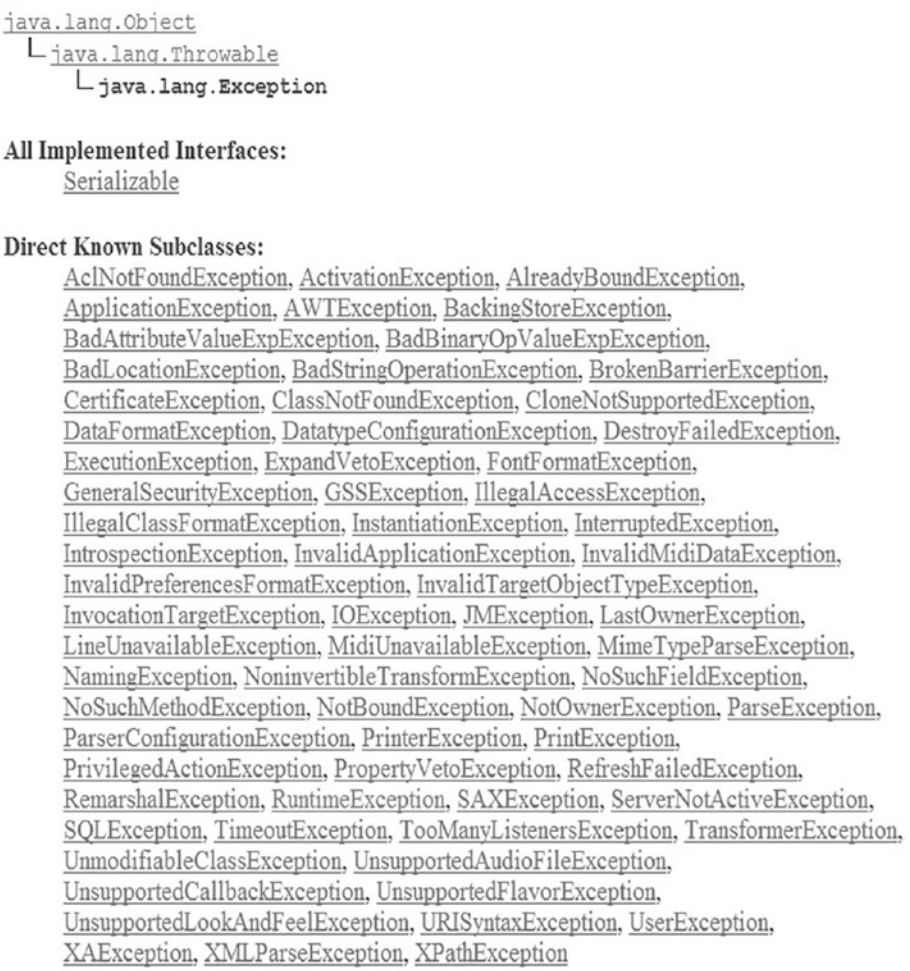


Figure 13-24. *The* `java.lang.Exception` *class has* ***many*** *"offspring"!*

A `catch` clause for a given exception type *X* will catch that specific type of exception ***or any of its subtypes***, by virtue of the "is a" nature of inheritance. For this reason, it's important to list `catch` clauses in most specific to least specific order after a `try` block; that is, from lowest-level subclass to superclass. Let's use a specific example to illustrate why this is so.

For our example, we'll perform a database access operation (as pseudocode); operations against databases, including possible exceptions that result from such operations, are governed by classes belonging to the `java.sql` package. We'll deal in this example with three types of successively more general exceptions:

- `DataTruncation` is the most specific type of exception that we'll concern ourselves with in this example. As its name implies, a `DataTruncation` exception occurs if data that is being written to a database happens to get ***truncated***, as, for example, when a particularly long `String` value is written to a database field that can only accommodate 255 characters.
- `DataTruncation` exceptions are a special case/subtype of the more general `SQLWarning` exception type. `SQLWarning` exceptions are thrown whenever any sort of database access problem occurs that, while not fatal, is significant enough to warrant alerting an application about.
- `SQLWarning` exceptions, in turn, are a special case/subclass of the more general `SQLException` class of exceptions. `SQLExceptions` are thrown when anything at all of concern arises in the course of interacting with a database, ranging from attempting to log on using an invalid password to trying to access a nonexistent table to attempting to submit a poorly formed SQL query.
- Finally, `SQLException` is a direct subclass of the generic `Exception` class.

This inheritance "lineage" is illustrated in Figure 13-25.

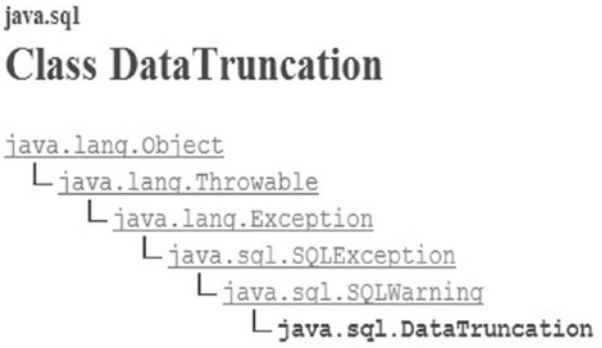


Figure 13-25. *The inheritance "lineage" of the* `java.sql.DataTruncation` *exception type*

The following code snippet presents one way for us to write the `try/catch` code for handling `DataTruncation` (and other sorts of database-related) exceptions; note that we'd need to include the appropriate `import` directive(s) to either import all of the `java.sql` classes or just those that we're referencing in our code:

```
try {
    // Pseudocode.
    attempt to write data to a database
}
catch (DataTruncation e1) {
    // Catch the most specific exception type first;
    // details omitted ...
}
catch (SQLWarning e2) {
    // ... then, the next most specific ...
    // Details omitted.
}
catch (SQLException e3) {
    // ... working our way up to the most general.
    // Details omitted.
}
```

Since a `DataTruncation` exception is a type of `SQLException` by virtue of inheritance, we could alternatively choose to write this code with a ***single*** `catch` clause, as follows:

```
try {
    // Pseudocode.
    database access operation ...
}
catch (SQLException e) {
    // This will catch DataTruncation exceptions along with
    // all other (sub)types of SQLException.
    // Details omitted.
}
```

Why would we ever bother to use ***three*** `catch` clauses when one will suffice? Because the more specific an exception we catch, the more specific our recovery code can be:

```
try {
    // Pseudocode.
    database access operations that are liable to throw, among other
    types of exceptions, DataTruncation exceptions ...
}
catch (DataTruncation e1) {
    // Pseudocode.
    respond SPECIFICALLY to data truncation issues ...
}
catch (SQLWarning e2) { ... }
catch (SQLException e3) { ... }
```

The following alternative version would ***not*** be appropriate because the three catch clauses are listed in least specific to most specific order; thus, the ***first*** `catch` clause would always catch all forms of `SQLException`, and the latter two `catch` clauses would never be reached:

```
try {
    // Pseudocode.
    database access operation ...
}
catch (SQLException e1) {
    // This catch clause will catch any/all SQLExceptions, including
    // SQLWarnings and DataTruncations ... details omitted.
```

```
}
catch (SQLWarning e2) {
    // This catch clause is wasted - it can never be reached!
}
catch (DataTruncation e3) {
    // This catch clause is also wasted - it can never be reached!
 }
```

Catching the Generic Exception Type

Some programmers use the "lazy" approach of catching the most generic `Exception` type and then doing ***nothing*** to recover, just to silence the compiler:

```
try {
    // Pseudocode.
    do anything!!!
}
catch (Exception e) { } // Empty braces => do NOTHING to recover!!!
```

This is not a good practice! By doing so, we're masking the fact that an exception has occurred: our program may be in a serious state of dysfunction, perhaps coming to a screeching halt (!), but will remain ***silent*** as to why because the `catch` clause shown previously suppresses the typical stack trace that would otherwise typically be displayed.

This is not to say that we should never catch generic `Exceptions`; one legitimate case where we might wish to do so is if we are writing a special-purpose error handling subsystem for an application, as suggested by the following pseudocode:

```
public class Example {
    public static void main(String[] args) {
        try {
            // Pseudocode.
            all of our main application logic is here ...
        }
        catch (Exception e) {
            // Invoke a static method on a custom
            // MyExceptionHandler class that we've written.
            MyExceptionHandler.handleException(e);
```

```
        }
    }
}
```

Here we assume that the static `handleException` method of a custom `MyExceptionHandler` class that we've developed encapsulates logic for handling all exceptions generated throughout our application in a central, consistent fashion (perhaps by recording them in an application log file or alerting a system administrator to the issue in real time).

Compiler Enforcement of Exception Handling

Generally speaking, the Java compiler will force us to enclose code that is liable to throw an exception in a `try` block with an appropriate `catch` block(s). For example, if we were attempting to read data from a file, as we'll do in earnest in Chapter 15

```
public class FileIOExample {
    public static void main(String[] args) {
        // Pseudocode.
        open the file of interest;
        while (end of file not yet reached) {
            read next line from file;
            // etc.
        }
    }
}
```

we'd get compiler errors complaining that our attempts to open a file, read from a file, etc. must be handled:

```
Unreported exception java.io.FileNotFoundException; must be caught or
declared to be thrown
```

In such a situation, we have two choices:

- Ideally, we'd enclose the code in question in a `try` block with an appropriate `catch` block(s):

  ```
  import java.io.FileNotFoundException;
  ```

```
public class FileIOExample {
    public static void main(String[] args) {
      try {
        // Pseudocode.
        open the file of interest;
        while (end of file not yet reached) {
            read next line from file;
            // etc.
        }
      }
      catch (FileNotFoundException e) { ... }
      // etc.
```

- Alternatively, we may add a throws clause to the header of the method in which the uncaught exception might arise, as illustrated in the following:

```
import java.io.FileNotFoundException;

public class FileIOExample {
    // By adding the throws clause to the main method declaration,
    we avoid
    // having to worry about catching FileNotFoundExceptions.
    public static void main(String[] args) throws
    FileNotFoundException {
        // Pseudocode.
        open the file of interest;
        while (end of file not yet reached) {
            read next line from file;
            // etc.
        }
    }
}
```

The only types of exception that the compiler doesn't mandate catching are those derived from the `RuntimeException` class, which is a direct subclass of the `Exception` class. Several of the more commonly encountered `RuntimeException` types are `NullPointerException`,

`ArithmeticException` (which arises when we attempt an illegal arithmetic operation, such as dividing by zero), and `ClassCastException` (which, as we discussed in Chapter 7, arises if we try to incorrectly cast an object reference to an inappropriate type). Typically, these types of exception represent design flaws that we should "bulletproof" our application against as we are building it, before it ever goes into production.

Taking Advantage of the Exception That We've Caught

Note that the declaration of a `catch` clause looks somewhat like a method header declaration in that we declare a parameter of type `Exception` (or one of its subtypes) to be passed in as an argument to the `catch` block:

```
catch (SomeExceptionType variableName) { ... }
```

However, we don't explicitly invoke a `catch` block from our program the way we directly invoke a method. Instead, as mentioned earlier, the JVM automatically transfers control to a `catch` block if an exception arises at run time; as it does so, the JVM also passes the `catch` block a reference to an object representing the type of exception that has occurred. We can therefore invoke methods on the exception object to help in diagnosing the problem, for example:

- We can invoke the `String getMessage()` method on the exception object to obtain a text message describing the problem that has arisen:

```
try {
    // Pseudocode.
    try to open a non-existent file named Foo.dat ...
}
catch (FileNotFoundException e) {
    System.out.println("Error opening file " + e.getMessage());
}
```

Output:

```
Error opening file Foo.dat (The system cannot find the file specified)
```

- We can invoke the `void printStackTrace()` method to display a traditional stack trace in the command window (recall that stack traces only occur ***automatically*** if we ***don't*** handle an exception):

```
try {
    // Pseudocode.
    try to open a non-existent file named Foo.dat ...
}
catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Nesting of Try/Catch Blocks

A `try` statement may be nested inside either the `try` or `catch` block of another `try` statement. We frequently have a need to nest an inner `try` within an outer `catch` block, in particular, because the recovery code that we write within a `catch` block may pose a risk of throwing additional exceptions of its own.

As an example, consider the following code snippet. We are attempting to open a user-specified file and so must stand ready to catch `FileNotFoundExceptions`:

```
try {
    // Pseudocode.
    open a user specified file
}
catch (FileNotFoundException e) {
    // Pseudocode.
    recovery code goes here ...
}
```

Our recovery plan, should a `FileNotFoundException` arise, is to open a default file instead; but, since the attempt to open a default file can also potentially throw a `FileNotFoundException` of its own, we must wrap the recovery code in its ***own*** nested `try` block, as illustrated in the following:

```
try {
    // Pseudocode.
```

```
    open a user specified file
}
catch (FileNotFoundException e) {
    // If we were unable to find the user-specified file, perhaps
    our way of
    // recovering will be to open a DEFAULT file ... but, what if
    the DEFAULT
    // file cannot be found, either?
    try {
        // Pseudocode.
        open a DEFAULT file instead ...
    }
    catch (FileNotFoundException e2) {
        // Pseudocode.
        attempt to recover ...
    }
}
```

User-Defined Exception Types

If we think of exceptions as signal flares that are thrown by the JVM to report issues, it's natural to want to extend this notion so as to allow our application to do the same: that is, to perhaps define our own ***custom*** exception types to signal various application-specific types of errors and to then ***programmatically*** throw them as needed when something goes awry. This is indeed possible in Java; let's explore the basics of declaring and using user-defined exception types.

To invent a custom exception type called `MissingValueException`, we harness the power of inheritance to extend one of the predefined Java exception classes; frequently, we'll extend the generic `Exception` class directly:

```
public class MissingValueException extends Exception { ... }
```

Of course, we'd store this class definition in a `.java` file–`MissingValueException.java` in this particular case–and compile it into a bytecode file named `MissingValueException.class`.

As to what we program in the body of our custom exception, we have several options. The simplest of all user-defined exceptions is a class with an ***empty body***:

```
public class MissingValueException extends Exception { }
```

As trivial as this class is, we've nonetheless accomplished an important goal: namely, we've defined a new exception type, `MissingValueException`, which can be explicitly thrown and caught by our application. We'll illustrate how this is done shortly, but before we do so, let's improve upon our custom exception class design a bit by introducing a constructor as shown in the following:

```
public class MissingValueException extends Exception {
    // We've added a constructor ...
    public MissingValueException(String message) {
        // ... which simply invokes the base class constructor.
        super(message);
    }
}
```

Why might we want to do this?

- Recall that the generic `Exception` class declares a method with the header `String getMessage()` that can be used to extract an informative message from an exception when we catch it.
- The text of this message is originally fed into an `Exception` instance as an argument via the constructor `public Exception(String message)`.

The `getMessage` method will of course be inherited by the `MissingValueException` class. However, as we discussed previously, constructors are not inherited, and so if we want to initialize the message of a `MissingValueException` instance in a similar fashion, we must explicitly declare such a constructor for our `MissingValueException` class; we may then reuse the code of the Exception superclass constructor via the syntax `super(message);`

Let's make one final enhancement to our `MissingValueException` class. As a subclass, we are free to add whatever features make sense above and beyond those that are inherited from the superclass; so we'll add one attribute, `Student student`, plus a `get` method for this attribute. The purpose for doing so will become clear in a moment.

We'll also modify the constructor header to accommodate ***two*** parameters–a `String` message and a `Student`–as shown in the following:

```
public class MissingValueException extends Exception {
    // We've added an attribute ...
    private Student student;

    // ... modified our constructor to take two arguments so
    // that we can pass in a value for both the Student attribute
    // and the message that this exception will carry with it ...
    public MissingValueException(Student s, String message) {
        super(message);
        student = s;
    }

    // ... and added a get method for our Student attribute.
    public Student getStudent() {
        return student;
    }
}
```

This is about as elaborate as a custom exception needs to be in order to give us maximum versatility.

Let's now look at how we'd go about using our new exception type to signal an error condition back to client code. First, let's put this exception type to work in the `Student` class. We'll present the code for the `Student` class in its entirety first and then narrate it in detail afterward:

```
public class Student {
    private String name;
    private String ssn;
    // Other details omitted.

    // Accessor methods.

    public String getSsn() { ... }
    public void setSsn() { ... }
    public String getName() { ... }
```

```
    public void setName(String n) throws MissingValueException {
        // We want to report an error if the String that has been
        // passed in is blank.
        if (n.equals("")) {
            throw new MissingValueException(this,
                "A student's name cannot be blank");
        }
        else {
            name = n;
        }
    }

    // etc.
}
```

The first bit of unusual syntax that we notice in the preceding code is in the method header declaration for the `setName` method:

```
public void setName(String n) throws MissingValueException {
```

Since we have the potential to throw a `MissingValueException` from within this method, we must declare that the `setName` method `throws MissingValueException`.

Then, inside the body of the `setName` method, we wish to throw a `MissingValueException` if we detect that client code has passed in an empty `String` (`""`) as the proposed name for this `Student`. We do so via the syntax

```
throw new MissingValueException(this, "A student's name cannot be blank");
```

That is, the keyword `throw` is followed by the `new` keyword, which in turn is followed by a call to our `MissingValueException` constructor. Thus, we've just "shot off a signal flare," carrying both a reference to the `Student` object in question and an informative message as to ***why*** we've shot off the flare: namely, because "A student's name cannot be blank."

Now, let's see how client code ***reacts*** to this "signal flare." If we were to attempt to write client code as follows

```
public class Example {
    public static void main(String[] args) {
        // Pseudocode.
```

```
String name = read value from GUI;
Student s = new Student();
s.setName(name);
// etc.
```

the following compiler error would arise on the line that attempts to invoke the `setName` method of `Student s`:

```
Unreported exception MissingValueException; must be caught or declared to
be thrown
    s.setName(name);
```

Because we included the `throws MissingValueException` clause on our `setName` method declaration in the `Student` class, we are ***forced*** by the Java compiler to catch this type of exception in our client code! That is, we must place our attempt to invoke the `setName` method on `s` in a `try` block and follow it with an appropriate `catch` block, as shown in the following:

```
public class Example {
    public static void main(String[] args) {
        // Pseudocode.
        String name = read value from GUI;
        Student s = new Student();
        try {
            s.setName(name);
        }
        catch (MissingValueException e) {
            System.out.println(e.getMessage());
            System.out.println("ID of affected student:  " +
                e.getStudent().getSsn());
        }
        // etc.
```

In our `catch` block, we've taken advantage of both the inherited `getMessage` method and the custom `getStudent` method of the `MissingValueException` class to produce output as follows:

```
A student's name cannot be blank
ID of affected student:  123-45-6789
```

Throwing Multiple Types of Exception

Note that we can throw more than one type of exception from the same method. In the following example, it's assumed that we've declared a custom `InvalidCharacterException` in addition to `MissingValueException`:

```
public class Student {
    // Details omitted.

    public void setName(String s) throws MissingValueException,
        InvalidCharacterException {
        if (s.equals("")) {
            throw new MissingValueException(this,
                "A student's name cannot be blank");
        }
        // Pseudocode.
        else if (s contains an invalid character) {
            throw new InvalidCharacterException(this, s +
                " contains a non-alphabetic character");
        }
        else name = s;
    }

    // etc.
}
```

Enum(eration)s

Situations often arise in an application where we wish to constrain the value that a variable can assume to a finite set of valid choices. For example, let's say that SRS University offers degrees in the following five major fields:

- Mathematics

- Biology
- Chemistry
- Computer Science
- Physical Education

We design our `Student` class so that one of its attributes, `String major`, reflects the discipline that a given student is majoring in. Then, to ensure that client code doesn't pass in an inappropriate value for the student's major field of study, we invent a custom exception called `InvalidMajorException` and use it to signal issues in this regard, as illustrated in the following:

```
public class Student {
    private String name;
    private String major;
    // etc.

    // Constructor.
    public Student(String name, String major) throws
    InvalidMajorException {
        this.setName(name);

        // Pseudocode.
        if (major not one of the five approved majors) {
            throw new InvalidMajorException();
        }
        else {
            this.setMajor(major);
        }
    }

    // Accessor methods.

    public void setName(String n) {
        name = n;
    }

    public void setMajor(String m) {
        major = m;
```

```
    }

    // etc.
}
```

In so doing, we cannot prevent or detect errors in client code at ***compile*** time

```
// Client code.

// The compiler forces us to place Student constructor calls in
// an appropriate try block ...
try {
    // ... but we nonetheless have WRITTEN code that is
    // guaranteed to throw a runtime exception, because the
    // COMPILER has no way to determine the error.
    Student s = new Student("Dorothy Jost", "Culinary Arts");
}
catch (InvalidMajorException e) { ... }
```

because the `InvalidMajorException` doesn't arise until this code is ***executed***. However, we ***know*** that there are five, and ***only*** five, valid `String` values for a student's major… Is there perhaps a way to constrain the `major` attribute so that we can prevent misassignment at ***compile time***? The answer is ***yes!*** We use a construct called an **enum**–short for "enumeration"–to define/***enumerate*** a finite set of values that a given variable may assume.

Along with `classes` and `interfaces`, an `enum` is another form of user-defined type in Java that lives in its own `.java` source code file and is compiled into bytecode:

```
// EnumName.java

public enum EnumName { ... }
```

More specifically, an `enum` is a very simplistic sort of class, consisting of only

- A single attribute, named `value`, declared to be of whatever (primitive or reference) type we wish for it to be:

```
// A single attribute.
// (Pseudocode.)
private final type value;
```

Note that `value` is declared to be both `private` and `final`; as we learned in Chapter 7, the keyword `final` indicates that an `enum` instance's `value` attribute may only be assigned a value ***once*** in its "lifetime," after which the value cannot be changed.

- A list (enumeration) of values that the `value` attribute is permitted to assume, represented as a comma-separated list of ***symbolic name–value pairs***:

```
// Comma separated list of name-value pairs.
// (Pseudocode.)
symbolicName1(value1),
symbolicName2(value2),
<...>
symbolicNameN(valueN);
```

- A simple constructor, used to initialize the `value` attribute:

```
// Constructor.
// (Pseudocode.)
EnumName(type v) {
    value = v;
}
```

Note that an enum's constructor ***isn't*** declared to be `public`, because it is not invoked from client code; rather, it is used internally to the `enum` (in declaring the preceding list of symbolic name–value pairs).

- A single accessor method, `value()`, used by client code to retrieve the value of the `enum`'s lone attribute.

Putting this all together, the general template for declaring an `enum` is as follows, where the *italicized* items are those that we can customize:

```
public enum EnumName {
    // Comma separated list of name-value pairs, with last one ending with
       a semicolon (;).
    // (Pseudocode.)
    symbolicName1(value1),
    symbolicName2(value2),
```

```
    <...>
    symbolicNameN(valueN);

    // A single attribute.
    // (Pseudocode.)
    private final type value;

    // A (non-public) constructor.
    // (Pseudocode.)
    EnumName(type v) {
        value = v;
    }

    // Accessor method.
    // (Pseudocode.)
    public type value() {
        return value;
    }
}
```

Let's go back to our example involving students' majors and retrofit an `enum` called `Major` to control the assignment of valid major fields ***at compile time***.

- Behind the scenes, the five values that `Major` can assume will be declared as `String`s:

```
// Major.java

public enum Major {
    // Comma separated list of symbolic name-value pairs,
    where the
    // values (enclosed in parentheses) are all String literals.
    Math("Mathematics"),
    Bio("Biology"),
    Chem("Chemistry"),
    CS("Computer Science"),
    PhysEd("Physical Education");
```

- Hence, the type of the `value` attribute, the type of the parameter passed to the `Major` constructor, and the return type of the `value()` method are all declared to be `String` to match:

```
    // We declare the value attribute to be of type String.
    private final String value;

    // The constructor takes a String argument.
    Major(String v) {
        value = v;
    }

    // The accessor method has a return type of String.
    public String value() {
        return value;
    }
}
```

Now, let's re-engineer our `Student` class to take advantage of the new `Major enum`/type. In particular, we'll

- Change the declaration of the `Student` class's `major` attribute to be of type `Major` instead of `String`.
- Change the type of the second parameter that we're passing into our `Student` constructor accordingly.
- Eliminate our use of the `InvalidMajorException`.
- Change the `setMajor` method to accept an argument of type `Major` vs. `String`.

And, while we're at it, let's add a `display` method for use in testing our improvements to the `Student` class. The "new and improved" `Student` class code is thus as follows (changes are **highlighted**):

```
public class Student {
    private String name;
    // We've changed the type of this attribute from String to Major.
    private Major major;

    // Other details omitted.
```

```
    // Constructor.
    // We've changed the type of the second constructor parameter from
    // String to Major, and have eliminated the "throws
    // InvalidMajorException" clause.
    public Student(String name, Major major) {
        this.setName(name);
        this.setMajor(major);
    }

    // Accessor methods.

    public void setName(String n) {
        name = n;
    }

    // We've changed the type of the parameter on this method
    // from String to Major.
    public void setMajor(Major m) {
        major = m;
    }

    // etc.

