MASTERING COMPUTER SCIENCE

Mastering Scala

A Beginner's Guide

EDITED BY

Sufyan bin Uzayr

CRC Press
Taylor & Francis Group

Mastering Scala

Scala is a multi-paradigm, general-purpose scripting language. It is a completely object-oriented programming language that supports a functional programming technique. This book is a detailed guide for beginners to understand Scala.

Concise and easy to understand, *Mastering Scala: A Beginner's Guide* covers a comprehensive understanding of Scala and its components, libraries, and advance concepts to help readers quickly advance with the necessary information.

This book provides functional approaches for solving queries using Scala. The fundamental principles of Scala explained here are helpful to beginner and intermediate users interested in learning this highly technical and diverse language.

Key Features:

- Follows a hands-on approach and offers practical lessons and tutorials related to Scala
- Includes detailed tutorials meant for beginners to Scala
- Discusses Scala in-depth to help build robust knowledge

About the Series

The *Mastering Computer Science* covers a wide range of topics, spanning programming languages as well as modern-day technologies and frameworks. The series has a special focus on beginner-level content, and is presented in an easy-to-understand manner, comprising:

- Crystal-clear text, spanning various topics sorted by relevance.
- Special focus on practical exercises, with numerous code samples and programs.
- A guided approach to programming, with step-by-step tutorials for the absolute beginners.
- Keen emphasis on real-world utility of skills, thereby cutting the redundant and seldom-used concepts and focusing instead of industry-prevalent coding paradigm,
- A wide range of references and resources, to help both beginner and intermediate-level developers gain the most out of the books.

Mastering Computer Science series of books start from the core concepts, and then quickly move on to industry-standard coding practices, to help learners gain efficient and crucial skills in as little time as possible. The books assume no prior knowledge of coding, so even the absolute newbie coders can benefit from this series.

Mastering Computer Science series is edited by Sufyan bin Uzayr, a writer and educator having over a decade of experience in the computing field.

For more information about this series, please visit: https://www.routledge.com/Mastering-Computer-Science/book-series/MCS

Mastering Scala
A Beginner's Guide

Edited by
Sufyan bin Uzayr

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

First Edition published 2024
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton, FL 33431

and by CRC Press
2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

Library of Congress Cataloging-in-Publication Data

Names: Bin Uzayr, Sufyan, editor.
Title: Mastering Scala : a beginner's guide / edited by Sufyan Bin Uzayr.
Description: First edition. | Boca Raton : CRC Press, 2024. | Series: Mastering computer science | Includes bibliographical references and index.
Identifiers: LCCN 2023017681 (print) | LCCN 2023017682 (ebook) | ISBN 9781032415291 (hbk) | ISBN 9781032415277 (pbk) | ISBN 9781003358527 (ebk)
Subjects: LCSH: Scala (Computer program language) | Object-oriented programming (Computer science) | Computer programming.
Classification: LCC QA76.73.S28 M37 2024 (print) | LCC QA76.73.S28 (ebook) | DDC 005.13/3--dc23/eng/20230601
LC record available at https://lccn.loc.gov/2023017681
LC ebook record available at https://lccn.loc.gov/2023017682

ISBN: 9781032415291 (hbk)
ISBN: 9781032415277 (pbk)
ISBN: 9781003358527 (ebk)

DOI: 10.1201/9781003358527

Typeset in Minion
by KnowledgeWorks Global Ltd.

For Mom

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

About the Editor

Sufyan bin Uzayr is a writer, coder, and entrepreneur having over a decade of experience in the industry. He has authored several books in the past, pertaining to a diverse range of topics, ranging from History to Computers/IT.

Sufyan is the Director of Parakozm, a multinational IT company specializing in EdTech solutions. He also runs Zeba Academy, an online learning and teaching vertical with a focus on STEM fields.

Sufyan specializes in a wide variety of technologies, such as JavaScript, Dart, WordPress, Drupal, Linux, and Python. He holds multiple degrees, including ones in Management, IT, Literature, and Political Science.

Sufyan is a digital nomad, dividing his time between four countries. He has lived and taught in universities and educational institutions around the globe. Sufyan takes a keen interest in technology, politics, literature, history, and sports, and in his spare time, he enjoys teaching coding and English to young students.

Learn more at sufyanism.com

Acknowledgments

There are many people who deserve to be on this page, for this book would not have come into existence without their support. That said, some names deserve a special mention, and I am genuinely grateful to:

- My parents, for everything they have done for me.
- The Parakozm team, especially Divya Sachdeva, Jaskiran Kaur, and Simran Rao, for offering great amounts of help and assistance during the book-writing process.
- The CRC team, especially Sean Connelly and Danielle Zarfati, for ensuring that the book's content, layout, formatting, and everything else remain perfect throughout.
- Reviewers of this book, for going through the manuscript and providing their insight and feedback.
- Typesetters, cover designers, printers, and everyone else, for their part in the development of this book.
- All the folks associated with Zeba Academy, either directly or indirectly, for their help and support.
- The programming community, in general, and the web development community, in particular, for all their hard work and efforts.

Sufyan bin Uzayr

Zeba Academy – Mastering Computer Science

The "Mastering Computer Science" series of books are authored by the Zeba Academy team members, led by Sufyan bin Uzayr, consisting of:

- Divya Sachdeva
- Jaskiran Kaur
- Simran Rao
- Aruqqa Khateib
- Suleymen Fez
- Ibbi Yasmin
- Alexander Izbassar

Zeba Academy is an EdTech venture that develops courses and content for learners primarily in STEM fields and offers educational consulting and mentorship to learners and educators worldwide.

Additionally, Zeba Academy is actively engaged in running IT schools in the CIS countries and is currently working in partnership with numerous universities and institutions.

For more info, please visit https://zeba.academy

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 1

Scala Overview

IN THIS CHAPTER

- Introduction and History
- Setting Up the Environment in Scala
- Hello World in Scala
- Uniform Access Principle

Scala is a multi-paradigm, general-purpose scripting language. It is a completely object-oriented programming (OOP) language that supports the functional programming (FP) technique. Programs written in Scala may be translated into bytecodes and run on the Java Virtual Machine (JVM). Scalable language is referred to as Scala. Runtimes for JavaScript are also accessible. Java and other programming languages like Lisp, Haskell, Pizza, and others have had a significant effect on Scala.

SCALA'S EVOLUTION

Scala was created by Martin Odersky, a German computer scientist and professor of programming techniques at École Polytechnique Fédérale de Lausanne (EPFL) in Switzerland. Martin Odersky also co-created Javac, Generic Java, and the EPFL Funnel programming language. He began designing the Scala in 2001. On the Java platform, Scala's original version was made available to the public in 2004. In June 2004, Scala was upgraded for the .Net Framework. The second version, v2.0, came out shortly after in 2006. The ScriptBowl competition at the 2012 JavaOne conference was won

DOI: 10.1201/9781003358527-1

by Scala. As of June 2012, Scala no longer supports the .Net Framework. The most recent version of Scala is 2.12.6, which was released on April 27, 2018.

VERSIONS OF SCALA

Version	Released Date
2.0	March 12, 2006
2.1.8	August 23, 2006
2.3.0	November 23, 2006
2.4.0	March 9, 2007
2.5.0	May 2, 2007
2.6.0	July 27, 2007
2.7.0	February 7, 2008
2.8.0	July 14, 2010
2.9.0	May 12, 2011
2.10	January 4, 2013
2.10.2	June 6, 2013
2.10.3	October 1, 2013
2.10.4	March 18, 2014
2.10.5	March 5, 2015
2.11.0	April 21, 2014
2.11.1	May 20, 2014
2.11.2	July 22, 2014
2.11.4	October 31, 2014
2.11.5	January 8, 2015
2.11.6	March 5, 2015
2.11.7	June 23, 2015
2.11.8	March 8, 2016
2.12.1	December 5, 2016

SCALA'S POPULARITY

- Twitter revealed that it has converted a significant amount of its backend from Ruby to Scala and planned to convert the remainder.
- Specific teams at Apple Inc. employ Scala, Java, and the Play framework.
- The New York Times' internal content management system, Blackbeard, was constructed using Scala, Akka, and Play Framework.

- Google teams employ Scala mostly as a result of acquisitions like Firebase and Nest.
- Scala is used by Walmart Canada's backend platform.