    // We've added a display method.
    public void display() {
        // Note that we are taking advantage of the enum's value()
        // method in the print statement below.
        System.out.println(name + " is a " + major.value() + " major.");
    }
}
```

Let's now demonstrate how our newly designed `Student` class might be utilized from client code:

```
public class EnumExample {
    public static void main(String[] args) {
        // Instantiate a Student, using our newly-created Major enum to
        // assign one of the five valid values for a student's major
```

```
        // (note the syntax -- Major.CS -- for referring to such a value).
        Student s = new Student("Fred Schnurd", Major.CS);
        s.display();
    }
}
```

When executed, this program would produce the following output

```
Fred Schnurd is a Computer Science major.
```

where the symbolic name `Major.CS` has been translated to its behind-the-scenes `String` equivalent, `"Computer Science"`, courtesy of our call to the `value()` method from within the `Student`'s `display` method. Had we instead written the `display` method as follows

```
    public void display() {
        // We've dropped the call to major.value(), and are printing
        // major directly instead.
        System.out.println(name + " is a " + major + " major.");
    }
```

then the symbolic name of the `enum` would be printed instead:

```
Fred Schnurd is a CS major.
```

It is now ***physically impossible*** to assign any value as a `Student`'s major other than one of the five ***enum-approved values*** `Major.Math`, `Major.Bio`, `Major.Chem`, `Major.CS`, and `Major.PhysEd`; anything else simply won't compile.

- The following line of code won't compile–a `String` is not a `Major`:

  ```
  Student s = new Student("Fred Schnurd", "Basketweaving");
  ```

Compiler error:

```
Student.java: cannot find symbol
symbol  : constructor Student(java.lang.String,java.lang.String)
location: class Student
    Student s = new Student("Fred Schnurd", "Basketweaving");
```

- This next line won't compile either–our `Major enum` doesn't define `Basketweaving` as a valid value:

  ```
  Student s = new Student("Fred Schnurd", Major.Basketweaving);
  ```

Compiler error:

```
Student.java: cannot find symbol
symbol  : variable Basketweaving
location: class Major
    Student s = new Student("Fred Schnurd", Major.Basketweaving);
                                            ^
```

Here's another example of an `enum` called `Grade` that renders a `double` value:

```
public enum Grade {
    // Enumerate the values that a Grade can assume.
    // The values are all double constants in this case.
    A(4.0),
    B(3.0),
    C(2.0),
    D(1.0),
    F(0.0);

    // Our value attribute is declared to be of type double.
    private final double value;

    Grade(double v) { // double parameter type
        value = v;
    }

    public double value() { // double return type
        return value;
    }
}
```

Here's a bit of client code to illustrate its use:

```
public class GradeDemo {
```

```
    public static void main(String[] args) {
        // Declare a variable of type Grade.
        Grade grade;

        // We only may assign one of the "approved" values.
        grade = Grade.A;

        // Display it symbolically ...
        System.out.println(grade);

        //... and display its equivalent value as a double.
        System.out.println(grade.value());
    }
}
```

Output:

```
A
4.0
```

Here's a final example of an `enum` called `StudentBody` that uses a reference type, `Student`, as the encapsulated `enum` type:

```
public enum StudentBody {
        // Assign symbolic names to actual Student instances.
    fred(new Student("Fred")),
    mary(new Student("Mary"));

    private final Student value;

    StudentBody(Student value) {
        this.value = value;
    }

    public Student value() {
        return value;
    }
}
```

Here's a bit of client code to illustrate its use:

```
public class EnumExample {
    public static void main(String[] args) {
        Student x = StudentBody.fred;
        // etc.
    }
}
```

Enumerations are a powerful feature of the Java language; be certain to take advantage of them in designing your applications, as we will in building the SRS.

Providing Input to Command-Line-Driven Programs

Virtually all applications require input of some sort, in the form of either

- Data to be processed by the application
- Configuration information to control the manner in which the application operates

With a classic information system like the SRS, such input is typically acquired in one of two ways:

- From users, through their interactions with an application's graphical user interface
- By retrieving information from persistent storage–a file or database

On occasion, we also have a need to write command-line-driven utility programs in Java; it's handy to know how to guide such programs' behavior in one of two additional ways:

- Through the use of command-line arguments when the program is first launched
- By prompting the user for textual inputs via the command window

Let's explore both of these latter techniques.

Accepting Command-Line Arguments: The args Array

We learned in Chapter 2 that to invoke a Java program from the command line, we type the command `java` (to launch the JVM) followed by the name of the class/bytecode file containing the official `main` method, for example:

```
java Simple
```

We've also learned that it's possible to control certain aspects of the JVM's behavior through the use of command-line options; such flags go between the `java` command and the class name. For example, we learned earlier in this chapter that we can establish the ***classpath*** of a program–that is, inform the JVM of where to search for bytecode files–by using the `-cp` option, as in

```
java -cp C:\Foo\A.jar;D:\Bar\B.jar Simple
```

We can also initialize our own programs by passing in command-line information of our own design when we invoke them. Such data comes ***after*** the name of the program on the command line:

```
java ClassFileName arg1 arg2 […] argN
```

For example:

```
java ComputeTotal 123 456 789
```

or

```
java AnalyzeWords –sort DOG GUPPY GIRAFFE HIPPOPOTAMUS CAT
```

Such so-called **command-line arguments** get passed to the `main` method of the Java program as a `String` array called `args` (or whatever else we wish to name it), as declared by the `main` method header:

```
public static void main(String[] args) { ...
```

Inside the `main` method, we can do with `args` whatever we'd do with any other array; for example, determine its length, access individual `String` items within the array, and so forth, as we discussed in Chapter 6.

Let's look at a few example programs that take advantage of command-line arguments. This first example is quite simple; it simply prints out information about the arguments that it has received:

```
public class FruitExample {
    public static void main(String[] args) {
        System.out.println("The args array contains " + args.length + "
        entries." );

        for (int i = 0; i < args.length; i++) {
            System.out.println("Argument #" + i + " = |" + args[i] + "|");
        }
    }
}
```

If we run the program as follows

```
java FruitExample apple banana cherry
```

we'd get the following output:

```
The args array contains 3 entries.
Argument #0 = |apple|
Argument #1 = |banana|
Argument #2 = |cherry|
```

If we want to provide arguments that contain spaces, we'd enclose them in double quotes, as follows:

```
java FruitExample "green apple" "yellow banana" "black cherry"
```

Output:

```
The args array contains 3 entries.
Argument #0 = |green apple|
Argument #1 = |yellow banana|
Argument #2 = |black cherry|
```

And if we were to run our program with no command-line arguments whatsoever

```
java FruitExample
```

it would report the following:

```
The args array contains 0 entries.
```

Introducing Custom Command-Line Flags to Control a Program's Behavior

Let's look at a second example that is a bit more elaborate; in this program, called `AnalyzeWords`, we introduce a custom command-line option of our own invention, `-sort`, to control the program's behavior.

- At a minimum, `AnalyzeWords` will inspect however many command-line arguments are provided by the user to determine the length of the shortest and longest of them.
- Optionally, based on the presence of the `-sort` option as a command-line argument, the program will also print out a list of the words in sorted order, eliminating duplicates.

We'll present the code in its entirety first, discussing it afterward:

```
import java.util.TreeSet;

public class AnalyzeWords {
    public static void main(String[] args) {
        // Let's start with a bit of error checking.
        // If the user forgot to provide command line input, let's
        // report this as an error.
        if (args.length == 0) {
            System.out.println("Usage:  java AnalyzeWords [-sort]
            list_of_words");
            System.out.println("e.g.:   java AnalyzeWords -sort ZEBRA " +
                "ELEPHANT RAT MONKEY");
            System.exit(0);
        }
```

```
// Initialize a few items.
boolean sort = false;
TreeSet<String> sortedWords = new TreeSet<>();
String shortest = null;
String longest = null;

for (int i = 0; i < args.length; i++) {
    // Watch for the presence of the -sort option.
    if (args[i].equals("-sort")) {
        sort = true;
        continue;
    }

    // If we haven't yet recorded a shortest or longest
    word, then by
    // definition this is the first, and hence both the shortest
    // and longest, word!
    if (shortest == null) {
        shortest = args[0];
        longest = args[0];
    }
    // Otherwise, compare this word to the shortest/longest
    seen so far.
    else {
        if (args[i].length() > longest.length()) longest = args[i];
        if (args[i].length() < shortest.length()) shortest =
       args[i];
    }

    // Add the word to the TreeSet so as to sort them
    automatically,
    // whether the user asked for them to be sorted or not; if the
    // user didn't ask for them to be sorted, we'll simply suppress
    // displaying this information.
    sortedWords.add(args[i]);
}
```

```
        if (sort) {
            System.out.println("Sorted words:");
            for (String s : sortedWords) {
                System.out.println("\t" + s);
            }
        }

        System.out.println("The shortest word was " + shortest.length() +
            " characters long.");
        System.out.println("The longest word was " + longest.length() +
            " characters long.");
    }
}
```

Let's now review noteworthy sections of the program in detail.

Because this program requires command-line input in order for it to do anything meaningful, we're going to check the length of (number of command-line arguments found within) the `args` array. If the length is 0, we're going to inform the user of how the program is to be used:

```
        // If the user forgot to provide command line input, let's
        // report this as an error.
        if (args.length == 0) {
            System.out.println("Usage:  java AnalyzeWords [-sort] list_of_
            words");
            System.out.println("e.g.:   java AnalyzeWords -sort ZEBRA " +
                "ELEPHANT RAT MONKEY");
```

Then we're going to terminate the program–there's no need to go any further if there's no input to process:

```
            System.exit(0);
        }
```

Next, we declare a few variables:

- A `boolean` flag, `sort`, that we'll set later in the program if we detect that the user has provided the optional `-sort` command-line argument:

```
// Keep track of whether the user wants to optionally sort
the words,
// in addition to analyzing them.
boolean sort = false;
```

- A `TreeSet` collection–recall from our discussion of collections in Chapter 6 that sets have the property of eliminating duplicates. `TreeSet`s, in particular, automatically sort their contents:

```
TreeSet<String> sortedWords = new TreeSet<>();
```

- Two `String`s that will maintain handles on the shortest and longest words that we've seen as of every new argument that we process:

```
// We'll keep track of the shortest and longest words as we go.
String shortest = null;
String longest = null;
```

We then iterate through the `args` array. If we discover the `-sort` option in the command-line input, we set our `boolean sort` flag to `true` and then use the `continue` statement to jump to the next word in the `args` array–we don't want to process the `-sort` option as a true "word" with respect to our analysis:

```
for (int i = 0; i < args.length; i++) {
    // Watch for the presence of the -sort flag.
    if (args[i].equals("-sort")) {
        sort = true;
        continue;
    }
```

If we haven't yet recorded any words as being either the shortest or longest, we'll record this (first) word as ***both*** the shortest and the longest that we've seen thus far:

```
    // If we haven't yet recorded a shortest or longest
    word, then by
    // definition this is the first, and hence both the shortest
    // and longest, word!
    if (shortest == null) {
        shortest = args[0];
```

```
            longest = args[0];
        }
```

Otherwise, if we have already assigned values to the `shortest` and `longest Strings`, we compare ***this*** word to each of them, to see if it is either longer than the longest or shorter than the shortest:

```
        else {
            // If the current word that we're processing is longer
            // than the longest that we've seen so far, remember IT
            // as the longest.
            if (args[i].length() > longest.length()) longest = args[i];

            // If the current word that we're processing is shorter
            // than the shortest that we've seen so far, remember IT
            // as the shortest.
            if (args[i].length() < shortest.length()) shortest =
            args[i];
        }
```

Then, ***whether or not the user has asked for us to sort the words***, we'll add this word to the `sortedWords TreeSet`. We do so for two primary reasons: (a) The `-sort` flag may appear ***later*** in the argument list–for example, `java AnalyzeWords DOG MONKEY ELEPHANT -sort`–and we want to be prepared to respond if it does. (b) It's so ***easy*** to sort the words, simply by adding them to a `TreeSet`, that there's no point in having to iterate through the `args` array a second time later on.

```
            // Add the word to the TreeSet so as to sort them
            automatically.
            // whether the user asked for them to be sorted or not; if the
            // user didn't ask for them to be sorted, we'll simply suppress
            // displaying this information.
            sortedWords.add(args[i]);
    }
```

We optionally display the sorted contents of the `TreeSet`:

```
    if (sort) {
        System.out.println("Sorted words:");
```

```
            for (String s : sortedWords) {
                System.out.println("\t" + s);
            }
        }
```

Finally, we display the results of our "shortest/longest" analysis:

```
        System.out.println("The shortest word was " + shortest.length() +
            " characters long.");
        System.out.println("The longest word was " + longest.length() +
            " characters long.");
    }
}
```

Let's look at a few different ways to invoke this program and the output that each would produce. First, we'll omit the `-sort` option:

```
java AnalyzeWords ZEBRA ELEPHANT RAT MONKEY
```

Output:

```
The shortest word was 3 characters long.
The longest word was 8 characters long.
```

Next, we'll include the `-sort` option:

```
java AnalyzeWords -sort ZEBRA ELEPHANT RAT MONKEY
```

Output:

```
Sorted words:
    ELEPHANT
    MONKEY
    RAT
    ZEBRA
The shortest word was 3 characters long.
The longest word was 8 characters long.
```

The same output results if the `-sort` option is at the ***end*** of the argument list:

```
java AnalyzeWords ZEBRA ELEPHANT RAT MONKEY -sort
```

Output:

```
Sorted words:
    ELEPHANT
    MONKEY
    RAT
    ZEBRA
The shortest word was 3 characters long.
The longest word was 8 characters long.
```

As we learned earlier, we must enclose multiword arguments in double quotes:

```
java AnalyzeWords -sort "LITTLE BO PEEP" RUMPELSTILTSKIN "EENY MEENY
MINEY MOE"
```

Output:

```
Sorted words:
    EENY MEENY MINEY MOE
    LITTLE BO PEEP
    RUMPELSTILTSKIN
The shortest word was 14 characters long.
The longest word was 20 characters long.
```

Accepting Keyboard Input: The Scanner Class

You learned in Part 1 of this book that Java provides a special `OutputStream` object called `System.out`, which in turn provides both `println` and `print` methods for displaying messages to the command-line window. Java also provides a special `InputStream` object

called System.in to read input from the command line as typed by a user. The java.util.Scanner class provides a convenient means of reading formatted data from input streams, such as System.in.

The Scanner class provides numerous overloaded constructors; the one that we'll utilize in our examples takes an InputStream as an argument, for example:

```
Scanner sc = new Scanner(System.in);  // reading from the keyboard
```

We then invoke various forms of next*Type*() method on the Scanner instance–nextBoolean(), nextInt(), nextDouble(), etc.–to read the next (white space–delimited) token from the input stream and automatically convert it to the type specified. The method next() reads and returns the token as a String.

In the example program that follows, we prompt the user to enter three values–a String, an int(eger), and a double, respectively. We provide exception handling logic for java.util.InputMismatchException, in case the user types the wrong sort of data for a given prompt:

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class ScannerExample {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        try {
                // Prompt the user for his/her first name.
                System.out.print("Please enter your FIRST name (only):  ");

                // Because we want to read the name as a String, we can use
                // the simple next() method.
                String name = sc.next();

                System.out.print("Please enter your age as an integer:  ");
                int age = sc.nextInt();

                System.out.print("Please enter your GPA as a double:  ");
                double gpa = sc.nextDouble();

                System.out.println();
                System.out.println(name + " is " + age + " years old.");
```

```
                System.out.println(name + "'s GPA is " + gpa + ".");
        }
        catch (InputMismatchException e) {
                System.out.println();
                System.out.println("Whoops!  You didn't follow the
                instructions " + "properly; please try again.");
        }
    }
}
```

Output:

```
Please enter your first name:  Herbie
Please enter your age as an integer:  32
Please enter your GPA as a double:  3.75

Herbie is 32 years old.
Herbie's GPA is 3.75.
```

Using Wrapper Classes for Input Conversion

Recall our introduction in Chapter 6 to Java's predefined ***wrapper classes*** for each of the eight distinct primitive Java types–`Integer`, `Float`, `Double`, `Byte`, `Short`, `Long`, `Boolean`, and `Character`–all within the core `java.lang` package. Our previous discussion of these classes centered on their use in "wrapping" primitive types for purposes of storing them in a Java collection. We're now going to explore an alternative use of these classes, as ***utility classes***, for performing data conversions.

One such example of when we might need to perform a data conversion operation is when obtaining data via a user interface. Whether we use a GUI to acquire user input or a command-line interface as we're exploring in this chapter, data is always acquired from a user in `String` format. We often need to convert this data into one of the Java numeric types (`int`, `double`, etc.) in order to be able to manipulate it mathematically.

Each of the wrapper classes declares a number of `static` methods that are useful when we need to perform data conversions. For example, the `Integer` class defines a `static` method with the header

```
static int parseInt(String s)
```

Pass in a `String` as an argument to this method, and the `Integer` class will convert it to an `int`(eger) for us if it represents a valid integer or will throw a `NumberFormatException` if it does not. Similarly, the `Double` class declares a `static parseDouble` method, the `Float` class declares a `static parseFloat` method, and so forth.

Here's a simple example to illustrate how the `Integer.parseInt` method might be used:

```
public class IntegerTest {
    public static void main(String[] args) {
        String[] ints = { "123", "foobar", "456", "789" };
        int i = 0;

        for (i = 0; i < ints.length; i++) {
            try {
                int test = Integer.parseInt(ints[i]);
                System.out.println(test + " converted just fine!");
            }
            catch (NumberFormatException e) {
                System.out.println("Whoops!  " + ints[i] +
                    " is an invalid integer.");
            }
        }
    }
}
```

This program, when run, produces the following output:

```
123 converted just fine!
Whoops!  foobar is an invalid integer.
456 converted just fine!
789 converted just fine!
```

Note that we inserted the `try` statement ***inside of*** the `for` loop, so that it would take effect once per loop iteration; if we had instead inserted the whole `for` loop into the `try` block, as illustrated in the following

```
public class IntegerTest {
    public static void main(String[] args) {
        String[] ints = { "123", "foobar", "456", "789" };
        int i = 0;

        try {
            // Now, the entire for loop is within the try block.
            for (i = 0; i < ints.length; i++) {
                int test = Integer.parseInt(ints[i]);
                System.out.println(test + " converted just fine!");
            }
        }

        catch (NumberFormatException e) {
            System.out.println("Whoops!  " + ints[i] + " is an invalid
            integer.");
        }
    }
}
```

then the first occurrence of a `NumberFormatException` would have ***terminated*** the `for` loop, producing the following alternate output:

```
123 converted just fine!
Whoops!  foobar is an invalid integer.
```

Another `static` method defined by the `Integer` class (and the other wrapper classes, as well) is used to do the reverse:

```
static String toString(int)
```

This method accepts an `int`(eger) value as an argument, converting it into a proper `String`, as the following example illustrates:

```
int i = 56;
```

```
String s = Integer.toString(i);     // s now has a String value of "56"
```

There is a shortcut way for doing the same thing, however: we simply need to concatenate the value of `i` to an empty `String` (`""`) as follows:

```
int i = 56;
String s = "" + i;   // s now has a String value of "56"
```

Features of the Object Class

We learned in Chapter 5 that the `Object` class, provided in the core `java.lang` package, is the ultimate superclass for all classes in Java, user-defined or otherwise. Thus, all Java objects, regardless of type, inherit a common set of interesting features from the `Object` class, which we'll explore in this section.

Determining the Class That an Object Belongs To

Every Java object inherits a method from the `Object` class with the header

```
Class getClass()
```

This method, when invoked on an object reference–for example, `x.getClass();`–returns a reference to an object of type `Class` representing an abstraction of the class that object `x` belongs to.

The `Class` class, in turn, defines a method with the header

```
String getName()
```

This method, when invoked on a `Class` reference, returns the ***fully qualified name*** of the class for classes that belong to a ***named*** package (*packagename.Classname*–for example, `java.util.ArrayList`) or a simple class name–for example, `Professor`–for classes that belong to the default (unnamed) package (as will be the case for our SRS classes).

Chaining these two methods together, we can ask any object reference to identify which class the object it references belongs to, as follows:

```
reference.getClass().getName();
```

For example:

```
Professor pr = new Professor();
System.out.println(pr.getClass().getName());
```

Output:

```
Professor
```

or

```
// Test to see if x is referring to a Professor object.
// Note that if x were null, we'd get a NullPointerException, so we've
// inserted a test for "nullness" before calling the getClass method on x.
if (x != null && x.getClass().getName().equals("Professor")) { ...
```

Returning to our discussion of exception handling from earlier in this chapter, we can also use the `getClass().getName()` approach within a `catch` block to determine what sort of exception has occurred:

```
try { ... }
catch (SomeExceptionType e) {
    System.out.println("Drat!  An exception of type " +
        e.getClass().getName() + " has occurred.");
}
```

Sample output:

```
Drat!  An exception of type java.lang.NullPointerException has occurred.
```

Another way to test whether a given object reference belongs to a particular class is via the `instanceof` operator. This is a `boolean` operator that allows us to determine if some reference variable *x* is referring to an object of class/type *Y*. Here's a simple code example to illustrate the use of this operator; in this example, we assume that `Person` is an abstract class and that both `Student` and `Professor` are concrete classes derived from `Person`:

```
Person x;

// Later in the program ...
x = new Professor();
```

```
// Still later in the program ...
// Determine the precise runtime identity of x.
if (x instanceof Student) {
    System.out.println("x is a Student");
}

if (x instanceof Professor) {
    System.out.println("x is a Professor");
}

if (x instanceof Person) {
    System.out.println("x is a Person");
}
```

Output:

```
x is a Professor
x is a Person
```

Testing the Equality of Objects

What does it mean to say that two objects are "equal"? When speaking of generic `Objects`, we say that two objects (or, more precisely, two object ***references***) are equal if they both refer to precisely the same object in memory (i.e., if the references both point to the same exact memory location in the JVM). Java provides two ways for determining the equality of two `Object` references `x` and `y`:

- The double equal sign operator (==), which we've seen used several times previously in this book, first to compare the values of simple data types in Chapter 2

  ```
  int x = 3;
  int y = 3;
  if (x == y) { ...
  ```

  and then to compare the identities of `String` references earlier in Chapter 13:

```
String x = "foo";
String y = "foo";
// Do x and y refer to the same String object?
if (x == y) { ...
```

- The `boolean equals` method, which is inherited by all objects from the `Object` class; we used this method with the `String` class earlier in this chapter.

As defined for `Objects` generically, the `==` operator and the `equals` method can be used interchangeably to test whether two references refer to exactly the same `Object`. Here's some example code to illustrate these two alternative approaches:

```
public class EqualsTest1 {
  public static void main(String[] args) {
    // We'll create one generic Object...
    Object o1 = new Object();

    // ... and maintain two handles on it (o1 and o2).
    Object o2 = o1;

    // We'll create a second different Object object, and will
    // use variable o3 to maintain a handle on it.
    Object o3 = new Object();
```

This is illustrated conceptually in Figure 13-26.

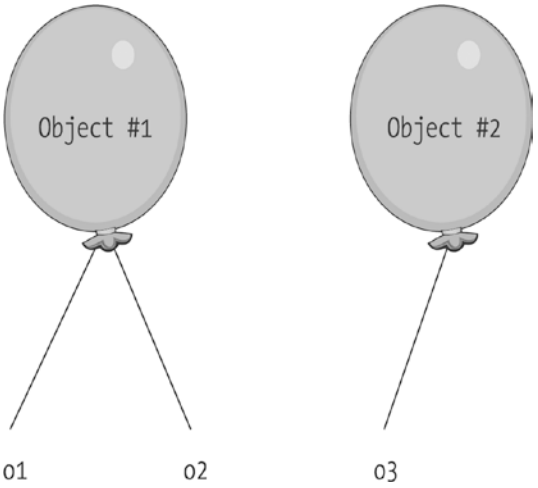


Figure 13-26. *We've created two generic* `Objects`

Then, if we execute the following code (note our use of nested `boolean` expressions within the `println` statements)

```
// Are o1 and o2 "equal"?
System.out.println("The expression o1 == o2 evaluates to: " +
(o1 == o2));
System.out.println("The expression o1.equals(o2) evaluates to: " +
    (o1.equals(o2)));

// Are o1 and o3 "equal"?
System.out.println("The expression o1 == o3 evaluates to: " +
(o1 == o3));
System.out.println("The expression o1.equals(o3) evaluates to: " +
    (o1.equals(o3)));
```

the following output results:

```
The expression o1 == o2 evaluates to: true
The expression o1.equals(o2) evaluates to: true
The expression o1 == o3 evaluates to: false
The expression o1.equals(o3) evaluates to: false
```

This is because o1 and o2 are truly referring to exactly the same Object, whereas o3 is referring to a different Object.

Let's repeat the example, this time using Person objects:

```
public class EqualsTest2 {
  public static void main(String[] args) {
    // We'll create one Person object ...
    Person p1 = new Person("222-22-2222", "Fred");

    // ... and maintain two handles on it (p1 and p2).
    Person p2 = p1;

    // We'll create a second different Person object with exactly the same
    // attribute values as the first Person object that we created,
    and will
    // use variable p3 to maintain a handle on this second object.
    Person p3 = new Person("222-22-2222", "Fred");
```

When we execute the following code

```
    // Are p1 and p2 "equal"?
    System.out.println("The expression p1 == p2 evaluates to: " +
    (p1 == p2));
    System.out.println("The expression p1.equals(p2) evaluates to: " +
        (p1.equals(p2)));

    // Are p1 and p3 "equal"?
    System.out.println("The expression p1 == p3 evaluates to: " +
    (p1 == p3));
    System.out.println("The expression p1.equals(p3) evaluates to: " +
        (p1.equals(p3)));
```

we get the same sort of result as we did for generic Objects:

```
The expression p1 == p2 evaluates to: true
The expression p1.equals(p2) evaluates to: true
The expression p1 == p3 evaluates to: false
The expression p1.equals(p3) evaluates to: false
```

Even though p1 and p3 are referring to Person objects with ***identical attribute values***, they are nonetheless physically distinct Person instances, and so p1 is not considered to be "equal to" p3.

Let's repeat this example one final time, this time using Strings. We'll formally instantiate the Strings to avoid the String literal pool, ensuring that we truly do create physically separate instances of String objects:

```
public class EqualsTest3 {
  public static void main(String[] args) {
    // We'll create one String object, using the formal method of
    // String instantiation ...
    String s1 = new String("hello");

    // ... and maintain two handles on it (s1 and s2).
    String s2 = s1;

    // We'll explicitly instantiate a second String object,
    // with EXACTLY THE SAME VALUE as the first String object -
    // "hello" -  and will use variable s3 to maintain a handle
    // on this second String.
    String s3 = new String("hello");
```

This is illustrated conceptually in Figure 13-27.

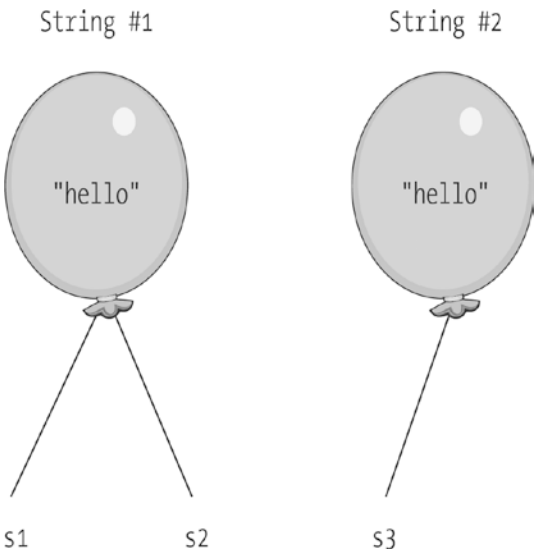


Figure 13-27. *We've created two physically distinct* `String` *instances with the same value, "*`hello`*"*

When we execute the following client code

```
// Are s1 and s2 "equal"?
System.out.println("The expression s1 == s2 evaluates to: " + (s1 == s2));
System.out.println("The expression s1.equals(s2) evaluates to: " + (s1.
equals(s2)));

// Are s1 and s3 "equal"?
System.out.println("The expression s1 == s3 evaluates to: " + (s1 == s3));
System.out.println("The expression s1.equals(s3) evaluates to: " + (s1.
equals(s3)));
```

the following output results:

```
The expression s1 == s2 evaluates to: true
The expression s1.equals(s2) evaluates to: true
The expression s1 == s3 evaluates to: false
The expression s1.equals(s3) evaluates to: true
```

Note that when comparing Strings with the equals method, the equals method behaves ***differently*** than it did for generic Objects and for Person objects: that is, s1 and s3 are deemed to be "equal" ***despite the fact that they refer to physically different*** **String** ***instances!*** Compare the preceding last line of output

```
The expression s1.equals(s3) evaluates to: true
```

with the corresponding lines of output from our Object and Person examples:

```
The expression o1.equals(o3) evaluates to: false
```

and:

```
The expression p1.equals(p3) evaluates to: false
```

How is it that the equals method behaves differently for Strings than it does for other object types? As it turns out, the String class ***overrides*** the equals method as defined by the Object class so that it compares String ***values*** rather than String ***identities***. In fact, many of the predefined Java classes have overridden the equals method as inherited from Object to perform a relevant, class-specific comparison, for example, the wrapper classes (Boolean, Integer, Double, etc.), the Date class, and others. And, of course, we can override the equals method for our own classes, as well; let's see how this is accomplished.

Overriding the equals Method

Let's say we want the equals method, when used to compare two Person instances, to deem them "equal" if they have the same value for their Social Security number (ssn) attribute. To accomplish this, we'd override the equals method as shown in the following; we'll present the code in its entirety first and then explain some of the details afterward:

```
public class Person {
    private String ssn;
    private String name;
    // etc.
```

```
// Constructor.
public Person(String s, String n) {
    this.setSsn(s);
    this.setName(n);
}

// Accessor methods.

public String getSsn() {
    return ssn;
}

// etc.

// Overriding the equals method that we inherited from the
Object class.
public boolean equals(Object o) {
    boolean isEqual;

    // Try to cast the Object reference into a Person reference.
    // If this fails, we'll get a ClassCastException.
    try {
        Person p = (Person) o;

        // If we make it to this point in the code, we know we're
        // dealing with a Person object; next, we'll compare ssn's.
        if (this.getSsn().equals(p.getSsn())) {
            // We'll deem p equal to THIS person.
            isEqual = true;
        }
        else {
            isEqual = false;
        }
    }
    catch (ClassCastException e) {
        // They're not equal - o isn't even a Person!
        isEqual = false;
```

```
        }

        return isEqual;
    }
}
```

Let's explore the `equals` method in detail:

- The `equals` method header as declared by the `Object` class accepts a generic `Object` as an argument. When overriding this method for `Person`, we may not change this signature to accept a `Person` reference explicitly. Rather, we'll attempt to ***cast*** `o` to be a reference to a `Person` to determine if, at run time, `o` really ***is*** referring to a `Person`; if the cast attempt succeeds, then we'll wind up with two handles on the same object–`Object o` and `Person p`:

```
// Overriding the equals method that we inherited from the
Object class.
public boolean equals(Object o) {
    boolean isEqual;

    // Try to cast the Object reference into a Person reference.
    // If this fails, we'll get a ClassCastException.
    try {
   Person p = (Person) o;
```

- If a `ClassCastException` is ***not*** thrown by the previous line of code, then we know that `o` is indeed referring to a `Person`. Our next step is to then compare the `ssn` attribute value of `p` (a.k.a. `o`) to the `ssn` attribute value of ***this*** `Person`–that is, the `Person` whose `equals` method we've invoked to begin with:

```
    // If we make it to this point in the code, we know we're
    // dealing with a Person object; next, we'll compare ssn's.
    if (this.getSsn().equals(p.getSsn())) {
```

- If the `ssn` values are equal, then the `Person` referred to by `p` (and `o`) is deemed "equal to" ***this*** `Person`; otherwise, they are not equal:

```
            // We'll deem p equal to THIS person.
            isEqual = true;
        }
        else {
            isEqual = false;
        }
    }
```

- Conversely, if a `ClassCastException` arose when we tried to cast `o` as a `Person`, then we know that `o` doesn't "equal" this `Person` to whom we're comparing–after all, `o` isn't even a `Person`!

```
catch (ClassCastException e) {
    // They're not equal - o isn't even a Person!
    isEqual = false;
}
```

Now that we've overridden the `equals` method for the `Person` class, let's return to our earlier example program where we were testing the equality of `Person` objects. We'll rerun the code exactly as it was written before (code repeated in the following for your convenience):

```
public class EqualsTest2 {
  public static void main(String[] args) {
    // We'll create one Person object ...
    Person p1 = new Person("222-22-2222", "Fred");

    // ... and maintain two handles on it (p1 and p2).
    Person p2 = p1;

    // We'll create a second different Person object with exactly the same
    // attribute values as the first Person object that we created,
    and will
    // use variable p3 to maintain a handle on this second object.
    Person p3 = new Person("222-22-2222", "Fred");

    // Are p1 and p2 "equal"?
    System.out.println("The expression p1 == p2 evaluates to: " +
    (p1 == p2));
```

```
System.out.println("The expression p1.equals(p2) evaluates to: " +
    (p1.equals(p2)));

// Are p1 and p3 "equal"?
System.out.println("The expression p1 == p3 evaluates to: " +
(p1 == p3));
System.out.println("The expression p1.equals(p3) evaluates to: " +
    (p1.equals(p3)));
```

However, because we've overridden the `equals` method for the `Person` class, we'll now get a different result when we execute the code; the output is as follows:

```
The expression p1 == p2 evaluates to: true
The expression p1.equals(p2) evaluates to: true
The expression p1 == p3 evaluates to: false
The expression p1.equals(p3) evaluates to: true
```

Note that even though `p1` and `p3` are referring to two physically distinct `Person` objects, these objects have ***identical values*** for their `ssn` attribute, and so they are now deemed to be "equal."

Consider overriding the `equals` method of any class for which you have a frequent need to test the equivalence of objects.

Overriding the toString Method

Recall that the (overloaded) `print` and `println` methods do their best to render whatever expression is passed in as an argument into an equivalent `String` representation. This is relatively straightforward for simple data types

```
int x = 7;
double y = 3.8;
boolean z = false;
System.out.println(x);
System.out.println(y);
System.out.println(z);
```

Output:

```
7
3.8
false
```

or for expressions that ***resolve*** to one of these types:

```
int x = 7;
double y = 3.8;
boolean z = false;
System.out.println(x + y);
System.out.println(x == y);
```

Output:

```
10.8
false
```

If we were to try to print the value of an expression that resolves to an ***object reference***, on the other hand

```
Student s = new Student("Harvey", "123-45-6789");

// Try to print the object reference directly.
System.out.println(s);
```

we'd most likely get a rather cryptic-looking result, similar to the following:

```
Student@71f71130
```

where "`Student@71f71130`" represents an internal object ID relevant only to the JVM. ***Why is this?***

It just so happens that all objects inherit a method from the `Object` class with the header

```
String toString();
```

As inherited from the `Object` class, the `toString` method is defined to print the name of the class to which an object belongs, followed by an "at" sign (@), followed by an internal object ID– *`Classname@internalID`*–as we saw with the preceding `Student` example.

Note that for expressions that resolve to an object reference, the `println` and `print` methods ***automatically*** invoke that object's `toString` method; that is, the following two lines of code are equivalent:

```
System.out.println(s.toString());
```

and

```
System.out.println(s);
```

What we probably meant to do was to print one or more of the `Student` object's attributes as a ***representation*** of the object, for example, perhaps the student's `name`, followed by their `ssn` in parentheses:

```
John Smith (123-45-6789)
```

We can accomplish this by ***overriding*** the `toString` method for the `Student` class to define what it is that we wish to have printed as a representation of a `Student`. For example, the `Student` class may override the `toString` method as follows:

```
public String toString() {
  return this.getName() + " (" + this.getSsn() + ")");
}
```

As a result, the following code

```
Student s = new Student("Harvey", "123-45-6789");
System.out.println(s);
```

would now produce the alternative output

```
Harvey (123-45-6789)
```

as desired.

It's generally a good idea to routinely override the `toString` method for all user-defined classes.

Static Initializers

We learned in Chapter 5 that constructors are used to initialize an object's state when it is first instantiated:

```
public class Student {
    private String name;
    private String major;
    // etc.

    // Constructor.
    public Student(String n) {
        setName(n);
        setMajor("TBD"); // default value
        // etc.
    }

    // etc.
}
```

The initialization code contained within a constructor is executed each time a new object of a given type is instantiated; thus, we wouldn't want to include code to initialize a `static` variable in such a constructor:

```
public class Student {
    private String name;
    private String major;
    private int studentIDNo;
    private static int nextAvailableStudentIDNo;
    // etc.

    // Constructor.
    public Student(String n) {
        setName(n);
        setMajor("TBD"); // default value
        // etc.

        // This would be a problem - we'd be resetting
        // the value of nextAvailableStudentIdNo
```

```
        // to 10000 with eachnew Student object that we create.
        nextAvailableStudentIdNo = 10000;
        setStudentIDNo(nextAvailableStudentIdNo++);
        // etc.
    }

    // etc.
}
```

It's possible to define initialization code that will get performed only ***once*** in a given application invocation–specifically, when the class's bytecode of which it is a part is first loaded into the JVM–by enclosing it in what is known as a `static` code block, thus creating a **static initializer**. The general syntax for doing so is shown in the following:

```
public class Student {
    private String name;
    private String major;
    private int studentIDNo;
    private static int nextAvailableStudentIDNo;
    // etc.

    // A static initializer.
    static {
        // Whatever code is enclosed within this block
        // will be executed once, when the enclosing
        // class (Student, in this case) is loaded into
        // the JVM's memory.
        nextAvailableStudentIDNo = 10000;
    }

    // Constructor.
    public Student(String n) {
        setName(n);
        setMajor("TBD"); // default value
        // etc.

        setStudentIDNo(nextAvailableStudentIdNo++);
        // etc.
```

```
    }

    // etc.
}
```

We can of course do anything that makes good sense within a static initializer: in addition to initializing static variables, we can establish resources for an application, such as a file or database connection, that need only be performed once per application session.

Variable Initialization, Revisited

Recall our discussion of variable initialization in Chapter 2:

> *In Java, most variables aren't automatically assigned an initial value when they are declared; we must explicitly assign a value to a variable before the variable's name is referenced in a subsequent Java expression in order to avoid compilation errors.*

For example, this next bit of code

```
1  public class Problem {
2      public static void main(String[] args) {
3          // Declare several local variables within the main() method.
4          int i;  // not automatically initialized
5          int j;  // ditto
6          j = i;  // compilation error!
7      }
8  }
```

was shown to produce the following compilation error on line 6:

```
Variable i may not have been initialized
```

We thus learned that it's best to explicitly initialize such variables, for example:

```
public class NotAProblem {
    public static void main(String[] args) {
        // Declare several local variables within the main method and
```

```
        // explicitly initialize them all.
        int i = 0;
        double x = 0.0;
        boolean test = false;
        Student student = null;
        // etc.
    }
}
```

Out of necessity, we oversimplified the explanation of variable initialization when we first discussed this topic in Chapter 2 because we hadn't yet introduced the notion of objects and attributes. To properly discuss initialization in Java, we must distinguish among

- ***Local variables***–that is, variables declared ***within a method*** and whose scope is therefore only the extent of the enclosing method
- ***Attributes*** of a class, a.k.a. ***instance variables*** in Java (because they are variables whose values are associated with an object/class ***instance*** as we discussed in Chapter 7)
- ***Static variables*** (informally referred to as "static attributes") of a class

As it turns out

- All ***local variables,*** whether declared to be of one of the eight primitive Java types or of a reference type, are considered by the compiler to be ***uninitialized*** until they have been explicitly initialized within a program.
- All ***instance/static variables***, on the other hand, whether declared to be of a simple Java type or of a reference type, are ***automatically initialized to the default zero-equivalent value for the type in question.***

Thus, in the following code example

```
public class Student {
  // Attributes/instance variables.
  private int age;
```

```
  private boolean isHonorsStudent;
  private double gpa;
  private Professor advisor;

  // A static/class variable.
  private static int totalStudents;

  void someMethod() {
    // Local variables.
    int x;
    boolean flag;
    double y;
    Professor p;
    // etc.

    // Details of method body omitted.
  }
  // etc.
```

variables would be initialized as shown in Table 13-2.

Table 13-2. *The State of Automatic Initialization of the Features of Class* `Student`

Variable	Initialized?
`age`	yes; to 0
`isHonorsStudent`	yes; to `false`
`gpa`	yes; to 0.0
`advisor`	yes; to `null`
`totalStudents`	yes; to 0
`x`	no
`flag`	No
`y`	No
`p`	No

Therefore, when first instantiated, an object has the appropriate data structure as prescribed by its class, but all of its attributes will be initialized to their zero-equivalent values.

Variable Arguments (varargs)

The **varargs**, or variable arguments, feature of Java introduced with version 5.0 allows us to declare methods that can accept a ***variable*** number of ***similarly typed*** arguments. To declare a method as accepting a variable number of arguments, the syntax is as follows:

```
accessibility returnType methodName(argType ... args) {
    // Iterate through the args array; details omitted.
}
```

where `args` is the name of a parameter representing an ***array*** of type *`argType`* and ***the explicit inclusion of an ellipsis*** "`...`" between *`argType`* and `args` signals that `args` is actually to be treated as an array of zero or more elements of type *`argType`*.

Within a so-called **varargs method**, we iterate through the `args` array in similar fashion to the way that we learned to iterate through the `args` array of the `main` method earlier in this chapter to read command-line arguments. Whereas the `main` method of an application can only accept an array of `String` values as `arg`(ument)`s`, however

```
public static void main(String[] args) { // etc.
```

a varargs method in general may declare its `args` parameter to be of ***any*** type:

- A predefined reference type, such as `String: public void foo(String ... args) {`
- A primitive type, such as `int: public void foo(int ... args) {`
- A user-defined type, such as `Person: public void foo(Person ... args) {`

(Note that square brackets [ ] are ***not*** needed for `args` to serve as an array; this happens automatically by virtue of our inclusion of an ellipsis "`...`".)

The following example illustrates the use of varargs methods to accept `String`, `int`, `Person`, and `Object` arguments, respectively; for purposes of this example, we assume that `Student` and `Professor` are subclasses of the `Person` (super)class and that `Pineapple`, `Bicycle`, and `Cloud` are three unrelated classes all derived directly from `Object`:

```
import java.util.ArrayList;

public class VarargsExample {
    public static void main(String[] args) {
        // Invoking methods that define variable argument
        // signatures, where the type of argument(s)
        // to be passed in is designated by the name
        // of the method.

        stringExample("foo", "bar");
        stringExample("eeny", "meeny", "miney", "mo");

        intExample(1, 3, 9, 27);
        intExample();

        Student student = new Student("Fred");
        Professor professor = new Professor("Dr. Carson");
        personExample(student, professor);

        ArrayList<String> arrayList = new ArrayList<>();
        arrayList.add("Hello!");
        arrayList.add("How are you?");
        arrayList.add("Goodbye ...");
        objectExample(student, arrayList);

        objectExample2(new Pineapple(), new Bicycle(), new Cloud());
    }

    //------------------------------
    // Here are our varargs methods.
    //------------------------------

    private static void stringExample(String ... args) {
        System.out.println("In stringExample, there were " +
            args.length + " arguments.");

        for (int i = 0; i < args.length; i++) {
            System.out.println("    " + args[i] + " is a " +
                args[i].getClass().getName());
        }
```

```
        System.out.println();
    }

    private static void intExample(int ... args) {
        System.out.println("In intExample, there were " +
            args.length + " arguments.");

        for (int i = 0; i < args.length; i++) {
            System.out.println("    " + args[i] +
                " is an int");
        }

        System.out.println();
    }

    private static void personExample(Person ... args) {
        System.out.println("In personExample, there were " +
            args.length + " arguments.");

        for (int i = 0; i < args.length; i++) {
            System.out.println("    " + args[i] + " is a " +
                args[i].getClass().getName());
        }

        System.out.println();
    }

    private static void objectExample(Object ... args) {
        System.out.println("In objectExample, there were " +
            args.length + " arguments.");

        for (int i = 0; i < args.length; i++) {
            System.out.println("    " + args[i] + " is a " +
                args[i].getClass().getName());
        }

        System.out.println();
    }
```

```
    private static void objectExample2(Object ... args) {
        // Here, we assume that we know that the args array will contain
        // an assortment of Pineapples, Bicycles, and Clouds.
        for (int i = 0; i < args.length; i++) {
            // Note casts.
            if (args[i] instanceof Pineapple) {
                ((Pineapple) args[i]).eat();
            }
            else if (args[i] instanceof Bicycle) {
                ((Bicycle) args[i]).ride();
            }
            else if (args[i] instanceof Cloud) {
                ((Cloud) args[i]).paint();
            }
        }

        System.out.println();
    }
}
```

Output:

```
In stringExample, there were 2 arguments.
    foo is a java.lang.String
    bar is a java.lang.String

In stringExample, there were 4 arguments.
    eeny is a java.lang.String
    meeny is a java.lang.String
    miney is a java.lang.String
    mo is a java.lang.String

In intExample, there were 4 arguments.
    1 is an int
    3 is an int
    9 is an int
    27 is an int
```

```
In intExample, there were 0 arguments.

In personExample, there were 2 arguments.
    Fred is a Student
    Dr. Carson is a Professor

In objectExample, there were 2 arguments.
    Fred is a Student
    [Hello!, How are you?, Goodbye ...] is a java.util.ArrayList

Eating a pineapple ...
Riding a bicycle ...
Painting a cloud ...
```

Summary

In this chapter, we discussed

- How formal Java-specific terminology differs from the informal terminology commonly used to describe object concepts
- The object nature of `String`s and some of the methods provided to manipulate them
- How we can form highly complex expressions by chaining messages
- How Java exceptions arise and how to gracefully handle them, including the ability to define our own custom exception types
- How to read input from the command line when a Java application is invoked, as well as how to prompt the user for keyboard inputs, useful techniques when running a command-line-driven application
- Using the `this` keyword to self-reference an object from within one of its methods
- The utility features of the wrapper classes–`Integer`, `Double`, etc.–to convert `String` data to numeric values and vice versa

- The nature of object identities in Java, how to discover the true class that an object belongs to, and how to test the equality of two Java objects
- The importance of overriding the `toString` method for all user-defined classes and how the `equals` method may similarly be overridden

With all of the Java knowledge at our fingertips, we are now ready to proceed to building the SRS application.

EXERCISES

1. [*Coding*] Write a Java program that will accept a series of individual characters, separated by one or more spaces, as command-line input and will then "glue" them together to form a word. For example, if we were to invoke the program as follows

   ```
   java Glue B A    N A   N    A
   ```

   then the program should output:

   ```
   BANANA
   ```

   with no spaces.

2. [*Coding*] Write a Java program that accepts a sentence as command-line input and outputs statistics about this sentence. For example, if we were to invoke the program as follows

   ```
   java SentenceStatistics this is my sample sentence
   ```

   then the program should output the following results:

   ```
   number of words:          5
   longest word(s):             sentence
   length of longest word(s):  8
   shortest word(s):          is my
   length of shortest word(s): 2
   ```

   (To keep things simple, do not use any punctuation in your sentence.)

3. [*Coding*] Practice declaring enums by declaring

   An enum called `Weekday` that represents the seven days of the week as `String`s.

   An enum called `SolarSystem` that represents the nine planets in our Solar System as `Planet` objects: Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune, Pluto. In support of this enum, declare a simple `Planet` class that has the following features: (a) a single `String` attribute representing the planet's name, (b) a constructor that takes the planet's name as an argument, (c) a `toString` method that returns the phrase "`Planet: planetName`".

4. [*Coding*] Practice with javadoc comments by retrofitting such comments into any Java code that you've previously written.

5. [*Coding: Advanced*] Use the `Scanner` class presented in this chapter as the basis of a simple program called `GuessIt` that asks a user to guess a number between 1 and 10 (program the "answer" to be guessed as a local `int` variable in the `GuessIt main` method).

A sample interactive scenario using the `GuessIt` program follows; for this example, assume that the right answer is 6:

C:\Programs> java GuessIt

Please type a number between 1 and 10, and press Enter: 3

Too low; please try again: 7

Too high; please try again: 6

CHAPTER 14

Transforming the Model into Java Code

It's now time to turn our attention back to the UML class diagram that we produced in Part 2 of the book in order to render that object-oriented "blueprint" into Java code, focusing solely on what it takes to accurately model the SRS domain information in an OO programming language. In this chapter, you'll learn how to represent all of the following object-oriented constructs in Java code

- Associations of varying multiplicities (one-to-one, one-to-many, and many-to-many), including aggregations
- Inheritance relationships
- Association classes
- Reflexive associations
- Abstract classes
- Metadata
- Static attributes and methods

along with practical guidelines as to when to use these various constructs. We'll also cover a technique for testing your core classes via a command-line–driven application.

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_14

Suggestions for Getting the Maximum Value from This Chapter

One of the ***best*** ways to master a language is to start with code that works and to experiment with it. A good approach is to get some hands-on experience with Java by actually compiling and running the SRS application; studying it, so as to familiarize yourself with the techniques that we've used; and, finally, modifying it yourself. As you know, exercises provided at the end of each chapter provide specific suggestions for experiments that you may wish to try. Therefore, before you dive into this chapter, ***I encourage you to download the Java source code for Chapter 14 from the book's GitHub repository (github.com/apress/beginning-java-objects-3e) if you haven't already done so***.