WHY USE SCALA?

The popularity of Scala among programmers is influenced by a number of variables. The advantages of using Scala are as follows:

- Simple to start with: Scala is a high-level programming language that is comparable to well-known ones like Java, C, and C++. Scala learning becomes incredibly easy for everyone as a result. For Java programmerslearning Scala is easier.
- Contains the best features: Scala enhances usability, scalability, and productivity by incorporating features from other languages such as C, C++, Java, etc.
- Java and Scala are closely integrated: The source code for Scala was written such that its compiler could understand Java classes. Additionally, its compiler is capable of using Java libraries, frameworks, and other tools. Scala programs can run on the JVM after compilation.
- Web-based and desktop application development: It offers support for web apps by compiling JavaScript. Similarly, it is possible to compile desktop apps to JVM bytecode.
- Utilized by major corporations: The majority of well-known corporations, including Apple, Twitter, Walmart, Google, etc., migrate the majority of their code from other languages to Scala because it is highly scalable and suitable for usage in backend operations.

People frequently mistake Scala for a Java extension. But this is not the case. It is just fully compatible with Java. After successful compilation, Scala programs are translated into .class files containing Java Byte Code and may subsequently be launched on JVM.

SCALA'S TOP 10 USES

The following are the top ten uses:

1. **A multi-paradigm language:** Scala is a worthwhile language since it supports object-oriented and functional programming. It develops imperative, logical, functional, and OOP abilities. We may efficiently study both functional and OOP simultaneously. Scala permits the definition of types with both data and behavior characteristics. Scala functions are a first class that permits passing values and enables anonymous functions. This is one of the primary reasons Scala has become so successful in the marketplace. It may use in conjunction with Java.

2. **Scala runs on Java Virtual Machine:** It is up to the Scala user to decide whether or not to utilize Java. This Java interoperability capability is one of Scala's most excellent alternatives. This allows developers to use all Java libraries straight from Scala code. It is also advantageous for Java engineers since they may use their talents in Scala relatively easily. It is also possible to invoke Scala code from Java, allowing users to write any portion of a program in Scala and the remainder in Java. This functionality allows users to develop code in Java and Scala and operate with both languages.

3. **Patterns are incorporated into the language:** Scala was created in a Swiss university to create new advances in studying computer languages like Java. This language already incorporates several best practices and design principles. In Java, variables are immutable and readily overloaded. In addition, it allows using new programming languages, such as Python, Ruby, etc., to perform FP.

4. **A language that communicates:** Scala is, by nature, a more expressive language than Java. Coding in Scala is more straightforward and exciting for programmers who have previously learned Java.

5. **Market demand is high:** A developer must constantly be in demand. The primary purpose or use of Scala is to improve economic development and employment. Learning Scala will boost our marketability and raise our demand. Scala is used by several firms, including Twitter, LinkedIn, Foursquare, etc. Once we have mastered Scala, we may quickly get the desired promotion. Due to its scalability, all

investment banks and financial institutions will use Scala in the near future. Several businesses discuss efficient Scala use techniques. It will soon be the first Java substitute.

6. **Language that is typed statically:** A statically typed language prevents errors in code and aids programmers in writing and debugging correct code. In dynamic programming languages, faults are only noticeable when a program is executed. Scala offers the advantages of both static and dynamic programming languages. The language seems dynamic but is mostly typed statically. Scala enables type inference for variables and functions, which is superior to Java and C# type inference. It also offers a compiler that makes extensive use of type references.

7. **Growing frameworks:** Scala applications supply several libraries; thus, they may utilize to develop numerous frameworks. Numerous organizations are attempting to make Scala a mainstream language. There are already several frameworks in existence, such as Lift and Play. Akka is a concurrent Scala-based framework that is developed as a toolkit and runtime for developing highly concurrent, distributed, and fault-tolerant systems. It also offers an improved platform for event-driven applications running on JVM.

8. **Creating a community:** Scala is a rapidly expanding language, and many programmers will soon join the Scala bandwagon. Even developers who are familiar with Java are beginning to learn Scala. Numerous new libraries and frameworks are being developed with the use of Scala. Other IDEs under development accept Scala and provide far more excellent support than Eclipse and IntelliJ. Scala is also advantageous due to its inherently dynamic nature. Additionally, it supports object-oriented and functional programming.