The code that I've written for the SRS application is sizeable; to have included the complete code listing for each and every Java class intact in every chapter would have been prohibitive. So, to make this as effective a learning experience as possible for you, I've chosen to feature just those portions of code that are particularly critical to your understanding of object concepts as they translate into the Java language.

I realize that you will, of course, need access to the complete source code to round out your understanding of the SRS application as it's been implemented, which is another compelling reason for you to download the SRS source code at this time. I recommend printing a copy of the SRS source code and putting it into a loose-leaf binder, so that you can jot down notes on the printouts as you read through this chapter.

The SRS Class Diagram Revisited

Let's turn our attention back to the SRS class diagram that we produced in Part 2 of the book. In speaking with our sponsors for the SRS, we learn that they've decided to cut back on a few features in the interest of reducing development costs:

- First of all, they have decided not to automate students' plans of study via the SRS. Instead, it will be up to each student to make sure that the courses that they register for are appropriate for the degree that they are seeking.
- Since automated plans of study are being eliminated, there will no longer be a need to track who a student's faculty advisor is. The only reason for modeling the *advises* relationship between the `Professor` and `Student` classes in the first place was so that a student's advisor could be called upon to approve a tentative plan of study when a student had first posted it via the SRS.
- Finally, our sponsors have decided that maintaining a waitlist for a section once it becomes full is a luxury that they can live without, since most students, upon learning that a section is full, immediately choose an alternative course anyway.

We've thus pared down the SRS class diagram accordingly, to eliminate these unnecessary features. Also, to keep the diagram from getting too cluttered, we've chosen not to reflect attribute types or full method signatures in the UML. The resultant diagram is shown in Figure 14-1.

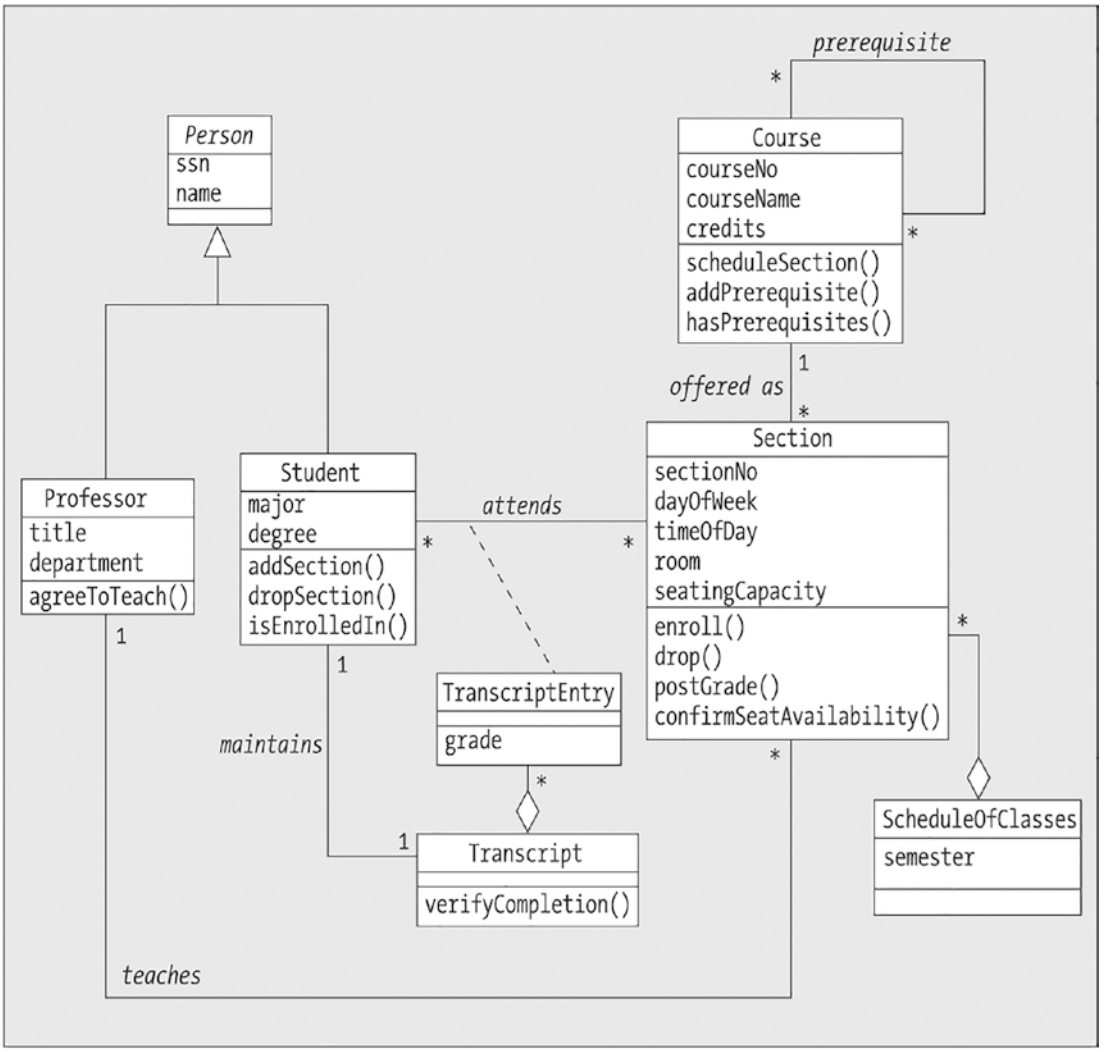


Figure 14-1. *Our new UML "blueprint" for Chapter 14*

Fortunately for us, the resultant model still provides examples of all of the key object-oriented elements that we need to learn how to program, as listed in Table 14-1.

Table 14-1. *OO Features as Reflected in Figure 14-1*

OO Feature	Embodied in the SRS Class Diagram As Follows
Inheritance	The `Person` class serves as the base class for the `Student` and `Professor` subclasses.
Aggregation	We have two examples of this: the `Transcript` class represents an aggregation of `TranscriptEntry` objects, and the `ScheduleOfClasses` class represents an aggregation of `Section` objects.
One-to-one association	The *maintains* association between the `Student` and `Transcript` classes.
One-to-many association	The *teaches* association between `Professor` and `Section`, the *offered as* association between `Course` and `Section`.
Many-to-many association	The *attends* association between `Student` and `Section`, the *prerequisite* (reflexive) association between instances of the `Course` class.
Association class	The `TranscriptEntry` class is affiliated with the *attends* association.
Reflexive association	The *prerequisite* association between instances of the `Course` class.
Abstract class	The `Person` class will be implemented as an abstract class.
Metadata	Each `Course` object embodies information that is relevant to multiple `Section` objects.
Static attributes	Although not specifically illustrated in the class diagram, we'll take advantage of `static` attributes when we code the `Section` class.
Static methods	Although not specifically illustrated in the class diagram, we'll take advantage of `static` methods when we code the `TranscriptEntry` class.

In this chapter, we're going to code the eight model classes called for by the SRS class diagram

```
Course.java
Person.java
Professor.java
ScheduleOfClasses.java
Section.java
```

```
Student.java
Transcript.java
TranscriptEntry.java
```

along with a "wrapper" class, `SRS`, that will encapsulate the `main` method of our application and a supporting `enum`(eration), `EnrollmentStatus`, the purpose for which will be explained in due time.

The Person Class (Specifying Abstract Classes)

Let's start with writing the code for the `Person` class (see Figure 14-2).

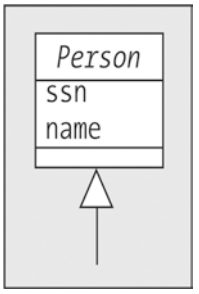


Figure 14-2. *The `Person` class*

The first thing that we notice in the UML diagram is that the name of the class is italicized, which as you learned in Chapter 10 means that `Person` is to be implemented as an abstract class. By including the keyword `abstract` in the class declaration, we prevent client code from ever being able to instantiate a `Person` object directly:

```
public abstract class Person {
```

Attributes of Person

The `Person` class icon specifies two simple attributes. We'll make ***all*** of our attributes `private` throughout the SRS application unless otherwise stated:

```
    //------------
    // Attributes.
    //------------

    private String name;
    private String ssn;
```

Person Constructors

We'll provide a constructor for the Person class that accepts two arguments, so as to initialize our two attributes:

```
//----------------
// Constructor(s).
//----------------

public Person(String name, String ssn) {
    this.setName(name);
    this.setSsn(ssn);
}
```

Note that we use the Person class's own set methods to set the values of the name and ssn attributes, a best practice that was recommended in Chapter 4.

And, because the creation of any constructor for a class eliminates that class's default constructor as we discussed in Chapter 4, we'll program a replacement for the default constructor, as well, to avoid some of the issues related with constructors and inheritance that we discussed in Chapter 5:

```
public Person() {
    this.setName("?");
    this.setSsn("???-??-????");
}
```

Person Accessor Methods

Next, we provide accessor methods for all of the attributes, observing proper accessor method signature syntax as recommended in Chapter 4. All of our accessor methods, in all classes, will be declared with public accessibility:

```
//------------------
// Accessor methods.
//------------------

public void setName(String n) {
    name = n;
}
```

```
    public String getName() {
        return name;
    }

    public void setSsn(String s) {
        ssn = s;
    }

    public String getSsn() {
        return ssn;
    }
```

toString( )

We'd like for all subclasses of the `Person` class to override the version of the `toString` method that would normally be inherited from the `Object` class, a practice that we discussed in Chapter 13. However, we don't want to bother coding the details of such a method for `Person`; we'd prefer to let each subclass handle the details of how the `toString` method will work in its own class-appropriate way.

The best way to enforce this requirement for a `toString` method is to declare an ***abstract method*** for this method in `Person`, as we discussed in Chapter 7:

```
    //-----------------------------
    // Miscellaneous other methods.
    //-----------------------------

    // We'll let each subclass determine how it wishes to be
    // represented as a String value.

    public abstract String toString();
```

This will ensure that all classes derived from `Person` override this abstract method with a concrete version of their own. (This is not mandatory in all cases; we simply wish for all of the SRS-derived classes to do so.)

Note that since the Person class itself would normally have inherited a ***generic*** version of toString from the Object class, we're in essence overriding the ***concrete*** toString of Object with an ***abstract*** version in Person—this is a perfectly fine thing to do.

display()

We also want all subclasses of Person to implement a display method, to be used for printing the values of a Person object's attributes to the command window. We'll use the display method solely for testing our application, to verify that an object's attributes have been properly initialized. But, rather than making this method abstract, as well, we'll go ahead and actually program the body of this method, since we know how we'd like the attributes of Person to be displayed when these are inherited, at a minimum:

```
public void display() {
    System.out.println("Person Information:");
    System.out.println("\tName:  " + this.getName());
    System.out.println("\tSoc. Security No.:  " + this.getSsn());
}
```

By doing so, we are facilitating code reuse: subclasses of Person (Student, Professor) will be able to use the super keyword to incorporate this logic in their own display methods without having to duplicate it, as we discussed in Chapter 5. As an example, here is a preview excerpt from the Student class's display method:

```
public void display() {
  // First, let's display the generic Person info.
  super.display();
  // etc.
```

Again, note that we are invoking the Person class's get methods from within the println calls vs. accessing attributes directly. Also observe that we are inserting a tab character (\t) in the second and third println calls so that those two lines of printed output will be indented by one tab stop.

That's all there is to programming the Person class—it's pretty straightforward. We'll tackle the Student and Professor subclasses of Person next.

The Student Class (Reuse Through Inheritance, Extending Abstract Classes, Delegation)

Figure 14-3 shows the UML representation of the `Student` class.

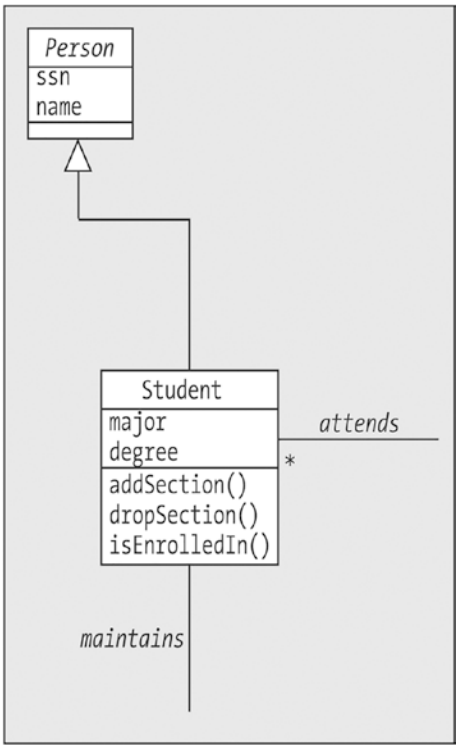


Figure 14-3. *The `Student` class*

We indicate that `Student` is a subclass of `Person` by using the `extends` keyword:

```
public class Student extends Person {
```

Attributes of Student

There are two attributes explicitly called out for the `Student` class in our UML diagram: `major` and `degree`. However, we learned in Chapter 10 that we must also encode associations as attributes. `Student` participates in two associations:

- *attends*, a many-to-many association with the `Section` class
- *maintains*, a one-to-one association with the `Transcript` class

So we must allow for each Student to maintain handles on a single Transcript object and on many Section objects.

Of the various types of Java collection covered in Chapter 6, an ArrayList seems like the best choice for managing multiple Section handles:

- An Array is a bit too rigid; we'd have to size the Array in advance to be large enough to accommodate references to all of the Sections that a Student will ever attend over the course of their studies at the university. An ArrayList, on the other hand, can start out small and automatically grow in size as needed.
- The decision of using an ArrayList vs. a dictionary type of collection comes down to whether or not we'll need to retrieve an object reference from the collection based on some key value. We don't anticipate the need for such a lookup capability as it pertains to the Sections that a Student has attended; we ***will*** need the ability to verify if a Student has taken a particular Section or not, but this can be accomplished by using the ArrayList class's contains method. That is, if attends is declared to be of type ArrayList, then we can use the statement

  ```
  if (attends.contains(someSection)) { ...
  ```

  to verify whether or not a Student has taken a particular Section. For most other uses of the attends collection, we'll need to step through the entire collection anyway, as when printing out a Student's course schedule. So an ArrayList should serve our purposes quite nicely.

The attributes for the Student class thus turn out as follows:

```
//------------
// Attributes.
//------------

private String major;
private String degree;
private Transcript transcript;
private ArrayList<Section> attends;
```

Of course, by virtue of inheritance, `Student` also inherits the attributes declared by `Person`—namely, `name` and `ssn`—but with ***private*** accessibility. Hence, as we discussed in Chapter 5, these attributes do indeed become part of the "bone structure" of a `Student` object, but the symbols `name` and `ssn` are not within the ***namespace*** of the `Student` class. Thus, we'll use the associated ***public*** accessor ("get"/"set") methods, also inherited from `Person`, to access them as needed.

Since we are using the `ArrayList` class, we'll need to remember to include the following `import` directive at the beginning of the `Student` class, ahead of the class declaration:

```
import java.util.ArrayList;
```

Student Constructors

We'll provide two constructors for convenience of initializing attributes:

```
    //----------------
    // Constructor(s).
    //----------------

    public Student(String name, String ssn, String major, String degree) {
```

In the first `Student` constructor, we use the construct `super(`*`arguments`*`)` to reuse the code of one of the `Person` class constructors from within our `Student` constructor, to establish the "`Person`ness" of a `Student`. Recall from our discussion of the `super` keyword in Chapter 5 that any invocation of `super(...)` must be the ***first*** such line in a subclass's constructor:

```
        // Reuse the code of the parent's constructor.
        super(name, ssn);
```

After setting the values of the `major` and `degree` attributes based on arguments passed into the constructor

```
        this.setMajor(major);
        this.setDegree(degree);
```

we set about to instantiate the `transcript` attribute as follows:

```
        // Create a brand new Transcript to serve as this Student's
        // transcript.
        this.setTranscript(new Transcript(this));
```

Let's evaluate the preceding line of code from the inside out:

- First, we instantiate a brand-new ***unnamed*** `Transcript` object, passing a reference to ***this*** `Student` in as the lone argument to the `Transcript` constructor:

  ```
  new Transcript(this)
  ```

  (Since we haven't discussed the structure of the `Transcript` class yet, the signature of its constructor may seem a bit puzzling, but it will make sense once we get a chance to review the `Transcript` class in its entirety later in this chapter.)

- We then nest this instantiation request inside of a call to `setTranscript`:

  ```
  this.setTranscript(new Transcript(this));
  ```

- Note that we ***could*** have accomplished this with ***two*** lines of code instead of one:

  ```
  Transcript t = new Transcript();
  this.setTranscript(t);
  ```

  But there is no point in going to the trouble of declaring a variable `t` to serve as a reference to a `Transcript` object if we're only going to reference the variable one time and then discard it when it goes out of scope as soon as the constructor exits. As we discussed in Chapter 13, it is commonplace to chain together/nest method calls in a single Java statement.

Finally, as we discussed in Chapter 6, we routinely instantiate collection attributes such as the `attends ArrayList` in a constructor to guarantee that we have an empty "egg carton" ready for us when it is time to add "eggs":

```
        // Note that we're instantiating empty support Collection(s).
        attends = new ArrayList<>();
    }
```

We choose to overload the Student constructor by providing a second constructor, shown in the following code, to be used if we wish to create a Student object for which we don't yet know the name, major field of study, or degree sought. As discussed in Chapter 4, we use the syntax this(arguments) to reuse the code from the first Student constructor within the second, passing in the String value "TBD" to serve as a temporary value for the name, major, and degree attributes:

```
// A second simpler form of constructor.

public Student(String ssn) {
    // Reuse the code of the first Student constructor.
    this("TBD", ssn, "TBD", "TBD");
}
```

Student Accessor Methods

We provide accessor methods for all of the simple (non-collection) attributes:

```
//------------------
// Accessor methods.
//------------------

public void setMajor(String major) {
    this.major = major;
}

public String getMajor() {
    return major;
}

public void setDegree(String degree) {
    this.degree = degree;
}

public String getDegree() {
    return degree;
}
```

```
public void setTranscript(Transcript t) {
    transcript = t;
}

public Transcript getTranscript() {
    return transcript;
}
```

And, as mentioned previously, `Student` inherits the accessor methods of the `Person` class, as well.

For the `attends` collection attribute, we will provide the methods `addSection` and `dropSection` in lieu of traditional `accessor` methods, to be used for adding and removing `Section` objects to and from the `ArrayList`; we'll talk about these methods momentarily.

display()

As we did for `Person`, we choose to provide a `display` method for `Student` for use in testing our command-line version of the SRS. Because a `Student` is a `Person` and because we've already gone to the trouble of programming a `display` method for the attributes inherited from `Person`, we'll reuse that method code by making use of the `super` keyword before going on to additionally display attribute values specific to a `Student` object:

```
public void display() {
    // First, let's display the generic Person info.
    super.display();

    // Then, display Student-specific info.
    System.out.println("Student-Specific Information:");
    System.out.println("\tMajor:  " + this.getMajor());
    System.out.println("\tDegree:  " + this.getDegree());
    this.displayCourseSchedule();
    this.printTranscript();

    // Finish with a blank line.
    System.out.println();
}
```

Note that we are calling two of the Student class's other methods, displayCourseSchedule and printTranscript, from within the Student's display method. We chose to program these as separate methods vs. incorporating their code into the body of the display method to keep the display method from getting too cluttered.

toString( )

By extending an abstract class, as we are in deriving the Student subclass from Person, we implicitly agree to override any abstract method(s) specified by the parent class with concrete methods. In the case of the Person class, we have one such method, toString:

```
// We are forced to program this method because it is specified
// as an abstract method in our parent class (Person); failing to
// do so would render the Student class abstract, as well.
//
// For a Student, we wish to return a String as follows:
//
//     Joe Blow (123-45-6789) [Master of Science - Math]

public String toString() {
    return this.getName() + " (" + this.getSsn() + ") [" + this.
    getDegree() +
           " - " + this.getMajor() + "]";
}
```

displayCourseSchedule()

The displayCourseSchedule method is a more complex example of delegation; we'll defer a discussion of this method until we've discussed a few more of the SRS classes.

addSection( )

When a Student enrolls in a Section, this method will be used to pass a reference to that Section to the Student object so that the Section reference may be stored in the attends ArrayList:

```
public void addSection(Section s) {
    attends.add(s);
}
```

This is yet another example of delegation: the Student object is delegating the work of organizing Section references to its encapsulated collection.

dropSection( )

When a Student withdraws from a Section, this method will be used to pass a reference to the "dropped" Section to the Student object. The Student object in turn delegates the work of removing the Section from the attends ArrayList by invoking its remove method:

```
public void dropSection(Section s) {
    attends.remove(s);
}
```

isEnrolledIn( )

This method is used to determine whether a given Student is already enrolled in a particular Section—that is, whether that Student's attends collection is already referring to the Section in question—by taking advantage of the ArrayList class's contains method:

```
// Determine whether the Student is already enrolled in THIS
// EXACT Section.
public boolean isEnrolledIn(Section s) {
    if (attends.contains(s)) return true;
    else return false;
}
```

As you can see, there is a lot of delegation going on within the Student class's methods. The ArrayList that we've encapsulated as an attribute of Student does a lot of behind-the-scenes work for the Student object to which it belongs.

isCurrentlyEnrolledInSimilar( )

Although not specified by our model, I've added another version of the `isEnrolledIn` method called `isCurrentlyEnrolledInSimilar`, because I found a need for such a method when coding the `Section` class (coming up later in this chapter). No matter how much thought you put into the object modeling stage of an OO software development project, you will inevitably determine the need for additional attributes and methods for your classes once coding is under way, because coding causes you to think at a very fine-grained level of detail about the "mechanics" of your application.

Because this method is somewhat complex, the method code is shown in its entirety first, followed by an in-depth explanation:

```
// Determine whether the Student is already enrolled in ANY
// Section of this SAME Course.
public boolean isCurrentlyEnrolledInSimilar(Section s1) {
    boolean foundMatch = false;

    Course c1 = s1.getRepresentedCourse();

    for (Section s2 : attends) {
        Course c2 = s2.getRepresentedCourse();
        if (c1 == c2) {
            // There is indeed a Section in the attends()
            // ArrayList representing the same Course.
            // Check to see if the Student is CURRENTLY
            // ENROLLED (i.e., whether or not he or she has
            // yet received a grade).  If there is no
            // grade, he/she is currently enrolled; if
            // there is a grade, then he/she completed
            // the course sometime in the past.
            if (s2.getGrade(this) == null) {
                // No grade was assigned!  This means
                // that the Student is currently
                // enrolled in a Section of this
                // same Course.
                foundMatch = true;
                break;
```

```
            }
        }
    }

    return foundMatch;
}
```

The details of this method are as follows.

In coding the `enroll` method of the `Section` class (to be discussed later in this chapter), I realized that we needed a way to determine whether a particular `Student` is enrolled in ***any*** `Section` of a given `Course`. That is, if a `Student` is attempting to enroll for Math 101, section 1, we want to reject this request if that student is already enrolled in Math 101, section 2:

```
// Determine whether the Student is already enrolled in ANY
// Section of this SAME Course.
public boolean isCurrentlyEnrolledInSimilar(Section s1) {
```

We initialize a local `boolean` variable to `false`, with the intention of resetting it to `true` later on if we do indeed discover that the `Student` is currently enrolled in a `Section` of the same `Course`:

```
boolean foundMatch = false;
```

Next, we obtain a handle on the `Course` object that the `Section` of interest represents, assigning it to reference variable `c1`:

```
Course c1 = s1.getRepresentedCourse();
```

We then use the technique discussed in Chapter 6 for iterating through a collection via a `for` loop, using the variable `s2` to maintain a temporary handle on each `Section` in the `attends` collection, one by one:

```
    for (Section s2 : attends) {
```

Within the `for` loop, we obtain a handle on a second `Course` object, `c2`—the `Course` object that `Section s2` is a `Section` of—and test the equality of the two `Course` objects:

```
        Course c2 = s2.getRepresentedCourse();
        if (c1 == c2) {
```

Note that we use the == test for equality, because as we discussed in Chapter 13, we do indeed wish to know if the two `Course` references `c1` and `c2` are indeed referring to the ***exact same object***.

If we find a match, we're not quite done yet, however, because the `attends` `ArrayList` for a `Student` holds on to all `Sections` that the `Student` has ***ever*** taken. To determine if `Section s2` is truly a `Section` that the `Student` is ***currently*** enrolled in, we must check to see if a grade has been issued for this `Section`. A missing grade—that is, a grade value of `null`—indicates that the `Section` is currently in progress:

```
                if (s2.getGrade(this) == null) {
                    // No grade was assigned!  This means
                    // that the Student is currently
                    // enrolled in a Section of this
                    // same Course.
```

As soon as we have found the ***first*** such match, we can set our `boolean` flag accordingly, break out of the enclosing `while` loop, and return a value of `true` to the caller:

```
                    foundMatch = true;
                    break;
                }
            }
        }

        return foundMatch;
    }
```

getEnrolledSections()

The `getEnrolledSections` method used previously is a simple one-liner:

```
    public Collection<Section> getEnrolledSections() {
        return attends;
    }
```

Note the return type of the method: we're returning what is actually an `ArrayList` reference as a ***generic*** `Collection` reference instead. As you learned in Chapter 7, `Collection` is a predefined interface within the `java.util` package, and since the

`ArrayList` class ***implements*** the `Collection` interface, an `ArrayList` ***is a*** `Collection`. By returning an interface type—`Collection`—rather than an explicit class type—`ArrayList`—from this method, we reserve the right to change the type of collection that we use internally to the `Student` in the future without subjecting client code to a ripple effect, a benefit of interfaces that we discussed at length in Chapter 7.

We must also remember to include the following `import` directive at the beginning of our `Student` class declaration:

```
import java.util.Collection;
```

printTranscript( )

The `printTranscript` method is a straightforward example of ***delegation***. We use the `Student`'s `getTranscript` method to retrieve a handle on the `Transcript` object that belongs to this `Student` and then invoke the `display` method for that `Transcript` object (you'll see the details of this later in the chapter, when we discuss the `Transcript` class):

```
public void printTranscript() {
    this.getTranscript().display();
}
```

Note that, once again, we could have accomplished this with two lines of code instead of one:

```
public void printTranscript() {
  Transcript t = this.getTranscript();
  t.display();
}
```

But the "chained" version is more concise.

Next, we'll turn our attention to the `Professor` class.

The Professor Class (Bidirectionality of Relationships)

Figure 14-4 shows the UML representation of the `Professor` class.

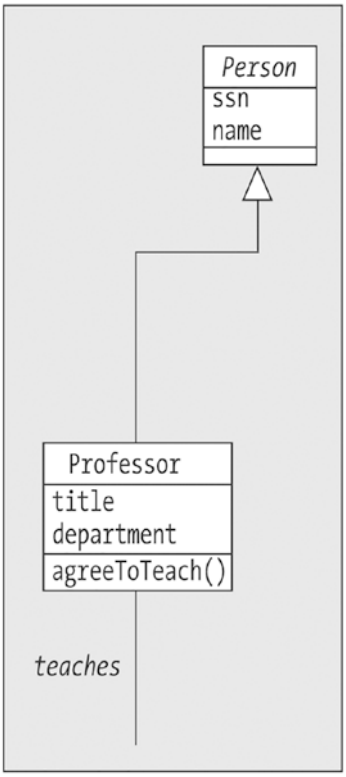


Figure 14-4. *The `Professor` class*

Because the code that is necessary to implement the `Professor` class of `Person` is so similar to that of `Student`, I'll comment only on those features of `Professor` that are particularly noteworthy. I encourage you to look at the full code of the `Professor` class, however, to reinforce your ability to read and interpret Java syntax.

Professor Attributes

The `Professor` class is involved in one association—the one-to-many *teaches* association with the `Section` class—and so we must provide a means for a `Professor` object to maintain multiple `Section` handles, which we do by creating a `teaches` attribute of type `ArrayList`:

```
    //------------
    // Attributes.
    //------------

    private String title;
    private String department;
    private ArrayList<Section> teaches;
```

agreeToTeach( )

Our class diagram calls for us to implement an `agreeToTeach` method. This method accepts a `Section` object reference as an argument and begins by storing this reference in the `teaches ArrayList`:

```
public void agreeToTeach(Section s) {
  teaches.add(s);
```

Associations, as modeled in a UML class diagram, are assumed to be bidirectional. When implementing associations in ***code***, however, we must think about whether or not bidirectionality is important on a case-by-case basis for each association.

- Can we think of any situations in which a `Professor` object would need to know which `Sections` it is responsible for teaching? Yes—for example, when we ask a `Professor` object to print out its teaching assignments.
- How about the reverse? That is, can we think of any situations in which a `Section` object would need to know who is teaching it? Yes—for example, when we print out a `Student`'s course schedule.

So not only must we store a reference to the `Section` object in the `Professor`'s `teaches ArrayList`, but also we must make sure that the `Section` object is somehow notified that this `Professor` is going to be its instructor. We accomplish this by invoking the `Section` object's `setInstructor` method, passing in a handle on the `Professor` object whose method we are in the midst of executing via the `this` keyword, a technique for object self-referencing that we discussed in Chapter 13:

```
    // We want to link these objects bidirectionally.
    s.setInstructor(this);
}
```

displayTeachingAssignments()

The `displayTeachingAssignments` method is very similar in concept to the `displayCourseSchedule` method of `Student`. We'll hold off on discussing the latter until later in this chapter, but once we've discussed `displayCourseSchedule` in detail, you'll be in a position to revisit the `displayTeachingAssignments` method of `Professor`, as well.

We'll turn our attention next to the `Course` class.

The Course Class (Reflexive Relationships, Unidirectional Relationships)

Figure 14-5 shows the UML representation of the `Course` class. The sections that follow provide more detail about this class.

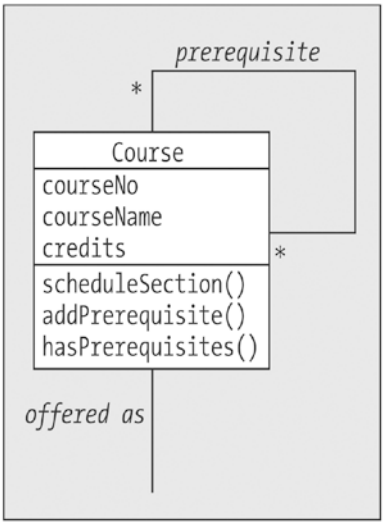


Figure 14-5. *The* `Course` *class*

Course Attributes

Referring back to the SRS class diagram, we see that `Course` has three simple attributes and participates in two associations

- ***offered as***, a one-to-many association with the `Section` class
- ***prerequisite***, a many-to-many reflexive association

for a total of five attributes:

```
//------------
// Attributes.
//------------

private String courseNo;
private String courseName;
private double credits;
private ArrayList<Section> offeredAsSection;
private ArrayList<Course> prerequisites;
```

Note that a reflexive association is handled in exactly the same way that any other association is handled—that is, we've provided the `Course` class with an `ArrayList` attribute called `prerequisites` that enables a given `Course` object to maintain handles on ***other*** `Course` objects.

We have chosen not to encode this reflexive association bidirectionally. That is, a given `Course` object X will know which other `Course` objects A, B, C, etc. serve as ***its*** prerequisites, but it will ***not*** know which `Course` objects L, M, N, etc. consider X to be one of ***their*** prerequisites, as illustrated in Figure 14-6.

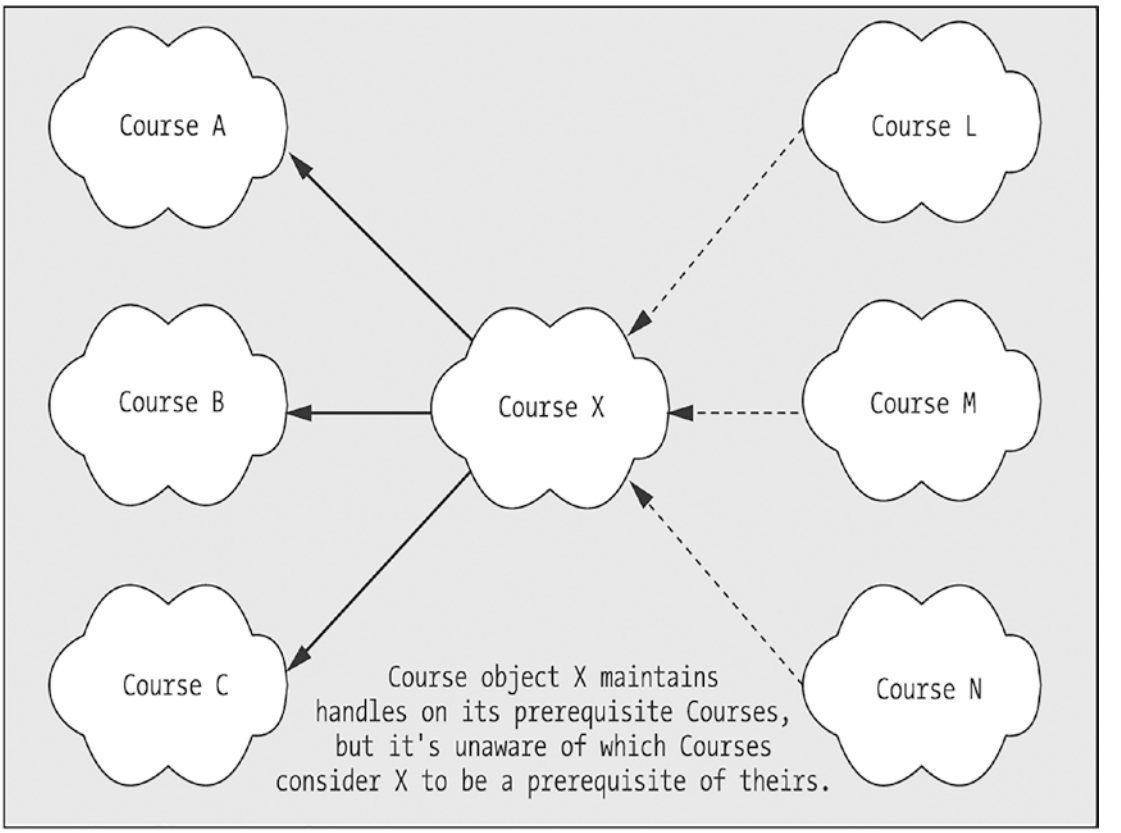


Figure 14-6. *The reflexive* `prerequisite` *association is not implemented bidirectionally*

Our rationale for implementing this association unidirectionally is as follows. A given `Course` X has reason to know which courses are its prerequisites, so that when a `Student` S attempts to enroll in X, X can inquire of S whether S has completed all such prerequisites. X doesn't need to know about courses that require X as a prerequisite—it's up to ***those*** courses to worry about X!

Had we wanted this association to be bidirectional, we would have had to include a ***second*** `ArrayList` of `Course` references as an attribute in the `Course` class, as shown in **bold**

```
//------------
// Attributes.
//------------
```

```
private String courseNo;
private String courseName;
private double credits;
private ArrayList<Section> offeredAsSection;
private ArrayList<Course> prerequisites;
private ArrayList<Course> prerequisiteOf;
```

so that Course object X could hold on to this latter group of Course objects separately.

Course Methods

Most of the Course methods use techniques that should already be familiar to you, based on our discussions of the Person, Professor, and Student classes. I'll highlight a few of the more interesting Course methods here and leave it for you as an exercise to review the rest.

hasPrerequisites( )

This method inspects the size of the prerequisites ArrayList to determine whether or not a given Course has any prerequisite Courses:

```
public boolean hasPrerequisites() {
    if (prerequisites.size() > 0) return true;
    else return false;
}
```

getPrerequisites( )

This method returns a reference to the prerequisites ArrayList as a generic Collection reference, hiding the true identity of the type of collection that we've encapsulated:

```
public Collection<Course> getPrerequisites() {
    return prerequisites;
}
```

We see this method in use within the Course display method, and we'll also see it in use by the Section class a bit later on.

scheduleSection( )

This method illustrates several interesting techniques:

```
public Section scheduleSection(char day, String time, String room,
    int capacity) {
    // Create a new Section (note the creative way in
    // which we are assigning a section number) ...
    Section s = new Section(offeredAsSection.size() + 1,
            day, time, this, room, capacity);

    // ... and then remember it!
    this.addSection(s);

    return s;
}
```

First, note that this method invokes the `Section` class constructor to instantiate a new `Section` object, `s`, storing ***one*** handle on this `Section` object in the `offeredAsSection` `ArrayList` before returning a ***second*** handle on the object to client code.

Second, we are generating the first argument to the `Section` constructor—representing the section number—as a "one-up" number by adding 1 to the size of the `offeredAsSection ArrayList`. The first time that we invoke the `scheduleSection` method for a given `Course` object, the `ArrayList` will be empty, and so the expression

```
offeredAsSection.size() + 1
```

will evaluate to 1, and hence we'll be creating `Section` number 1. The second time that this method is invoked for the same `Course` object, the `ArrayList` will already contain a handle on the first `Section` object that was created, so the expression

```
offeredAsSection.size() + 1
```

will evaluate to 2, and hence we'll be creating `Section` number 2, and so forth.

There is, however, a flaw with this approach: if we were to create, and then ***delete***, `Section` objects, the size of the `ArrayList` would expand and contract, and we could wind up with duplicate `Section` numbers.

Now, let's turn our attention to the `Section` class.

The Section Class (Representing Association Classes, Public Static Final Attributes, Enums)

Figure 14-7 shows the UML representation of the `Section` class. The sections that follow provide more detail about this class.

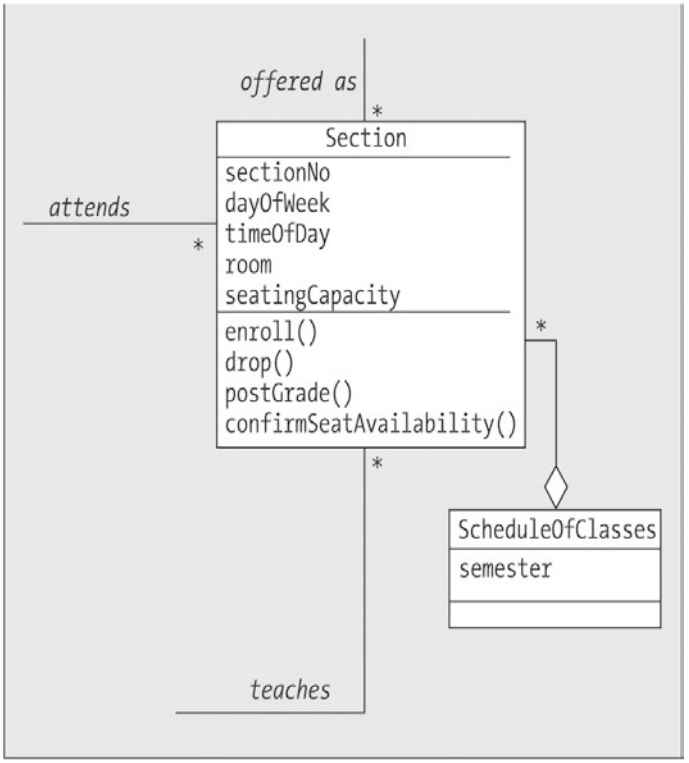


Figure 14-7. *The `Section` class*

Section Attributes

In addition to the following handful of relatively simple attributes

```
//------------
// Attributes.
//------------

private int sectionNo;
private char dayOfWeek;
private String timeOfDay;
```

```
private String room;
private int seatingCapacity;
```

the `Section` class participates in numerous relationships with other classes:

- ***offered as***, a one-to-many association with `Course`
- An unnamed, one-to-many aggregation with `ScheduleOfClasses`
- ***teaches***, a one-to-many association with `Professor`
- ***attends***, a many-to-many association with `Student`

The *attends* association is in turn affiliated with an association class, `TranscriptEntry`. You learned in Chapter 10 that an association class can alternatively be depicted in a class diagram as having direct relationships with the classes at either end of the association, as shown in Figure 14-8.

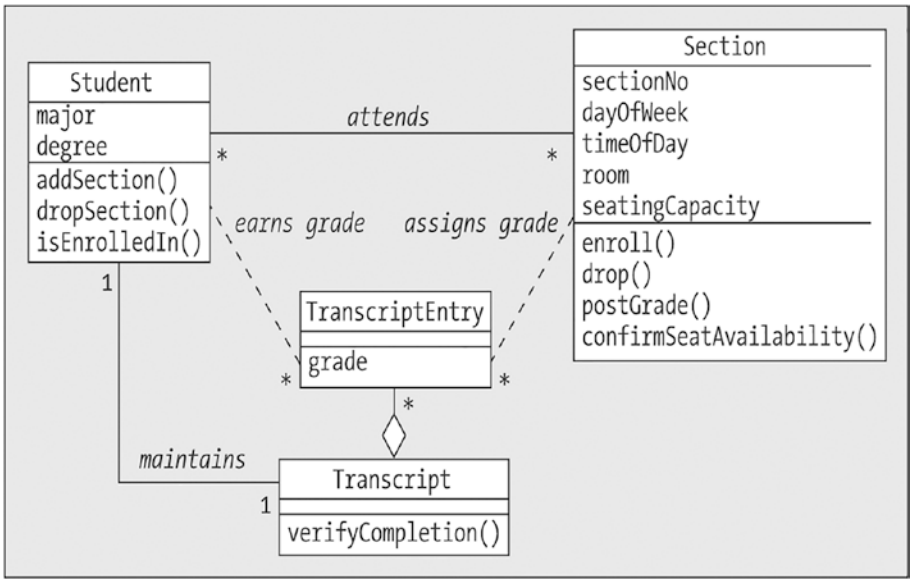


Figure 14-8. *An alternative UML representation of the* `TranscriptEntry` *association class*

And so we'll encode a fifth relationship for the Section class, namely

- ***assigns grade***, a one-to-many association with the TranscriptEntry class

  (You may be wondering whether we should now go back and adjust the Student class to reflect the *earns grade* association with the TranscriptEntry class as a Student class attribute. The decision of whether or not to implement a particular relationship in code depends in part on what we anticipate our usage patterns to be, as discussed in Chapter 10. We'll defer the decision of what to do with *earns grade* until we talk about the TranscriptEntry class in a bit more depth later in this chapter.)

We'll represent these five relationships in terms of Section attributes as follows:

- A Section object need only maintain a handle on one other object for those one-to-many relationships in which Section occupies the "many" end, namely:

```
private Course representedCourse;
private ScheduleOfClasses offeredIn;
private Professor instructor;
```

- For the two situations in which Section needs to maintain handles on ***collections*** of objects—Students and TranscriptEntrys—we are going to employ HashMaps instead of ArrayLists this time. We do so because it is conceivable that we'll have a frequent need to "pluck" a given item from the collection directly, and a HashMap—as a dictionary-type collection—provides a key-based lookup mechanism that is ideal for this purpose.
- For the HashMap of Student object references, we'll use a String representing the Student's Social Security number (ssn) as a key for looking up a Student:

```
// The enrolledStudents HashMap stores Student object references,
// using each Student's ssn as a String key.
private HashMap<String, Student> enrolledStudents;
```

- For the HashMap of TranscriptEntry object references, on the other hand, we'll use a Student object as a whole as a key for looking up that particular Student's TranscriptEntry as issued by this Section:

```
// The assignedGrades HashMap stores TranscriptEntry object
// references, using a reference to the Student to whom it belongs
// as the key.
private HashMap<Student, TranscriptEntry> assignedGrades;
```

The Use of an Enum(eration) Type

In the Section class, we encounter our first use of an enum(eration), which as you learned in Chapter 13 is an excellent means of defining a finite list of symbolic values. In this particular situation, we want to define some status codes that the Section class can use when signaling the outcome of an enrollment attempt.

We discussed the syntax of an enum type in detail in Chapter 13. The following code defines four symbolic values to represent four different outcomes of an enrollment attempt—success, secFull, prereq, and prevEnroll—each of which correlates to a String value as indicated:

```
public enum EnrollmentStatus {
    // Enumerate the values that the enum can assume.
    success("enrollment successful!  :o)"),
    secFull("enrollment failed:  section was full.  :op"),
    prereq("enrollment failed; prerequisites not satisfied.  :op"),
    prevEnroll("enrollment failed; previously enrolled.  :op");

    // This represents the value of an enum instance.
    private final String value;

    // A "constructor" of sorts (used above).
    EnrollmentStatus(String value) {
        this.value = value;
    }
```

```
    // Accessor for the value of an enum instance.
    public String value() {
        return value;
    }
}
```

Let's look at how these values are put to use by studying the `enroll` method of `Section`.

enroll()

This is a very complex method. I'll list the code in its entirety first without discussing it, and then I'll explain it in detail:

```
    public EnrollmentStatus enroll(Student s) {
        // First, make sure that this Student is not already
        // enrolled for this Section, and that he/she has
        // NEVER taken and passed the course before.

        Transcript transcript = s.getTranscript();


        if (s.isCurrentlyEnrolledInSimilar(this) ||
            transcript.verifyCompletion(this.getRepresentedCourse())) {
            return EnrollmentStatus.prevEnroll;
        }

        // If there are any prerequisites for this course,
        // check to ensure that the Student has completed them.

        Course c = this.getRepresentedCourse();
        if (c.hasPrerequisites()) {
            for (Course pre : c.getPrerequisites()) {
                // See if the Student's Transcript reflects
                // successful completion of the prerequisite.

                if (!transcript.verifyCompletion(pre)) {
                    return EnrollmentStatus.prereq;
                }
            }
        }
```

```
        // If the total enrollment is already at the
        // the capacity for this Section, we reject this
        // enrollment request.

        if (!this.confirmSeatAvailability()) {
            return EnrollmentStatus.secFull;
        }

        // If we made it to here in the code, we're ready to
        // officially enroll the Student.

        // Note bidirectionality:  this Section holds
        // on to the Student via the HashMap, and then
        // the Student is given a handle on this Section.

        enrolledStudents.put(s.getSsn(), s);
        s.addSection(this);

        return EnrollmentStatus.success;
    }
```

Note that the return type of this method is declared to be `EnrollmentStatus`—that is, we are going to return an instance of the `EnrollmentStatus enum` as a symbolic value representing the outcome of this enrollment attempt.

We begin by verifying that the `Student` seeking enrollment (represented by arguments) hasn't already enrolled for this `Section` and furthermore that the student has ***never*** taken and successfully completed this `Course` (any `Sections`) in the past. We obtain a handle on the `Student`'s transcript and store it in a locally declared reference variable called `transcript`, because we are going to need to consult with the `Transcript` object twice in this method:

```
    public EnrollmentStatus enroll(Student s) {
        // First, make sure that this Student is not already
        // enrolled for this Section, and that he/she has
        // NEVER taken and passed the course before.

        Transcript transcript = s.getTranscript();
```

We then use an `if` statement to test for either of two conditions: (a) is the `Student` currently enrolled in this `Section` or another `Section` of the same `Course`, and/or (b) does the `Student`'s `Transcript` indicate successful prior completion of the `Course` that is represented by this `Section`?

```
if (s.isCurrentlyEnrolledInSimilar(this) ||
    transcript.verifyCompletion(this.getRepresentedCourse())) {
```

Because we have need to use this `Course` object only once in this method, we don't bother to save the handle returned to us by the `getRepresentedCourse` method in a variable; we just nest the invocation of this method within the call to `verifyCompletion`, so that the `Course` object can be retrieved by the former and immediately passed along as an argument to the latter.

Whenever we encounter a `return` statement midway through a method as we have here, the method's execution will immediately terminate without running to completion:

```
    return EnrollmentStatus.prevEnroll;
}
```

Note our use of `EnrollmentStatus.prevEnroll`, one of the symbolic values defined by the `EnrollmentStatus enum`, as a return value. Declaring and using such standardized values is a great way to communicate status back to client code.