9. **Precise syntax:** Scala's exact grammar provides an additional benefit. Java has a lengthy syntax. Scala is simultaneously more readable and concise. Scala's compiler, Scalac, may build and operate with improved code, such as String(), equals(), etc.

10. **Relatively simple to learn:** Any functional language is tough to learn for a Java programmer. Object-oriented functionality makes Scala simple to use. Scala features clear syntax, decent libraries, excellent online documentation, and widespread industrial use.

SCALA'S ADVANTAGES OVER JAVA

- **Code simplicity and size:** The most common pro-java argument is that Java is incredibly straightforward and intuitive to learn. However, the verbose nature of Java increases code size and frequently makes it more difficult to understand. The Scala compiler, on the other side, is smarter since it does not need an explicit declaration of things that the compiler may deduce. Consider the "Hello Everyone" example.

```
// Java
public class HelloEveryone {
    public static void main(String[] args) {
        System.out.println("Hello Everyone");
    }
}
//scala
object HelloEveryone {
    def main(args: Array[String]): Unit = {
        println("Hello Everyone")
    }
}
```

Even in this simple instance, we can see how Scala eliminates the need for redundant code. The same is true in complex circumstances.

```
public class Bus {
    private String type;

    public String getType() {
        return type;
    }

    public void setType(String name) {
        this.type = type;
    }
}
```

Scala makes this much easier.

```
class Bus {
    var type: String = _
}
```

Scala does not have the limits of OO conventions for implementing our code, resulting in concise code. Fewer lines of code eventually result in increased testing and development speed.

- **Performance:** Google conducted research many years ago comparing C++, Java, Scala, and Go. The research revealed that Scala is quicker than Java and Go when average developers write code without excessive optimization consideration. The research used the idiomatic default data structures in each language. This is because they believe that developers are under time constraints and produce idiomatic code for the language utilizing approaches that are simple and quick for developers.

  According to several websites, Scala is quicker than Java. Several programmers claim that Scala is 20% quicker than Java. Scala and Java both run on JVM. Consequently, their code must compile into bytecode before execution on JVM. However, the Scala compiler enables tail call recursion as an efficient approach. The optimization allows Scala code to be compiled more quickly than Java code.

- **Typed statically:** A statically typed language, such as Java, identifies problems at build time and requires us to declare the variable type before using it, whereas a dynamic language, such as Python, only detects errors at runtime.

  Scala is blessed with the finest of both worlds. It is heavily statistically coded yet has a dynamic vibe about it. The compiler guarantees that a specific type Int value is utilized correctly throughout the program and that nothing other than an Int may retain in that value's memory address during runtime. The Scala compiler makes extensive use of type inference.

```
var c = 20
```

  Scala's type inference for variables and functions is far superior to Java's restricted type inference.

- **Advanced structures:** Scala's syntax is quite close to that of Java. Scala, however, has a lot more sophisticated structures than Java. Scala, for example, offers case classes that represent immutable value objects and sophisticated automated type inference. Because of Scala's highly organized nature, DSL is particularly popular. As a result, programmers may alter Scala's appearance by writing their little sublanguage as necessary.

- **Framework and community development:** Scala's ecosystem is constantly expanding. Many big firms have begun to use Scala. Because of its widespread usage, many excellent frameworks like Lift, Finch, and Play exist.

Akka is a Scala-based concurrent framework well-known as a toolkit and runtime for developing highly concurrent, distributed, and fault-tolerant event-driven systems on the JVM. Because of its excellent concurrency mechanism, developers choose Akka over competitors.

START WITH SCALA PROGRAMMING

- **Locating a compiler:** Several online IDEs, such as Scala Fiddle IDE and others, may run Scala applications without installing anything.
- **Programming in Scala:** Because Scala is syntactically comparable to other commonly used languages, it is easy to code and learn in Scala. Scala programs may develop in any of the frequently used text editors such as Notepad++, gedit, and so on. After developing the program, save it with the extension .sc or .scala.
- **For Windows and Linux:** Before installing Scala on Windows or Linux, we must have Java Development Kit (JDK) 1.8 or higher installed on our machine. Because Scala requires Java 1.8 or above to execute.

```
// Scala program to print Hello, Everyone by using
the object-oriented approach

// creation of object
object Everyone {

    // the main method
    def main(args: Array[String])
    {

        // prints Hello, Everyone
        println("Hello, Everyone")
    }
    }
```

- **Comments:** Comments are used to describe the code in the same way as in Java, C, or C++. Compilers ignore and do not execute comment items. Comments might be single or many lines long.

- **Single-line comments:**

  Syntax:

  ```
  // Single-line-comment
  ```

- **Multi-line comments:**

  Syntax:

  ```
  /* Multi-line
  Comments */
  ```

- **Object Everyone:** Object is the term that is used to construct objects. The item is called "Everyone" in this case.
- **def main(args: Array[String]):** In Scala, the term def is used to declare the function, and "main" is the name of the Main Method. The command line parameters are represented by args: Array[String].
- **println("Hello, Everyone"):** println is a Scala method that displays a string on the console.

  It should note that a functional technique may also be employed in Scala projects. Some online IDEs do not support it.

SCALA FEATURES

Several characteristics distinguish it from other languages.

- **Object-oriented:** Since every value in Scala is an object, this programming language is entirely object-oriented. In Scala, classes and characteristics illustrate the behavior and type of objects.
- **Functional:** It is a FP language since each function is a value, and each value is an object. It supports high-order functions, nested functions, anonymous functions, and so on.
- **Statically typed:** Scala is statically typed, meaning that the process of checking and enforcing type restrictions occurs at build time. Scala, unlike other statically typed programming languages such as C++, C, etc., does not need the user to provide redundant type information. Most of the time, the user is not required to provide a type.
- **Extensible:** Scala is extensible in that new language constructs may be introduced as libraries. Scala is developed for compatibility with the JRE (Java Runtime Environment).

- **Concurrent and synchronize processing:** Scala enables users to construct immutable code, simplifying parallelism (synchronize) and concurrency.
- **Run on JVM and able to execute Java code:** Java and Scala share the same runtime environment, the JVM, so that the user may transition quickly from Java to Scala. The Scala compiler generates the application into a .class file that contains executable Bytecode for JVM. Scala can use all classes of the Java Standard Development Kit. The Java classes may be customized with the aid of Scala.

ADVANTAGES

- Scala's sophisticated characteristics allowed for superior code and performance efficiency.
- Scala's developments include tuples, macros, and functions.
- It includes object-oriented and functional programming, which makes it a potent programming language.
- It is very scalable and hence offers superior support for backend activities.
- It mitigates the greater risk associated with Java's thread safety.
- The functional approach typically results in fewer lines of code and defects, leading to increased productivity and quality.
- Scala computes expressions only when the program needs them due to lazy evaluation.
- Scala lacks both static methods and variables. It employs the unique object (class with one object in the source file).
- It also includes the idea of traits. The collection of abstract and non-abstract methods that may compile into Java interfaces is known as a trait.

DISADVANTAGES

- Occasionally, two techniques make Scala challenging to comprehend.
- Comparatively, there are fewer Scala developers available than Java developers.

- As it runs on JVM, it has no true-tail recursive optimization.
- Scala is an object-oriented computer program where each function is a value and each value is an object.

APPLICATIONS

- It is mainly used for data analysis using spark.
- Utilized in the development of web apps and API.
- It facilitates the development of frameworks and libraries.
- Preferred for usage in backend processes to increase developer efficiency.
- Scala may use for parallel batch processing.

AN INTRIGUING FACT ABOUT SCALA

Scala (pronounced "skah-lah") is a computer language created by Martin Odersky. Scala's development began in 2001 at EPFL in Lausanne, Switzerland. Scala was first made public in 2004 on the Java platform. Scala is intended to be brief and solves Java's shortcomings. Scala source code is converted to Java byte code, then executed on a JVM 2.12.8, which is the most recent version.