Next, we check to see if the `Student` has satisfied the prerequisites for this `Section`, if there are any. We use the `Section`'s `getRepresentedCourse` method to obtain a handle on the `Course` object that this `Section` represents and then invoke the `hasPrerequisites` method on that `Course` object. If the result returned is `true`, then we know that there are prerequisites to be checked:

```
// If there are any prerequisites for this course,
// check to ensure that the Student has completed them.

Course c = this.getRepresentedCourse();
if (c.hasPrerequisites()) {
```

If there are indeed prerequisites for this `Course`, we use the `getPrerequisites` method defined by the `Course` class to obtain a collection of all prerequisite `Courses` and iterate through this collection via a `for` loop:

```
    for (Course pre : c.getPrerequisites()) {
```

For each Course object reference pre that we extract from the collection, we invoke the verifyCompletion method on the Student's Transcript object, passing in the prerequisite Course object reference pre. We haven't taken a look at the inner workings of the Transcript class yet, so for now, all we need to know about verifyCompletion is that it will return a value of true if the Student has indeed successfully taken and passed the Course in question; otherwise, it will return a value of false. We want to take action in situations where a prerequisite was ***not*** satisfied, so we use the unary negation operator (!) in front of the expression to indicate that we want the if test to succeed if the method call returns a value of false:

```
            // See if the Student's Transcript reflects
            // successful completion of the prerequisite.
            if (!transcript.verifyCompletion(pre)) {
                return EnrollmentStatus.prereq;
            }
        }
```

If the student is found to have not satisfied any one of the prerequisites, the return statement is triggered, and we return the status value EnrollmentStatus.prereq. If, on the other hand, we make it through the prerequisite check without triggering the return statement, the next step in this method is to verify that there is still available seating in the Section. We return the status value EnrollmentStatus.secFull value if there is not:

```
        // If the total enrollment is already at the
        // the capacity for this Section, we reject this
        // enrollment request.
        if (!this.confirmSeatAvailability()) {
            return EnrollmentStatus.secFull;
        }
```

Finally, if we've made it through ***both*** of the tests unscathed, we're ready to officially enroll the Student. We use the HashMap class's put method to insert the Student reference into the enrolledStudents HashMap, invoking the getSsn method on the Student to retrieve the String value of its ssn attribute, which we pass in as the key value:

```
        enrolledStudents.put(s.getSsn(), s);
```

To achieve bidirectionality of the link between a `Student` and a `Section`, we then turn around and invoke the `addSection` method on the `Student` object reference, passing it a handle on ***this*** `Section`. We then return the value `EnrollmentStatus.success` to signal successful enrollment:

```
            s.addSection(this);
            return EnrollmentStatus.success;
        }
```

drop()

The `drop` method of `Section` performs the reverse operation of `enroll`. We start by verifying that the `Student` in question is indeed enrolled in this `Section`, since we can't drop a `Student` who isn't enrolled in the first place:

```
    public boolean drop(Student s) {
        // We may only drop a student if he/she is enrolled.
        if (!s.isEnrolledIn(this)) return false;
```

If the student truly is enrolled, we use the `HashMap` class's `remove` method to locate and delete the `Student` reference, again via its `ssn` attribute value:

```
        else {
            // Find the student in our HashMap, and remove it.
            enrolledStudents.remove(s.getSsn());
```

In the interest of bidirectionality, we invoke the `dropSection` method on the `Student`, as well, to get rid of the handles at ***both*** ends of the link:

```
            // Note bidirectionality.
            s.dropSection(this);

            return true;
        }
    }
```

postGrade( )

The postGrade method is used to assign a grade to a Student by creating a TranscriptEntry object to link this Section to the Student being assigned a grade.

We begin by validating that the proposed grade to be assigned to Student s (both s and grade are passed in as arguments to the postGrade method) is properly formed by calling a static utility method, validateGrade, that is defined by the TranscriptEntry class. The business rules that govern what constitutes a "valid" grade representation are encoded within that method; these business rules will be revealed when we explore the TranscriptEntry class later in this chapter. For the time being, all we need know is that, if the validateGrade method rejects the proposed grade by returning a value of false, we in turn exit the postGrade method, returning a value of false to client code to indicate that the request to post a grade for Student s has been rejected:

```
public boolean postGrade(Student s, String grade) {
    // First, validate that the grade is properly formed by calling
    // a utility method provided by the TranscriptEntry class.

    if (!TranscriptEntry.validateGrade(grade)) return false;
```

Next, to ensure that we aren't inadvertently trying to assign a grade to a particular Student more than once, we first check the assignedGrades HashMap to see if it already contains an entry for this Student. If the get method call on the HashMap returns anything but null, then we know a grade has already been posted for this Student, and we terminate execution of the method, once again returning a value of false:

```
    // Make sure that we haven't previously assigned a
    // grade to this Student by looking in the HashMap
    // for an entry using this Student as the key.  If
    // we discover that a grade has already been assigned,
    // we return a value of false to indicate that
    // we are at risk of overwriting an existing grade.
    // (A different method, eraseGrade(), can then be written
    // to allow a Professor to change his/her mind.)

    if (assignedGrades.get(s) != null) return false;
```

Assuming that a grade was not previously assigned, we invoke the appropriate constructor to create a new TranscriptEntry object. As you will see when we study the inner workings of the TranscriptEntry class, this object will maintain handles on both the Student to whom a grade has been assigned and the Section for which the grade was assigned:

```
        // First, we create a new TranscriptEntry object.  Note
        // that we are passing in a reference to THIS Section,
        // because we want the TranscriptEntry object,
        // as an association class ..., to maintain
        // "handles" on the Section as well as on the Student.
        // (We'll let the TranscriptEntry constructor take care of
        // linking this T.E. to the correct Transcript.)

        TranscriptEntry te = new TranscriptEntry(s, grade, this);
```

To enable this latter link to be bidirectional, we also store a handle on the TranscriptEntry object in the Section's assignedGrades HashMap for this purpose:

```
        // Then, we "remember" this grade because we wish for
        // the connection between a T.E. and a Section to be
        // bidirectional.
        assignedGrades.put(s, te);

        return true;
    }
```

getGrade( )

The getGrade method uses the Student reference passed in as an argument to this method as a lookup key for the assignedGrades HashMap, to retrieve the TranscriptEntry stored therein for this Student:

```
public String getGrade(Student s) {
        String grade = null;

        // Retrieve the associated TranscriptEntry object for this specific
        // student from the assignedGrades HashMap, if one exists, and in turn
        // retrieve its assigned grade.

        TranscriptEntry te = assignedGrades.get(s);
```

If a TranscriptEntry is found, we use its getGrade method to retrieve the actual grade (as a String value) so that it may be returned by this method:

```
        if (te != null) {
            grade = te.getGrade();
        }
```

Otherwise, we return a value of null to signal that no grade has yet been assigned for the Student of interest:

```
        // If we found no TranscriptEntry for this Student, a null value
        // will be returned to signal this.

        return grade;
    }
```

confirmSeatAvailability()

The confirmSeatAvailability method called from within enroll is an internal "housekeeping" method. By declaring it to have private vs. public visibility, we restrict its use so that only other methods of the Section class may invoke it:

```
    private boolean confirmSeatAvailability() {
        if (enrolledStudents.size() < this.getSeatingCapacity()) return true;
        else return false;
    }
```

Delegation Revisited

In discussing the Student class, I briefly mentioned the displayCourseSchedule method as a complex example of delegation and promised to come back and discuss it further.

What are the "raw materials"—data—available for an object to use when it is responding to a service request by executing one of its methods? By way of review, an object has at its disposal the following data sources:

- Simple data and/or object references (handles) that have been ***encapsulated as attributes*** within the object itself
- Simple data and/or object references that are ***passed in as arguments*** in the method signature

- Data that is made available ***globally*** to the application as `public static` attributes of some other class
- Data that ***can be requested*** from any of the objects that this object has a handle on

It is this last source of data—data available by ***collaborating with other objects***—that is going to play a particularly significant role in implementing the `displayCourseSchedule` method for the `Student` class.

Let's say we want the `displayCourseSchedule` method to display the following information for each `Section` that a `Student` is currently enrolled in:

```
Course No.:
Section No.:
Course Name:
Meeting Day and Time Held:
Room Location:
Professor's Name:
```

For example:

```
Course Schedule for Fred Schnurd
  Course No.:  CMP101
  Section No.:  2
  Course Name:  Beginning Computer Technology
  Meeting Day and Time Held:  W - 6:10 - 8:00 PM
  Room Location:  GOVT202
  Professor's Name:  Claudio Cioffi
  -----
  Course No.:  ART101
  Section No.:  1
  Course Name:  Beginning Basketweaving
  Meeting Day and Time Held:  M - 4:10 - 6:00 PM
  Room Location:  ARTS25
  Professor's Name:  Snidely Whiplash
  -----
```

Let's start by looking at the attributes of the Student class, to see which of this information is readily available to us. Student inherits from Person

```
private String name;
private String ssn;
```

and adds

```
private String major;
private String degree;
private Transcript transcript;
private ArrayList<Section> attends;
```

Let's begin to write the method. By stepping through the attends ArrayList, we can gain access to Section objects one by one:

```
public void displayCourseSchedule() {
  // Display a title first.
  System.out.println("Course Schedule for " + this.getName());

  // Step through the ArrayList of Section objects,
  // processing these one by one.
  for (Section s : attends) {
    // Now what goes here????
    // We must create the rest of the method ...
  }
}
```

Now that we have the beginnings of the method, let's determine how to fill in the gap in the preceding code.

Looking at all of the method headers declared for the Section class as evidence of the services that a Section object can perform, we see that several of these can immediately provide us with useful pieces of information relative to our mission of displaying a Student's course schedule—namely, those that are flagged (***) in the following code:

```
public void setSectionNo(int no)
public int getSectionNo() ***
public void setDayOfWeek(char day)
public char getDayOfWeek() ***
```

```
  public void setTimeOfDay(String time)
  public String getTimeOfDay() ***
  public void setInstructor(Professor prof)
  public Professor getInstructor()
  public void setRepresentedCourse(Course c)
  public Course getRepresentedCourse()
  public void setRoom(String r)
  public String getRoom() ***
  public void setSeatingCapacity(int c)
  public int getSeatingCapacity()
  public void setOfferedIn(ScheduleOfClasses soc)
  public ScheduleOfClasses getOfferedIn()
  public String toString()
  public int enroll(Student s)
  public boolean drop(Student s)
  public int getTotalEnrollment()
  public void display()
  public void displayStudentRoster()
  public String getGrade(Student s)
  public boolean postGrade(Student s, String grade)
  public boolean successfulCompletion(Student s)
  public boolean isSectionOf(Course c)
```

Let's put the four designated methods to use, and where we can't yet fill the gap completely, we'll insert "???" as a placeholder:

```
public void displayCourseSchedule() {
  // Display a title first.
  System.out.println("Course Schedule for " + this.getName());

  // Step through the ArrayList of Section objects,
  // processing these one by one.
  for (Section s : attends) {
    // Since the attends ArrayList contains Sections that the
    // Student took in the past as well as those for which
    // the Student is currently enrolled, we only want to
```

```
    // report on those for which a grade has not yet been
    // assigned.
    if (s.getGrade(this) == null) {
      System.out.println("\tCourse No.:  " + ???
      System.out.println("\tSection No.:  " + s.getSectionNo());
      System.out.println("\tCourse Name:  " + ???
      System.out.println("\tMeeting Day and Time Held:  "  +
        s.getDayOfWeek() + " - " + s.getTimeOfDay());
      System.out.println("\tRoom Location:  " + s.getRoom());
      System.out.println("\tProfessor's Name:  " + ???
      System.out.println("\t-----");
    }
  }
}
```

Now, what about the remaining "holes"? Two of the `Section` methods

```
public Professor getInstructor()
public Course getRepresentedCourse()
```

will each hand us yet another object that we can "talk to"—namely, the `Professor` who teaches this `Section` and the `Course` that this `Section` represents. Let's now look at what ***these*** objects can perform in the way of services:

- A `Professor` object can perform the following services (the first four are inherited from `Person`). Again, those that seem relevant to the mission we're trying to accomplish with the `displayCourseSchedule` method of the `Student` class are flagged (***):

```
public void setName(String n)
public String getName()        ***
public void setSsn(String ssn)
public String getSsn()
public void display()
public void setTitle(String title)
public String getTitle()
public void setDepartment(String dept)
public String getDepartment()
```

```
public void display()
public String toString()
public void displayTeachingAssignments()
public void agreeToTeach(Section s)
```

- A Course object can perform these services (the relevant methods are flagged [***]):

```
public void setCourseNo(String cNo)
public String getCourseNo() ***
public void setCourseName(String cName)
public String getCourseName() ***
public void setCredits(double c)
public double getCredits()
public void display()
public String toString()
public void addPrerequisite(Course c)
public boolean hasPrerequisites()
public Collection<Course> getPrerequisites()
public Section scheduleSection(char day, String time, String room,
  int capacity)
```

If we bring all of the flagged (***) methods to bear, we can wrap up the displayCourseSchedule method of the Student class as follows:

```
public void displayCourseSchedule() {
  // Display a title first.
  System.out.println("Course Schedule for " + getName());

  // Step through the ArrayList of Section objects,
  // processing these one by one.
  for (Section s : attends) {
    // Since the attends ArrayList contains Sections that the
    // Student took in the past as well as those for which
    // the Student is currently enrolled, we only want to
    // report on those for which a grade has not yet been
    // assigned.
```

```
      if (s.getGrade(this) == null) {
        System.out.println("\tCourse No.:  " +
          s.getRepresentedCourse().getCourseNo());
        System.out.println("\tSection No.:  " + s.getSectionNo());
        System.out.println("\tCourse Name:  " +
          s.getRepresentedCourse().getCourseName());
        System.out.println("\tMeeting Day and Time Held:  "  +
          s.getDayOfWeek() + " - " + s.getTimeOfDay());
        System.out.println("\tRoom Location:  "  +
          s.getRoom());
        System.out.println("\tProfessor's Name:  " +
          s.getInstructor().getName());
        System.out.println("\t-----");
      }
    }
  }
```

This method is a classic example of ***delegation***:

- We start out asking a `Student` object to do something for us—namely, to display the `Student`'s course schedule.
- The `Student` object in turn has to talk to the `Section` objects representing sections that the student is enrolled in, asking each of them to perform some of their services (methods).
- The `Student` object also has to ask those `Section` objects to hand over references to the `Professor` and `Course` objects that the `Section` objects know about, in turn asking ***them*** to perform some of their services.

This multitiered collaboration is depicted conceptually in Figure 14-9.

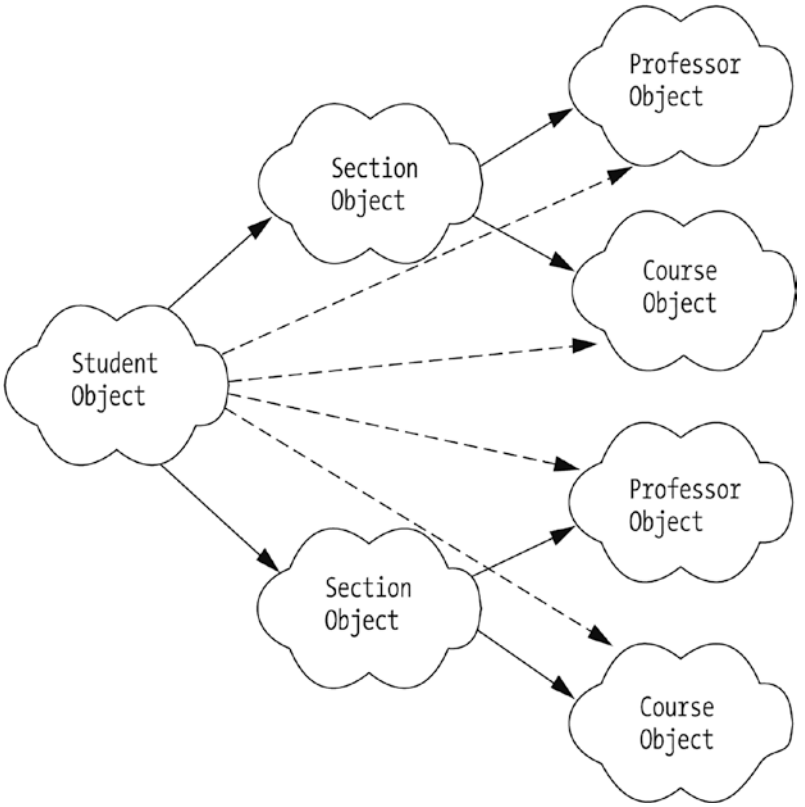


Figure 14-9. *A multitiered collaboration among objects*

The ScheduleOfClasses Class

Figure 14-10 shows the UML representation of the `ScheduleOfClasses` class. The sections that follow provide more detail about this class.

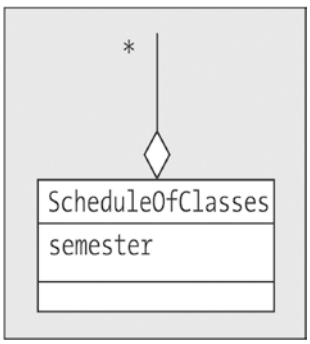


Figure 14-10. *The `ScheduleOfClasses` class*

ScheduleOfClasses Attributes

The ScheduleOfClasses class is a fairly simple class that serves as an example of how we might wish to encapsulate a collection object within some other class. It consists of only two attributes: a simple String representing the semester for which the schedule is valid (e.g., "SP2005" for the Spring 2005 semester) and a HashMap used to maintain handles on all of the Sections that are being offered that semester:

```
private String semester;

// This HashMap stores Section object references, using
// a String concatenation of course no. and section no. as the
// key, for example, "MATH101 - 1".
private HashMap<String, Section> sectionsOffered;
```

Aside from a simple constructor, a display method, and accessor methods for the attributes, the ScheduleOfClasses class declares three relatively simple methods, as follows.

addSection( )

This method is used to add a Section object to the HashMap and then to bidirectionally link this ScheduleOfClasses object back to the Section:

```
public void addSection(Section s) {
    // We formulate a key by concatenating the course no.
    // and section no., separated by a hyphen.
    String key = s.getRepresentedCourse().getCourseNo() +
             " - " + s.getSectionNo();
    sectionsOffered.put(key, s);

    // Bidirectionally link the ScheduleOfClasses back to the Section.
    s.setOfferedIn(this);
}
```

findSection()

This is a convenience method that is used to look up a `Section` in the encapsulated collection using the full section number that is passed in as the lookup key:

```
// The full section number is a concatenation of the
// course no. and section no., separated by a hyphen;
// e.g., "ART101 - 1".

public Section findSection(String fullSectionNo) {
    return sectionsOffered.get(fullSectionNo);
}
```

isEmpty()

This is another convenience method that is used to determine whether the encapsulated collection is empty. Internally, it delegates the work of making such a determination to the `sectionsOffered` collection:

```
public boolean isEmpty() {
    if (sectionsOffered.size() == 0) return true;
    else return false;
}
```

The TranscriptEntry Association Class (Static Methods)

Figure 14-11 shows the UML representation of the `TranscriptEntry` class. The sections that follow provide more detail about this class.

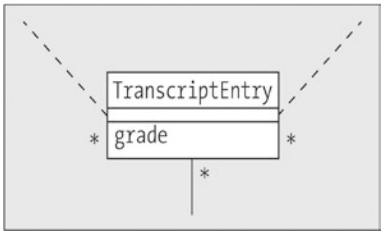


Figure 14-11. *The `TranscriptEntry` class*

TranscriptEntry Attributes

As you saw earlier in this chapter, the `TranscriptEntry` class has one simple attribute, `grade`, and maintains associations with three other classes (see Figure 14-12):

- ***earns grade***, a one-to-many association with `Student`
- ***assigns grade***, a one-to-many association with `Section`
- An unnamed, one-to-many aggregation with the `Transcript` class

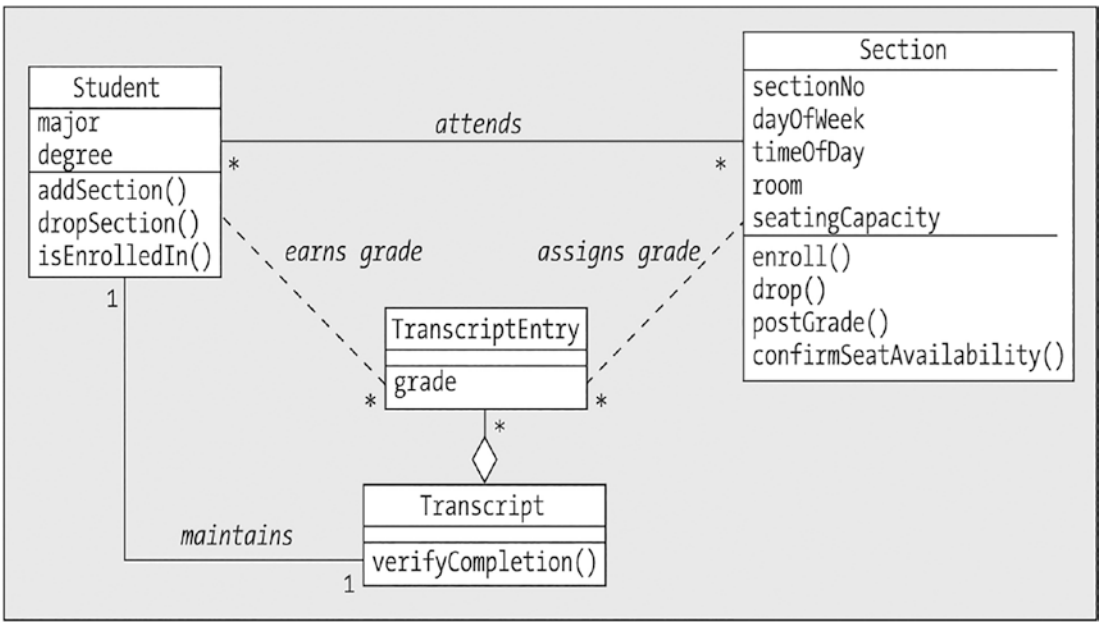


Figure 14-12. *`TranscriptEntry` maintains numerous relationships with other SRS classes*

`TranscriptEntry` is at the "many" end of all of these associations, and so it only needs to maintain a single handle on each type of object—no collection attributes are required:

```
private String grade;
private Student student;
private Section section;
private Transcript transcript;
```

TranscriptEntry Constructor

The constructor for this class does most of the work of maintaining all of these relationships.

Via a call to `setStudent`, it stores the associated `Student` object's handle in the appropriate attribute:

```
//----------------
// Constructor(s).
//----------------

public TranscriptEntry(Student s, String grade, Section se) {
    this.setStudent(s);
```

Note that we have chosen ***not*** to maintain the *earns grade* association bidirectionally; that is, we have provided no code in either this or the `Student` class to provide the `Student` object with a handle on this `TranscriptEntry` object. This decision was based upon the fact that we don't expect a `Student` to ever have to manipulate `TranscriptEntry` objects directly. Every `Student` object has an indirect means of reaching all of its `TranscriptEntry` objects, via the handle that a `Student` object maintains on its `Transcript` object as a whole and the handles that the `Transcript` object in turn maintains on its `TranscriptEntry` objects. You might think that giving a `Student` object the ability to directly pull a given `TranscriptEntry` might be useful when we want to determine the grade that the `Student` earned for a particular `Section`, but we have provided an alternative means of doing so, via the `Section` class's `getGrade` method.

Even though it may not appear so, we are maintaining the *assigns grade* association with `Section` bidirectionally. We see only half of the "handshake" in the `TranscriptEntry` constructor:

```
    this.setSection(se);
```

But recall that when we looked at the `postGrade` method of the `Section` class, we discussed the fact that `Section` was responsible for maintaining the bidirectionality of this association. When the `Section`'s `postGrade` method invokes the `TranscriptEntry` constructor, the `Section` object is returned a handle on this `TranscriptEntry` object, which it stores in the appropriate attribute. So we only need to worry about the second half of this "handshake" here in `TranscriptEntry`.

On the other hand, the TranscriptEntry object has full responsibility for maintaining the bidirectionality of the association between itself and the Transcript object:

```
        // Obtain the Student's transcript ...
        Transcript t = s.getTranscript();

        // ... and then link the Transcript and the TranscriptEntry
        // together bidirectionally.
        this.setTranscript(t);
        t.addTranscriptEntry(this);
    }
```

validateGrade(), passingGrade( )

The TranscriptEntry class provides our first SRS example of public static methods. It declares two methods, validateGrade and passingGrade, that may be invoked as utility methods on the TranscriptEntry class from anywhere in the SRS application:

- The first is used to validate whether or not a particular string—say, "B+"—is a valid grade. Here we see the business rules disclosed for what constitutes such a grade:

```
public static boolean validateGrade(String grade) {
    boolean outcome = false;

    if (grade.equals("F") ||
        grade.equals("I")) {
        outcome = true;
    }

    if (grade.startsWith("A") ||
        grade.startsWith("B") ||
        grade.startsWith("C") ||
        grade.startsWith("D")) {
        if (grade.length() == 1) outcome = true;
        else if (grade.length() == 2) {
            if (grade.endsWith("+") ||
                grade.endsWith("-")) {
```

```
                outcome = true;
            }
        }
    }

    return outcome;
}
```

- The second is used to determine whether or not a particular string—say "D+"—is a ***passing*** grade. A slightly different set of business rules is applied in this case:

```
public static boolean passingGrade(String grade) {
    boolean outcome = false;

    // First, make sure it is a valid grade.
    if (validateGrade(grade)) {
        // Next, make sure that the grade is a D or better.
        if (grade.startsWith("A") ||
            grade.startsWith("B") ||
            grade.startsWith("C") ||
            grade.startsWith("D")) {
            outcome = true;
        }
    }

    return outcome;
}
```

As we discussed in Chapter 7, `public static` methods can be invoked on the hosting class as a whole—in other words, an object needn't be instantiated in order to use these methods.

The Transcript Class

Figure 14-13 shows the UML representation of the Transcript class. The sections that follow provide more detail about this class.