- **Name:** Scala is an abbreviation for Scalable Language.
- **A Hybrid Language:** A Mixed Language: Scala is an OOP and FP hybrid. OOP is a programming paradigm built on the notion of "objects," which are data structures that hold data in the form of fields and code in the form of procedures or methods. On the other side, FP is a programming paradigm in which computer programs are built with a structure and elements such that the evaluation of mathematical functions is considered computation and prevents changeable data and changing states. Scala is distinguished from other programming languages by these two paradigms.
- **Auto-inference:** Scala uses auto-inference to infer type information. The user only provides type information when it is required.
- **Mutable and immutable variables:** Scala allows us to declare any variable as mutable or immutable. The term var indicates that a

variable is changeable, whereas the keyword val indicates that a variable is immutable.

- **No semicolon:** In most current programming languages (C, C++, Java, etc.), the semicolon serves as a separator and must be typed after every statement. Scala, on the other hand, does not require a semicolon after each statement. A newline character can use to separate Scala statements.
- **Import statements:** Not all import statements must be written at the program's start. Scala classes can import at any time.
- **Features of Scala:** Scala incorporates characteristics of functional programming languages such as Scheme, Standard ML, and Haskell, in addition to Java's OOP capabilities, such as currying, type inference, immutability, lazy evaluation, and pattern matching.
- **Functions and procedures:** Functions and procedures are different in Scala and should not be used interchangeably. A function prototype can return any type and contains the = sign. On the other hand, the procedure does not include a = sign and always returns Unit(). In general, print statements are discouraged in the function definition.

  Example:

```
def func1():Int = {
    //this is function returns Int
    }

    def proc1() {
    //this is procedure returns void(Unit())
    }
```

- **Higher-order functions:** We can send a function as a parameter to another function in Scala. These are known as higher-order functions.

  Example:

```
val l = List(1, 2, 3)
l.foreach(println) // println passed as argument
to foreach function
```

  In addition, the return value of a function might be another function.

Example:

```
def square(c:Float) = { pow(c, 2) }
```

- **Supports nested functions:** We may define and utilize a function within another function as needed. Any point inside the outer function's scope can invoke the nested function.
- **Industry of big data:** Apache Spark is a commonly used large data processing solution that is an open-source cluster computing platform. Scala is used to write Spark programs because of its JVM scalability. Scala is the most often used programming language among big data specialists working on Spark projects. Instances of Spark using Scala include Alibaba, Netflix, and Pinterest.

SETTING UP THE SCALA ENVIRONMENT

Scala is a reasonably compatible language; thus, it can simply install on Windows and Unix operating systems. In this portion, we will study how to proceed with the Scala environment installation and configuration. The most fundamental prerequisite is that our computer has installed Java 1.8 or a later version. We'll look at the stages for Windows and Unix individually.

- **Step 1: Validate Java packages:** The first need is a Java Software Development Kit (SDK) installed on the computer. We must validate these SDK packages and install them if not already installed. Open the command prompt and enter the following commands:

  For Windows

  ```
  C:\Users\Your_PC_username>java -version
  ```

  When we run this command, the result will display the Java version, which is as follows:

  ```
  java version "1.8.0_111"
  Java(TM) SE Runtime Environment (build
  1.8.0_111-b14)
  Java HotSpot(TM) 64-Bit Server VM (build 25.111-
  b14, mixed mode)
  ```

  For Linux

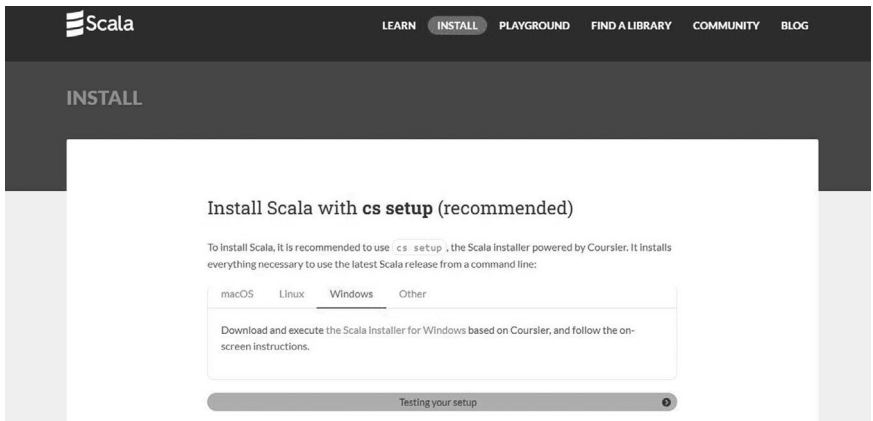
  ```
  $ java -version
  ```

When we run this command, the results will show the Java version and will look like this:

```
java version "1.8.0_20"
Java(TM) SE Runtime Environment (build
1.8.0_20-b26)
Java HotSpot(TM) 64-Bit Server VM (build 25.20-
b23, mixed mode)
```