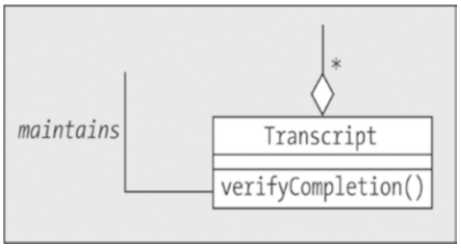


Figure 14-13. *The Transcript class*

Transcript Attributes

The Transcript class participates in two relationships:

- ***maintains***, a one-to-one association with Student
- An unnamed, one-to-many aggregation with TranscriptEntry

The SRS class diagram does not call out any other attributes for the Transcript class, so we only encode these two:

```
private ArrayList<TranscriptEntry> transcriptEntries;
private Student studentOwner;
```

verifyCompletion( )

The Transcript class has one particularly interesting method, verifyCompletion, which is used to determine whether or not the Transcript contains evidence that a particular Course requirement has been satisfied. This method steps through the ArrayList of TranscriptEntries maintained by the Transcript object:

```
public boolean verifyCompletion(Course c) {
  boolean outcome = false;

  // Step through all TranscriptEntries, looking for one
  // that reflects a Section of the Course of interest.
  for (TranscriptEntry te : transcriptEntries) {
```

For each entry, it obtains a handle on the `Section` object represented by this entry and then invokes the `isSectionOf` method on that object to determine whether or not that `Section` represents the `Course` of interest:

```
Section s = te.getSection();

if (s.isSectionOf(c)) {
```

Assuming that the `Section` is indeed relevant, the method next uses the `static` `passingGrade` method of the `TranscriptEntry` class to determine whether or not the grade earned in this `Section` was a passing grade. If it was a passing grade, we can terminate the loop immediately, since we need to find only ***one*** example of a passing grade for the `Course` of interest in order to return a `true` outcome from this method:

```
        // Ensure that the grade was high enough.
        if (TranscriptEntry.passingGrade(te.getGrade())) {
          outcome = true;

          // We've found one, so we can afford to
          // terminate the loop now.
          break;
        }
      }
    }

    return outcome;
  }
}
```

The SRS Driver Program

Now that we've coded all of the classes called for by our model of the SRS, we need a way to test these. We could wait to put our application through its paces until we've built a GUI front end; however, it would be nice to know sooner rather than later that our core classes are working properly. One very helpful technique for doing so is to write a command-line–driven program to instantiate objects of varying types and to invoke their critical methods, displaying the results to the command-line window for us to inspect.

We'll develop just such a program by creating a class called `SRS` with a `main` method that will serve as our test "driver."

Public Static Attributes

We are going to instantiate some `Professor`, `Student`, `Course`, and `Section` objects in this program, so we need a way to organize references to these objects. We'll create collection objects as attributes of the `SRS` class to hold each of these different object types. While we're at it, we'll declare them to be `public static` attributes, which means that we're making these main object collections globally available to the entire application; these can then be accessed throughout the SRS application as `SRS.`*`collectionName`* (e.g., `SRS.faculty`):

```
// We can effectively create "global" data by declaring
// collections of objects as public static attributes in
// the main class;

public static ArrayList<Professor> faculty;
public static ArrayList<Student> studentBody;
public static ArrayList<Course> courseCatalog;

// The next collection -- of Section object references -- is
   encapsulated
// within a special-purpose class by virtue of how we modeled the
   SRS in UML;
// note that we could have encapsulated the preceding three
   collections in
// similar fashion.

public static ScheduleOfClasses scheduleOfClasses =
          new ScheduleOfClasses("SP2005");
```

The SRS `ScheduleOfClasses` class serves as a collection point for `Section` objects; for the other types of objects, we use simple `ArrayLists`, although we could go ahead and design classes comparable to `ScheduleOfClasses` to serve as encapsulated collections, perhaps named `Faculty`, `StudentBody`, and `CourseCatalog`, respectively. We don't need a collection for `Transcript` objects—we'll get to these via the handles that `Student` objects maintain—or for `TranscriptEntry` objects (we'll get to these via the `Transcript` objects themselves).

The main Method

We'll now dive into the `main` method for the SRS driver class. We'll start by declaring reference variables for each of the four main object types:

```
public static void main(String[] args) {
    Professor p1, p2, p3;
    Student s1, s2, s3;
    Course c1, c2, c3, c4, c5;
    Section sec1, sec2, sec3, sec4, sec5, sec6, sec7;
```

We'll then use their various constructors to fabricate object instances, storing handles in the appropriate collections:

```
    // Create various objects by calling the appropriate
    // constructors.  (We'd normally be reading in such data
    // from a database or file ...)

    // -----------
    // Professors.
    // -----------

    p1 = new Professor("Jacquie Barker", "123-45-6789",
               "Adjunct Professor", "Information Technology");

    p2 = new Professor("Claudio Cioffi", "567-81-2345",
               "Full Professor", "Computational Social Sciences");

    p3 = new Professor("Snidely Whiplash", "987-65-4321",
               "Full Professor", "Physical Education");

    // Add these to the appropriate ArrayList.
    faculty = new ArrayList<Professor>();
    faculty.add(p1);
    faculty.add(p2);
    faculty.add(p3);

    // ---------
    // Students.
    // ---------
```

```
s1 = new Student("Joe Blow", "111-11-1111", "Math", "M.S.");

s2 = new Student("Fred Schnurd", "222-22-2222",
         "Information Technology", "Ph. D.");

s3 = new Student("Mary Smith", "333-33-3333", "Physics", "B.S.");

// Add these to the appropriate ArrayList.
studentBody = new ArrayList<Student>();
studentBody.add(s1);
studentBody.add(s2);
studentBody.add(s3);

// --------
// Courses.
// --------

c1 = new Course("CMP101",
        "Beginning Computer Technology", 3.0);

c2 = new Course("OBJ101",
        "Object Methods for Software Development", 3.0);

c3 = new Course("CMP283",
        "Higher Level Languages (Java)", 3.0);

c4 = new Course("CMP999",
        "Living Brain Computers", 3.0);

c5 = new Course("ART101",
        "Beginning Basketweaving", 3.0);

// Add these to the appropriate ArrayList.
courseCatalog = new ArrayList<Course>();
courseCatalog.add(c1);
courseCatalog.add(c2);
courseCatalog.add(c3);
courseCatalog.add(c4);
courseCatalog.add(c5);
```

We use the addPrerequisite method of the Course class to interrelate some of the Courses, so that c1 is a prerequisite for c2, c2 for c3, and c3 for c4. The only Courses that do not specify prerequisites in our test case are c1 and c5:

```
// Establish some prerequisites (c1 => c2 => c3 => c4).
c2.addPrerequisite(c1);
c3.addPrerequisite(c2);
c4.addPrerequisite(c3);
```

To create Section objects, we take advantage of the Course class's scheduleSection method, which, as you may recall, contains an embedded call to a Section class constructor. Each invocation of scheduleSection returns a handle to a newly created Section object, which we store in the appropriate collection:

```
// ---------
// Sections.
// ---------

// Schedule sections of each Course by calling the
// scheduleSection method of Course (which internally
// invokes the Section constructor).

sec1 = c1.scheduleSection('M', "8:10 - 10:00 PM", "GOVT101", 30);

sec2 = c1.scheduleSection('W', "6:10 - 8:00 PM", "GOVT202", 30);

sec3 = c2.scheduleSection('R', "4:10 - 6:00 PM", "GOVT105", 25);

sec4 = c2.scheduleSection('T', "6:10 - 8:00 PM", "SCI330", 25);

sec5 = c3.scheduleSection('M', "6:10 - 8:00 PM", "GOVT101", 20);

sec6 = c4.scheduleSection('R', "4:10 - 6:00 PM", "SCI241", 15);

sec7 = c5.scheduleSection('M', "4:10 - 6:00 PM", "ARTS25", 40);

// Add these to the Schedule of Classes.
scheduleOfClasses.addSection(sec1);
scheduleOfClasses.addSection(sec2);
scheduleOfClasses.addSection(sec3);
scheduleOfClasses.addSection(sec4);
```

```
scheduleOfClasses.addSection(sec5);
scheduleOfClasses.addSection(sec6);
scheduleOfClasses.addSection(sec7);
```

Next, we use the agreeToTeach method declared for the Professor class to assign Professors to Sections:

```
// Recruit a professor to teach each of the sections.

p3.agreeToTeach(sec1);
p2.agreeToTeach(sec2);
p1.agreeToTeach(sec3);
p3.agreeToTeach(sec4);
p1.agreeToTeach(sec5);
p2.agreeToTeach(sec6);
p3.agreeToTeach(sec7);
```

We then simulate student registration by having Students enroll in the various Sections using the enroll method. Recall that this method returns one of a set of predefined status values as defined by our EnrollmentStatus enum(eration), and so in order to display which status is returned in each case, we created a "housekeeping" reportStatus method solely for the purpose of formatting an informational message:

```
System.out.println("================================");
System.out.println("Student registration has begun!");
System.out.println("================================");
System.out.println();

// Simulate students attempting to enroll in sections of
// various courses.

System.out.println("Student " + s1.getName() +
           " is attempting to enroll in " +
           sec1.toString());

EnrollmentStatus status = sec1.enroll(s1);
this.reportStatus(status);
```

Since the preceding three lines of code—the `println`, `enroll`, and `reportStatus` method calls—are going to be repeated multiple times, we have written a "housekeeping" method called `attemptToEnroll` that does these three things and will use the more concise `attemptToEnroll(...)` syntax for the remainder of this program. (The `attemptToEnroll` method is discussed separately shortly.)

```
// Try concurrently enrolling the same Student in a
   different Section
// of the SAME Course!  This should fail.
attemptToEnroll(s1, sec2);

// This enrollment request should be fine ...
attemptToEnroll(s2, sec2);

// ... but here, the student in question hasn't satisfied the
// prerequisities, so the enrollment request should be rejected.
attemptToEnroll(s2, sec3);

// These requests should both be fine.
attemptToEnroll(s2, sec7);
attemptToEnroll(s3, sec1);

// When the dust settles, here's what folks wound up
// being SUCCESSFULLY registered for:
//
// sec1:  s1, s3
// sec2:  s2
// sec7:  s2
```

Next, we simulate the assignment of grades at the end of the semester by invoking the `postGrade` method for each `Student`-`Section` combination:

```
// Semester is finished (boy, that was quick!). Professors
// assign grades for specific students.
sec1.postGrade(s1, "C+");
sec1.postGrade(s3, "A");
sec2.postGrade(s2, "B+");
sec7.postGrade(s2, "A-");
```

Finally, we put our various display methods to good use by displaying the internal state of the various objects that we created—in essence, an "object dump":

```
        // Let's see if everything got set up properly
        // by calling various display() methods.

        System.out.println("====================");
        System.out.println("Schedule of Classes:");
        System.out.println("====================");
        System.out.println();
        scheduleOfClasses.display();

        System.out.println("======================");
        System.out.println("Professor Information:");
        System.out.println("======================");
        System.out.println();
        p1.display();
        p2.display();
        p3.display();

        System.out.println("====================");
        System.out.println("Student Information:");
        System.out.println("====================");
        System.out.println();
        s1.display();
        s2.display();
        s3.display();
    }
```

Here is the attemptToEnroll housekeeping method mentioned earlier, along with the reportStatus method that it in turn uses:

```
    private static void reportStatus(EnrollmentStatus s) {
        System.out.println("Status:  " + s.value());
        System.out.println();
    }
```

```
    private static void attemptToEnroll(Student s, Section sec) {
        System.out.println("Student " + s.getName() +
                   " is attempting to enroll in " +
                   sec.toString());

        // Utilize one housekeeping method from within another!
        reportStatus(sec.enroll(s));
    }
}
```

When compiled and run, the SRS program produces the following command prompt output:

```
================================
Student registration has begun!
================================

Student Joe Blow is attempting to enroll in CMP101 - 1 - M - 8:10 - 10:00 PM
Status:  enrollment successful!  :o)

Student Joe Blow is attempting to enroll in CMP101 - 2 - W - 6:10 - 8:00 PM
Status:  enrollment failed; previously enrolled.  :op

Student Fred Schnurd is attempting to enroll in CMP101 - 2 - W -
6:10 - 8:00 PM
Status:  enrollment successful!  :o)

Student Fred Schnurd is attempting to enroll in OBJ101 - 1 - R -
4:10 - 6:00 PM
Status:  enrollment failed; prerequisites not satisfied.  :op

Student Fred Schnurd is attempting to enroll in ART101 - 1 - M -
4:10 - 6:00 PM
Status:  enrollment successful!  :o)

Student Mary Smith is attempting to enroll in CMP101 - 1 - M - 8:10 - 10:00 PM
Status:  enrollment successful!  :o)
```

```
====================
Schedule of Classes:
====================

Schedule of Classes for SP2005

Section Information:
    Semester:  SP2005
    Course No.:  ART101
    Section No:  1
    Offered:  M at 4:10 - 6:00 PM
    In Room:  ARTS25
    Professor:  Snidely Whiplash
    Total of 1 students enrolled, as follows:
        Fred Schnurd

Section Information:
    Semester:  SP2005
    Course No.:  CMP283
    Section No:  1
    Offered:  M at 6:10 - 8:00 PM
    In Room:  GOVT101
    Professor:  Jacquie Barker
    Total of 0 students enrolled.

Section Information:
    Semester:  SP2005
    Course No.:  OBJ101
    Section No:  1
    Offered:  R at 4:10 - 6:00 PM
    In Room:  GOVT105
    Professor:  Jacquie Barker
    Total of 0 students enrolled.

Section Information:
    Semester:  SP2005
    Course No.:  CMP101
```

```
    Section No:  2
    Offered:  W at 6:10 - 8:00 PM
    In Room:  GOVT202
    Professor:  Claudio Cioffi
    Total of 1 students enrolled, as follows:
        Fred Schnurd

Section Information:
    Semester:  SP2005
    Course No.:  CMP999
    Section No:  1
    Offered:  R at 4:10 - 6:00 PM
    In Room:  SCI241
    Professor:  Claudio Cioffi
    Total of 0 students enrolled.

Section Information:
    Semester:  SP2005
    Course No.:  CMP101
    Section No:  1
    Offered:  M at 8:10 - 10:00 PM
    In Room:  GOVT101
    Professor:  Snidely Whiplash
    Total of 2 students enrolled, as follows:
        Joe Blow
        Mary Smith

Section Information:
    Semester:  SP2005
    Course No.:  OBJ101
    Section No:  2
    Offered:  T at 6:10 - 8:00 PM
    In Room:  SCI330
    Professor:  Snidely Whiplash
    Total of 0 students enrolled.
```

```
======================
Professor Information:
======================

Person Information:
    Name:  Jacquie Barker
    Soc. Security No.:  123-45-6789
Professor-Specific Information:
    Title:  Adjunct Professor
    Teaches for Dept.:  Information Technology
Teaching Assignments for Jacquie Barker:
    Course No.:  OBJ101
    Section No.:  1
    Course Name:  Object Methods for Software Development
    Day and Time:  R - 4:10 - 6:00 PM
    -----
    Course No.:  CMP283
    Section No.:  1
    Course Name:  Higher Level Languages (Java)
    Day and Time:  M - 6:10 - 8:00 PM
    -----

Person Information:
    Name:  Claudio Cioffi
    Soc. Security No.:  567-81-2345
Professor-Specific Information:
    Title:  Full Professor
    Teaches for Dept.:  Computational Social Sciences
Teaching Assignments for Claudio Cioffi:
    Course No.:  CMP101
    Section No.:  2
    Course Name:  Beginning Computer Technology
    Day and Time:  W - 6:10 - 8:00 PM
    -----
    Course No.:  CMP999
    Section No.:  1
```

```
    Course Name:  Living Brain Computers
    Day and Time:  R - 4:10 - 6:00 PM
    -----

Person Information:
    Name:  Snidely Whiplash
    Soc. Security No.:  987-65-4321
Professor-Specific Information:
    Title:  Full Professor
    Teaches for Dept.:  Physical Education
Teaching Assignments for Snidely Whiplash:
    Course No.:  CMP101
    Section No.:  1
    Course Name:  Beginning Computer Technology
    Day and Time:  M - 8:10 - 10:00 PM
    -----
    Course No.:  OBJ101
    Section No.:  2
    Course Name:  Object Methods for Software Development
    Day and Time:  T - 6:10 - 8:00 PM
    -----
    Course No.:  ART101
    Section No.:  1
    Course Name:  Beginning Basketweaving
    Day and Time:  M - 4:10 - 6:00 PM
    -----

====================
Student Information:
====================

Person Information:
    Name:  Joe Blow
    Soc. Security No.:  111-11-1111
Student-Specific Information:
    Major:  Math
    Degree:  M.S.
```

```
Course Schedule for Joe Blow
Transcript for:  Joe Blow (111-11-1111) [M.S. - Math]
    Semester:        SP2005
    Course No.:      CMP101
    Credits:         3.0
    Grade Received:  C+
    -----

Person Information:
    Name:  Fred Schnurd
    Soc. Security No.:  222-22-2222
Student-Specific Information:
    Major:  Information Technology
    Degree:  Ph. D.
Course Schedule for Fred Schnurd
Transcript for:  Fred Schnurd (222-22-2222) [Ph.D. - Information
Technology]
    Semester:        SP2005
    Course No.:      CMP101
    Credits:         3.0
    Grade Received:  B+
    -----
    Semester:        SP2005
    Course No.:      ART101
    Credits:         3.0
    Grade Received:  A-
    -----

Person Information:
    Name:  Mary Smith
    Soc. Security No.:  333-33-3333
Student-Specific Information:
    Major:  Physics
    Degree:  B.S.
Course Schedule for Mary Smith
Transcript for:  Mary Smith (333-33-3333) [B.S. - Physics]
```

```
Semester:        SP2005
Course No.:      CMP101
Credits:         3.0
Grade Received:  A
-----
```

We've thus demonstrated a successful test of our model! Of course, the SRS driver program could be extended to test various other scenarios; some of the exercises at the end of this chapter suggest ways that you might wish to try doing so.

Summary

You've now seen some ***serious*** Java in action! We've built a command-line–driven version of the SRS application. Although this is not typically how most applications are invoked—most "industrial-strength" applications have GUI front ends—developing such a version is a crucial step in testing our core model classes to ensure that all methods are working properly.

EXERCISES

All of the following exercises involve making modifications/extensions to the SRS code presented in this chapter. If you have not already done so, please download the code from the book's GitHub repository.

1. [*Coding*] Expand the `SRS` class's `main` method to represent a ***second*** semester's worth of course registrations. (Hint: This will require a second instantiation of the `ScheduleOfClasses` class.)
    - Change the grades received by some `Students` in the first semester to failing grades, and then attempt to register the `Student` for a course in the second semester requiring successful completion of one of these failed courses in a previous semester.
    - Try registering a `Student` for a course in the second semester that the `Student` has already successfully completed in the first semester.

2. [*Coding*] Improve the logic of the `addPrerequisite` method of the `Course` class to ensure that a `Course` cannot accidentally be assigned as its own prerequisite.

3. [*Coding*] Improve the logic of the `agreeToTeach` method of the `Professor` class so that a `Professor` cannot accidentally agree to teach two different `Sections` that meet on the same day/at the same time.

4. [*Coding*] Implement a `cancelSection` method for the `Course` class, and then correct the erroneous logic of the `scheduleSection` method having to do with the manner in which `Section` numbers are assigned. (Hint: Consider introducing a static attribute to the `Course` class for this purpose.)

5. [*Coding*] The `enroll` method of the `Section` class does not take into account the fact that a `Student` may simultaneously be registered for a course and its prerequisite. Modify this method to allow for this possibility.

6. [*Coding*] The `postGrade` method of the `Section` class makes mention of the need for an `eraseGrade` method, in the event that a `Professor` wishes to change their mind about the grade that has been issued to a `Student`. Create the `eraseGrade` method.

7. [*Coding*] Modify the `scheduleSection` method of the `Course` class to prevent two `Sections` from being scheduled for the same classroom on the same day/at the same time.

CHAPTER 15

Building a Three-Tier User Driven Application

Congratulations! After completing Chapters 1–14, you now know what many Java developers unfortunately don't get a solid jump-start on, namely:

- How to take full advantage of Java as an object-oriented language
- How to develop the model layer of an application to properly address the requirements of an application at the outset of development

You've also now learned enough Java to be able to build a command-line-driven program. Is this knowledge useful? Absolutely! Command-line-driven (Java) programs are often used to automate "back-end" (server-side) processes as part of a larger enterprise application or system.

That being said, you may ultimately wish to learn how to develop full-blown user-driven applications like the SRS, including

- A graphical user interface, by which users will be able to "drive" the model–most likely a browser-based UI
- A means of persisting the state of our objects from one invocation of the application to the next by storing information about their states (attribute values) in a database

The technologies involved with these aspects of building a user-driven, browser-based application are constantly evolving. Each of these alternative technologies warrants an entire book (or more) to do justice to it and is outside of the scope of this book to address in detail. However, we want you to understand ***conceptually*** what's involved with accomplishing both of these tasks.

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6_15

In this chapter, you will learn

- The typical architecture for a three-tier user-driven application
- An approach to building the persistence tier
- An approach to building the presentation tier
- The role of the controller in the application tier
- The importance of achieving independence among these three tiers

---

A bit of nomenclature: The terms "tier" and "layer" are often used interchangeably.

The terms "data" and "persistence" are often used interchangeably.

The terms "view" and "presentation" are often used interchangeably.

A Three-Tier Architecture

Figure 15-1 illustrates a typical three-tier architecture for a browser-based application.

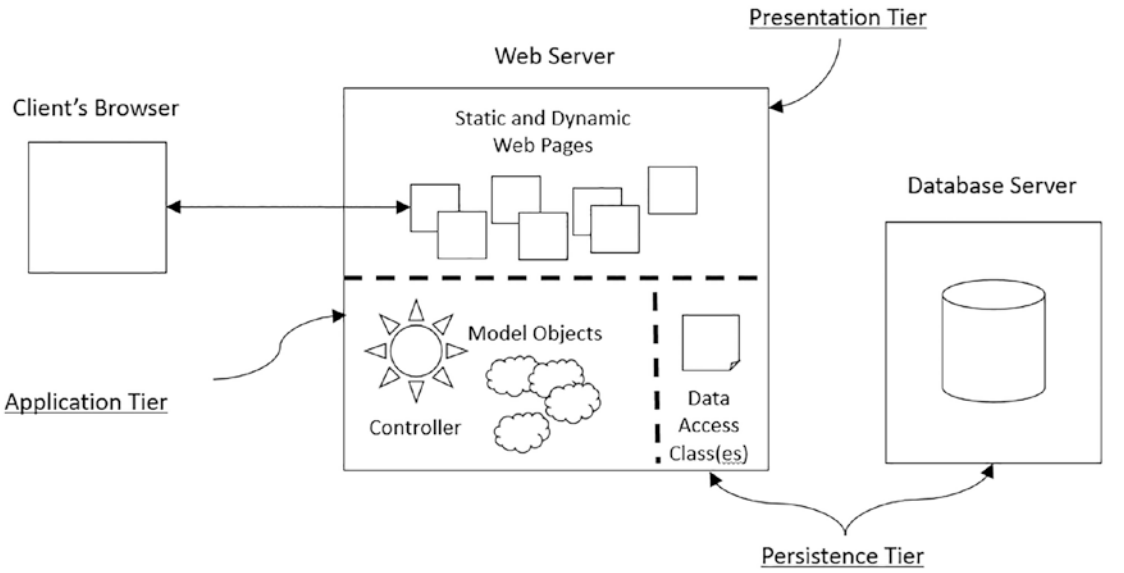


Figure 15-1. *Three tiers of a browser-based application*

- The **presentation tier** consists of a collection of web pages comprising the user interface, residing on a (hardware) web server under control of a (software) web server (e.g., Apache Tomcat).
- The **application tier** consists of the model classes/objects plus **controller** software (more about this in a moment).
- The **persistence tier** consists of one or more classes resident on the web server responsible for communicating with the database that stores (persists) objects' states, which typically resides on a separate hardware server.

What Does the Controller Do?

The controller software is at the heart of the application in that it is responsible for

- Monitoring what the user is doing in the presentation layer–specifically, what actions they are taking in the browser: clicking a button to view their course load, selecting a new section to add to the course load, dropping a course, and so forth
- Communicating with the persistence layer to retrieve the data necessary to reconstitute objects as needed: students, the courses they are registered for, the semester schedule of classes, etc.
- Determining which of the web pages is to be displayed next
- Dynamically generating content for that web page based on the reconstituted objects
- Dispatching the dynamically updated page back to the user's browser

At this point the controller cycle begins anew.

We'll illustrate all of these in pseudocode fashion later in this chapter, but first, let's take a closer conceptual look at how to build the persistence tier.

Building a Persistence/Data Tier

In order for the SRS application to be useful, it needs a way to store the results of objects' interactions–the objects' states–in some sort of permanent (persistent) storage. Depending on what storage technology is chosen for the application–that is, a traditional SQL database, a NoSQL database, etc.–the code for accessing the database to retrieve or store information will vary widely. ***Our goal is to totally encapsulate this logic in a separate new class (or set of classes) without modifying the model layer classes that we've perfected in Chapter 14.***

Let's start by defining an interface that lays out all of the operations necessary to interact with the database; because this is an interface of our own design, we are free to name it as we wish:

```
public interface PersistenceLayerInterface {
      // This method is used to establish a connection with the underlying
      // database when a student logs on; if the log on fails (unknown
      // student or incorrect password, we throw a custom exception of
      // our own design.
      void initialize(String studentid, String password) throws
            InitializationFailureException;

      // This method retrieves all of the data (attribute values)
         related to
      // the designated student, including his/her registered course list,
      // calls our model layer Student constructor to fabricate a Student
      // object, and returns it to the controller.
      Student retrieveStudent(String studentid);

      // This method does the opposite of storeStudent():  passed in a
      // Student object as an argument by the controller, it uses the get
      // methods of the Student to extract attribute values from the object
      //  so as to store them back in the database.
      void storeStudent(Student student);

      // This method retrieves all of the information about sections being
      // offered this semester from the database, formulating a Collection
```

```
      // of Section objects to return to the controller.
      Collection<Section> getSemesterScheduleOfAvailableCourses();

      // etc.
}
```

Next, we create a class that implements the interface, coding each method with vendor-specific database logic to accomplish the task at hand:

```
public class SRSPersistenceLayer implements PersistenceLayerInterface { ...
      // This method is used to establish a connection with the underlying
      // database when a student logs on; if the log on fails (unknown
      // student or incorrect password, we throw a custom exception of
      // our own design.
      void initialize(String studentid, String password) throws
            InitializationFailureException {
            Attempt to establish a connection to the database with this
            username/password combination

            if (connection fails) {
                  throw new InitializationFailureException();
            } else {
                  Store connection in session for subsequent reuse
            }

            return;

      // This method retrieves all of the data (attribute values)
         related to
      // the designated student, including his/her registered course list,
      // calls our model layer Student constructor to fabricate a Student
      // object, and returns it to the controller.
      Student retrieveStudent(String studentid) {
            Select student with indicated ID from database and fully
            populate
            a new Student object, including student's registered courses
            Student s = new Student(...);
```

```
            for (all registered courses found in database for this
            student) {
                  Reconstitute section object
                  Section sec = new Section(...);
                  s.registerForCourse(sec);
            }

            return s;
}
      // This method does the opposite of storeStudent():  passed in a
      // Student object as an argument by the controller, it uses the get
      // methods of the Student to extract attribute values from the object
      // so as to store them back in the database.
      void storeStudent(Student student)
            Details omitted

      // This method retrieves all of the information about sections being
      // offered this semester from the database, formulating a Collection
      // of Section objects to return to the controller.
      Collection<Section> getSemesterScheduleOfAvailableCourses() {
            Collection<Section> semesterSchedule = new ArrayList(section);

            Reconstitute courses retrieved from the database as Section
            Objects and add them to semesterSchedule collection.

            return semesterSchedule;


      // etc.

}
```

We'll put this SRSPersistenceLayer class to work shortly when we talk about the job of the controller, but first let's take a look at building the presentation layer.

Building a Web-Based Presentation Layer

In order to build a web-based presentation layer, we must first determine all of the different web pages that will be necessary to provide the user with the functionality called for by the original requirements. For the SRS, a minimum collection of pages might include those illustrated in Figure 15-2:

- ***A***: An initial logon page
- ***B***: A page that displays the logged-on student's current course load, with buttons to add a course, drop a course, or log off
- ***C***: A page that lists all available sections sorted by department so that the student may search for and select courses of interest, with a button to register for a section or to return to their current course load page view without taking action

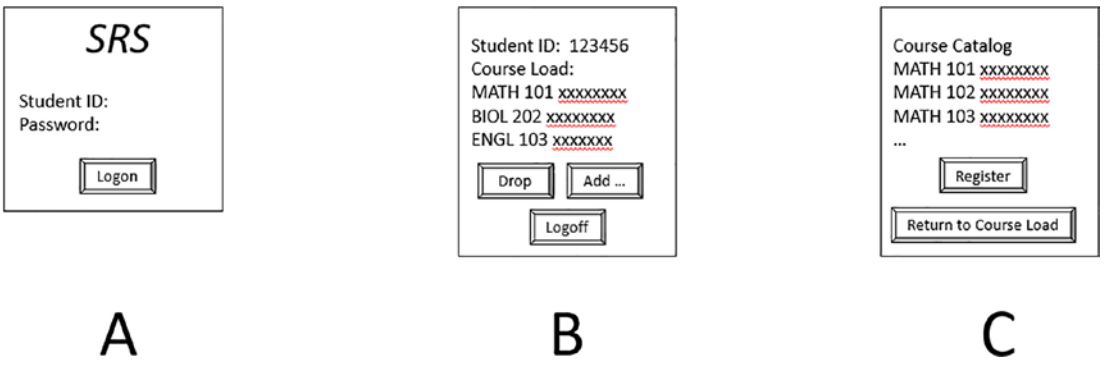


Figure 15-2. *Minimal set of web pages required to build a simple SRS user interface*

Each page is comprised of static HTML content along with optional embedded executable code necessary to customize the page content for that user.

For example, the initial logon page (A) would be all static HTML: a simple form with two input fields to collect the student's unique ID number and password plus a submit button as sketched out in the following (note that this is incomplete HTML):

```
<FORM>
<INPUT TYPE=TEXT NAME=STUDENTID></INPUT>
<BR>
```

```
<INPUT TYPE=TEXT NAME=PASSWORD></INPUT>
<BR>
<INPUT TYPE=SUBMIT NAME=LOGON>
</FORM>
```

The format of the logon page will never change and hence is wholly static HTML.

By contrast, the page that displays the student's list of registered courses needs to be ***dynamically generated*** to display the correct student's current list of enrolled courses and thus might contain Java logic (underlined in the following) as follows (note that this is incomplete HTML):

```
<FORM>
STUDENT NAME: student.getName()
<BR>
<BR>
CURRENTLY ENROLLED IN:
<BR>
<BR>
Collection<Section> enrolledIn = student.getEnrolledSections();
for (Section section : enrolledIn) {
      out.println(section.getSectionNo() + " " section.getDayOfWeek()
      + " " + section.getTimeOfDay() + " " + section.getRoom());

<INPUT TYPE=SUBMIT NAME=DROP></INPUT>
<INPUT TYPE=SUBMIT NAME=ADD></INPUT>
<INPUT TYPE=SUBMIT NAME=LOGOFF></INPUT>
</FORM>
}
```

The manner in which the student object referenced previously gets populated so that this code can be executed is illustrated in the section to follow.

Note that each web page contains one or more FORMs, which in turn contain one or more SUBMIT buttons; clicking a SUBMIT button on a web page is what sends the completed form to the controller to start processing/reacting to the user's actions.

Example Controller Logic

In some respects, you can think of the controller as a master "if-then-else" construct, determining which web page to display next based on the results of the user's interaction with the previous web page displayed. This flow is illustrated with numbered arrows in Figure 15-3; each arrow involves processing by the controller before the next page is displayed.

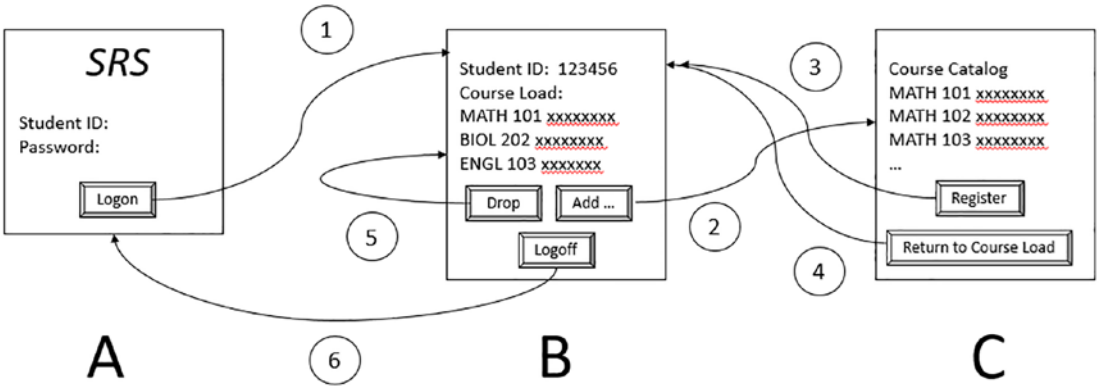


Figure 15-3. *The Controller monitors actions taken by the user (button presses) to determine which actions are to be performed behind the scenes. (Numbers in circles correspond with entries in the numbered event list below.)*

1. If the student types in their student ID and password on web page A and clicks the Logon button, the controller will use the information provided by the student in the STUDENTID and PASSWORD input fields, calling the initialize() method of the SRSPersistenceLayer to query the database for a student with these credentials; if logon is successful, the controller will display web page B after executing the embedded code in that page to display the student's current list of courses.
2. If the student clicks the Add button on web page B, the controller will use the SRSPersistenceLayer to query the database for a list of all available courses; the controller will then display web page C after executing code in web page C to display all of the courses as a list.

3. If the student selects a course on web page C and clicks the Register button, the controller will update the student object's course load, use the SRSPersistenceLayer to store the newly updated student's data in the database, and return to web page B after rerunning the code to display all of the students' registered courses so that the newly added course will appear.
4. The Return to Course Load button on web page C effectively performs in similar fashion to the Register button on web page C.
5. If the student is viewing page B and wants to drop a section, they select the section from the list and click the Drop button. The controller will update the student object's course load, use the data layer to store the newly updated student's data in the database, and refresh page B by rerunning the code to display all of the student's registered courses with the dropped course having been removed.
6. Clicking the Logoff button on screen B will disconnect the session from the database, logging the student off, and will redisplay logon page A.

Let's look at a pseudocode snippet from the controller:

```
// Pseudocode
// Case 1
If (user pressed the Logon button on the logon page A) {
      String studentid = obtain the studentid from the user's response in
      the Student ID field
      String password = obtain the password from the user's response in the
      Password field

      try {
            SRSPersistenceLayer.initializeConnection(studentid, password);
      } catch (InitializationException e) {
            Dispatch an error message to user's browser
      }
```

```
      // Reconstitute student object by calling the appropriate
         method on the
      // SRSPersistenceLayer class.
      Student student = SRSPersistenceLayer.retrieveStudent(studentid);

      Dispatch list of registered courses web page B to user's browser
      after executing the embedded Java code so that the student's full
      list of sections will be displayed


// Case 2
} else if (user pressed the Add button on page B) {
      Collection<Section> scheduleOfClasses =
            SRSPersistenceLayer.getScheduleOfClasses();

      Dispatch list of courses page C to user's browser after executing the
      embedded Java code so that all available sections are displayed

// Case 3
} else if (Register button on register for a course page C is pressed) {
      Use student object that was already created in this session

      String sectionNo = Obtain section number of the course that was
      selected by the user

      // Reconstitute section object.
      Section section = SRSPersistenceLayer.retrieveSection(sectionNo);

      // Register the student for the selection course.
      student.registerForCourse(section);
      // Because the student's state has changed with the addition
         of another
      // registered course, we persist the modified student ...
      SRSPersistenceLayer.storeStudent(student);

      Dispatch list of registered courses web page B to user's
      browser after
      executing the embedded Java code so that the student's full list of
      sections including the one that we just added will be displayed
```

```
// Case 4
} else if (user presses Return button on page C) { ...
// Case 5
} else if (user presses Drop button on page B) { ...
// Case 6
} else if (user presses Logoff button on page B) { ... }
```

There will typically be one if/else-if clause in the controller logic for every possible button click in the user interface (fewer if the same clause can handle multiple similar situations), and in each such clause, the controller will

- Retrieve whatever data the user provided on the currently displayed page.
- Optionally reconstitute objects using the persistence layer as necessary.
- Optionally modify the objects' states based on the user's actions.
- Optionally store the objects back in the database using the persistence layer.
- Determine which page is to be dispatched back to the user's browser, running optional embedded code in that page to refresh the contents as viewed by the user.

This controller loop will continue running until the user exits the session by closing the browser page.

The terminology **model-view-controller** is alternatively used to refer to this controller loop—in essence, the controller serves to keep the view in sync with the model based on user interactions with the application.

The Importance of Model–Data Layer–View Separation

In building both the presentation and persistence layers of our three-tier SRS application, note that the model layer as developed in Chapter 14 remained untouched. Why is this so important? For two reasons:

- By keeping the model pure/uncomplicated by presentation or persistence logic, the model can be reused as is to build other applications for the same client/organization.
- Another very important benefit: By keeping all three layers separate, our application becomes much more adaptable to future changes.
    - If we were to want to switch persistence technologies in the future, all we would need to change would be the internal logic of the PersistenceLayer class–the rest of the application would remain unchanged!
    - If we were to want to replace the presentation layer, there might be impacts on the controller logic, but the model and persistence layers would remain unchanged!

Developers who set out to build a three-tier application without understanding this concept often get persistence logic mixed into the controller or presentation tier logic, which makes it next to impossible to swap out technologies.

By developing our SRS model classes first and using a command-line program to put them through their paces, we've virtually guaranteed **loose coupling**—that is, ease of interchangeability—between tiers if we build the persistence and presentation tiers as suggested in this chapter.

The model layer of an application is typically the layer that changes the least often over an application's lifetime, because the fundamentals of real-world objects (i.e., the business rules for how they operate) are fairly stable, once properly modeled. This is in contrast with both the presentation and data layers of an application, which undergo fairly frequent change due to technology shifts:

- It isn't unusual for the same application to transition from one database back end to another, or even from one type of persistent storage to another, over its lifetime.
- It isn't unusual for the same application to periodically get a "face-lift" in terms of how information is presented to a user.

Unfortunately, many developers dive into Java development by acquiring and using a graphics-oriented Java integrated development environment (IDE). As a result, they wind up using the drag-and-drop features of such a tool to build a GUI front end that

directly connects to a file system or database back end, with ***no model layer whatsoever*** in the middle! Hence, while such applications are, strictly speaking, Java applications, they aren't truly ***object-oriented applications***.

If we instead approach OO application development properly, by doing the following

1. Building the model layer ***first***, as a true OO abstraction of the real-world problem that we are trying to automate
2. Architecting the data and presentation layers separately from the model layer so as to be easily upgradeable when necessary ***without impacting the model***

the resultant applications will be much more resilient to change and will enjoy much longer life cycles, thereby reducing overall software development costs for an organization.

Summary

In this chapter, you've learned

- The typical architecture for a three-tier user-driven application, consisting of a presentation tier, application tier, and persistence tier
- The role of the controller in the application tier
- An approach to building the persistence tier
- An approach to building the presentation tier
- The importance of achieving independence among these three tiers for purposes of flexibility and longevity of the application you are building

Further Reading

Within the Apress library, the following books would be excellent next steps in your Java journey:

- *Beginning Jakarta EE* by Peter Späth
- *Java 17 Recipes* by Josh Juneau and Luciano Manelli

- *Java Challenges* by Michael Inden
- *Java EE to Jakarta EE 10 Recipes* by Josh Juneau and Tarun Telang
- *Beginning Java 17 Fundamentals: Object-Oriented Programming in Java 17* by Kishori Sharan and Adam L. Davis

APPENDIX A

Alternative Case Studies

This appendix proposes some alternative case studies that can be used as the basis of formal coursework or personal study applications. It is also intended to serve as a supplement to many of the end-of-chapter exercises found throughout the book.

Case Study #1: Prescription Tracking System

This case study is relatively straightforward and hence can be tackled by most beginning modelers fairly effortlessly.

Background

Drugs For You pharmacy wishes for us to design and develop an automated Prescription Tracking System (PTS). The requirements are as follows:

- The system is to keep track of the following information for each customer:
  - Customer's name
  - Telephone number
  - Date of birth
  - Insurance provider
  - Insurance policy number
  - A prescription history, detailed next

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6

- Each customer's prescription history will record the following information about each prescription:
  - A unique prescription ID number assigned by the pharmacy
  - The medication being prescribed
  - The prescribing physician's name and telephone number
  - The date of issue
  - Expiration date
  - Number of refills authorized
  - Number of "units" per prescription refill, where a "unit" might be a pill, a teaspoon, a milliliter (ml), etc. (see the discussion of medications next)
  - Whether or not it's okay to provide the customer with a generic substitute, if one exists
- For each medication stocked by the pharmacy, the system will track
  - Its name
  - The "unit" by which the medication is prescribed (pills, teaspoons, ml, etc.)
  - Which medications can serve as "generic" equivalents of which other medication(s)
  - Any common side effects associated with taking the medication
- The system is required to support the following queries (some will be printed as hard-copy reports, whereas others will be viewed online only):
  - A prescription history—that is, a report of all prescriptions ever issued to a given customer—as requested by a given customer
  - A report of all side effects of a given medication, to be enclosed with each prescription dispensed

- A list of all generic substitutes available for a given medication
- Whether a given prescription is refillable—that is, whether any refills remain and whether the prescription has yet to expire

All of the preceding will be accessible via a secure website to individual customers as well as to the in-store pharmacist.

Simplifying Assumptions

A real-life Prescription Tracking System would be quite complicated. I suggest the following simplifications to make the PTS problem a bit more tractable for beginning-level object modelers:

- The system isn't to be concerned with billing matters in any way; that is, we aren't going to worry about computing the price to be paid for a prescription, and we won't be concerned with trying to get a customer's insurance company to reimburse the pharmacy in any way.
- We'll assume that there is only one Drugs For You pharmacy location; that is, it isn't part of a chain of multiple stores.
- The system isn't responsible for inventory control; that is, we'll assume that "infinite" quantities of all medications are in stock or, conversely, that medications are immediately available on demand from a warehouse.
- Assume that the prescription is always refilled with the same medication as was issued for that prescription the first time around; that is, we'll never initially fill the prescription with a generic medication and then refill it with a nongeneric equivalent or vice versa.

Case Study #2: Conference Room Reservation System

This is an advanced case study that involves scheduling complexities and other elaborate requirements, representative of a real-world modeling challenge. It's best suited to an instructor-led group modeling exercise rather than as an individual exercise for a beginning-level modeler.

Background

We've been asked to develop an automated Conference Room Reservation System (CRRS) for our organization:

- A total of a dozen conference rooms are scattered across the four different buildings that comprise our facility. These rooms differ in terms of their seating capacities as well as what audio-visual (A/V) equipment is permanently installed in each room.
- Each of these rooms is overseen by a different administrative staff member, known as a conference room coordinator.
- Reservations are presently being recorded manually by the various conference room coordinators. The name of the person reserving the room, as well as their telephone number, is jotted by hand in an appointment book; the start and stop times of the meeting are also noted.
- A separate, central organization called the A/V equipment group provides "loaner" A/V equipment to supplement any equipment that may be permanently installed in a given conference room. Equipment that is available for temporary use through this group includes conventional overhead projectors, televisions, VCRs, LCD projectors for use with PCs, electronic whiteboards, laptop computers, tape recorders, and slide projectors. Personnel from this group deliver equipment directly to the locale where it's needed and pick it up after the meeting is concluded.

The following problems have been noted regarding the present manual system:

- Currently, no supplemental information regarding the number of attendees or planned A/V equipment usage is being noted by the conference room coordinators for a given meeting.
    - If someone planning a meeting involving only 4 people schedules a room with the capacity for 20, the excess capacity in that room will be wasted. Meanwhile, someone truly needing a room for 20 people will be left short.
    - Meeting planners must also be responsible for separately coordinating with the A/V equipment group; if they forget to do so, panic often ensues as folks scramble to arrange necessary equipment at the last minute.
- Whenever a given room's coordinator is away from their desk, information about that room's availability is inaccessible, unless the inquirer wishes to walk to the coordinator's office and inspect the appointment book directly. However, due to the size of the office complex, this isn't practical, so inquirers typically leave a voicemail message or send an email to the coordinator, who gets back to them at a later point in time.
- People are lax about canceling reservations when a room is no longer needed, so rooms often sit vacant that could otherwise be put to good use. Similarly, they often forget to cancel A/V equipment reservations.
- Pertinent information about the rooms (e.g., their seating capacity, whether or not they have a whiteboard, whether or not they have built-in A/V facilities, whether or not they are "wired" into the company's LAN) isn't presently published anywhere. Someone unfamiliar with the amenities of the various rooms often winds up having to call all 12 of the conference room coordinators in search of an appropriate meeting location.

Goals for the System

We've been asked by management to design a system for providing online, automated conference room and equipment scheduling to remedy the problems of the current manual approach. The goals of this project are to provide the ability for any employee to directly connect to the system to perform the following tasks:

- If the user is interested in scheduling a room for a meeting, they will be required to complete an online questionnaire regarding the parameters of the meeting, to include
  - The scheduler's name, title, department, and telephone number
  - The number of attendees anticipated
  - A date range, indicating the earliest and latest acceptable dates for the meeting
  - The length of time that the room will be required, in half-hour increments
  - An earliest acceptable start time and latest acceptable stop time
  - A list of all A/V equipment required
- As soon as this questionnaire is completed, the system will present the user with a list of all available suitable room alternatives. The user will be able to select from these options to reserve a room or change their criteria and repeat the search.

Note that the system need not "remember" the conference room criteria after the user logs off.

- When confirming a reservation, the user must designate a subject or purpose for the meeting, such as "Demo of CRRS Prototype."
- After a room has been selected, the system will then determine what "loaner" A/V equipment will be needed to supplement the equipment that is permanently installed in that room and will automatically arrange for its delivery.

For purposes of this case study, we won't worry about running out of equipment—we'll assume an infinite supply of everything—although in real life this would also have to be a consideration.

- If no rooms meeting the user's requirements are available, the user will be presented with a list of suitable rooms with the number of people waitlisted for each. The user will be able to optionally place their name at the end of the waiting list for one of these rooms.
    - When such a request is posted, the system is to send a courtesy email to the person holding that room's reservation, asking that person to rethink their need for the room.
    - Should the room become available, it will automatically be temporarily reserved for the first person on the waiting list. An email message is to be sent automatically to the requestor, giving that person 72 hours to confirm their selection before the room either (a) is reassigned (again, temporarily) to the next person on the waiting list or (b) becomes generally available if the waiting list has been exhausted.
- A user should be permitted to query the system as to who has a particular room reserved at a given date and time or to perform a search for a given meeting that the user is to attend based on (a) scheduler or (b) subject.
- The user must be able to cancel a room reservation at any time, whether confirmed or waitlisted.
- The A/V equipment group wishes to periodically run a report, sorted by equipment type, indicating how many times a given piece of equipment was used over a 12-month period.

Case Study #3: Blue Skies Airline Reservation System

This is the most complex case study of all. Please see the introductory comments for case study #2.

Background

Blue Skies Airline, a new airline, offers services between any two of the following US cities: Denver; Washington, D.C.; Los Angeles; New York City; Atlanta; and Cleveland.

When a customer calls Blue Skies to make a flight reservation, the reservation agent first asks them for

- The desired travel dates
- The departure and destination cities
- The seat grade desired (first class, business class, or economy)

The reservation agent then informs the customer of all available flights that meet their criteria. For each flight, the flight number, departure date and time, arrival date and time, and round-trip price are communicated to the customer. If the customer finds any of the available flights acceptable, they may either pay for the ticket via credit card or request that the seat be held for 24 hours. (A specific seat assignment—row and seat number—isn't issued until the seat is paid for.)

A limited number of seats on each flight are earmarked as frequent flyer seats. A customer who is a frequent flyer member may reserve and "pay for" one of these seats by giving the agent their frequent flyer membership number. The agent then verifies that the appropriate balance is available in the customer's account before the seat can be confirmed, at which point those miles are deducted from the account.

The customer has two ticketing options: they may request that a conventional "paper" ticket be issued and mailed to their home address, or an electronic ticket (e-ticket) may instead be assigned, in which case the customer is simply informed of the e-ticket serial number by telephone. (With an e-ticket, the customer simply reports to the airport at the time of their departure and presents suitable ID to a ticket agent at the gate. No paperwork is exchanged.) In both cases, the reservation agent records the serial number of the (conventional or electronic) ticket issued to this customer.

The number of seats available for a given flight in each of the seat grade categories is dependent on the type of aircraft assigned to a given flight.

Other Simplifying Assumptions

As with the PTS case study, there are several simplifying assumptions that can be made as compared with a real-life Airline Reservation System to make this case study more tractable:

- Assume that all flights are round-trip between two cities (no itineraries of three or more legs are permitted).
- Disregard the complication that airlines sometimes have to switch aircraft at the last minute due to mechanical difficulties, thus disrupting the seating assignments.

Index

A

B

J. Barker, *Beginning Java Objects*, https://doi.org/10.1007/978-1-4842-9060-6

C

D

E

F

G

H

I

J, K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y, Z