If we obtain the result above, we have installed the most recent Java SDK and are ready to proceed to STEP 2. If we don't already have the SDK installed, go to https://www.oracle.com/technetwork/java/javase/downloads/jdk12-downloads-5295953.html and download the latest version based on our machine requirements.

- **Step 2: Install Scala:** Now that we've installed Java let's install the Scala packages. Downloading these packages from the official site is the best option: https://www.scala-lang.org/download/ the packages on the URL above have about 100MB of storage. Once the packages have been downloaded, open the downloaded.msi file and follow the steps outlined below:



Installation of Scala.

1: Press the NEXT button.

2: The screen will show when we click the NEXT button. Check the "I Agree" box, and then click NEXT.

3: Continue with the installation.
Select the INSTALL option.

4: The installation procedure begins.
Allow time for the packages to install.

5: Installation is Complete.
Select the FINISH option.
Now that the packages are complete, we can begin utilizing Scala.

- **Step 3: Run and test the Scala commands:** Now launch the command prompt and enter the following commands.

```
C:\Users\Your_PC_username>scala
```

Now that we have Scale installed, we can write some commands to test several Scala statements:

```
scala>println("Hi, Everyone who is Learning
Scala")
scala>14+6
scala>8-5
```

The Scala environment is now operational. We may now work with Scala by entering commands into the command prompt window.

SCALA INSTALLATION IN LINUX

Before we begin, we must first install Scala on our system. We need a first-hand understanding of what the Scala language is and what it performs. Scala is a multi-paradigm, general-purpose language of programming. It is an FP that allows OOP. Scala has no concept of basic data since everything is an object. It is intended to convey general programming patterns in a polished, concise, and type-safe manner. Scala programs may be converted to bytecodes and executed on the JVM. Scala is an abbreviation for Scalable language. It also includes Javascript runtimes. Scala is heavily influenced by Java and other computer languages like Lisp, Haskell, Pizza, and others.

Scala is a relatively compatible programming language that can readily load into the Linux operating system. The most fundamental prerequisite is having Java 1.8 or above installed on our PC.

VERIFYING JAVA PACKAGES

The first step is to install an SDK on the computer. We must validate these SDK packages and install them if not already installed. Simply enter the following command into the Terminal:

```
java --version
```

When we run this command, the result will display the Java version.

If we do not have the SDK installed, use the following command to obtain the newest version based on the machine's requirements:

```
sudo apt-get install default-jdk
```

SCALA DOWNLOAD AND INSTALLATION

Scala Download

We must first download it before proceeding with the installation. Scala for Linux is available in all versions at https://www.scala-lang.org/download/.

Scala may download here, and installation instructions can find here. However, the current version of Scala may simply install on Ubuntu by using the following command:

```
sudo apt-get install scala
```

To begin with, the installation:

- Getting started
- Begin the installation procedure
- Completed Installation

After installing Scala, any IDE or text editor may use to develop Scala code and run it on the IDE or the terminal using the command:

```
scalac filename.Scala
scala classname
```

Here's an example program to get us started with Scala programming:

Consider a basic Hello World program.

```
// program to print Hello World
object Peeks
```

```
{
    // the main Method
    def main(args: Array[String])
    {
        // prints the Hello World
        println("Hello World")
    }
}
```

WHAT CAUSES SCALA TO BE SCALABLE?

A language's scalability is influenced by various elements ranging from syntactic nuances to component abstraction constructs. Scala's fundamental feature that makes it scalable is that it combines object-oriented and functional programming. It supports high-order functions, tail-call optimization, immutable values, pattern matching, polymorphism, abstraction, inheritance, and other programming features. Scala also has its interpreter, which may use to execute instructions without first compiling them. Another significant component is the parallel collections library, which is meant to assist developers in dealing with parallel programming patterns.

Other features include the following:

- Scala is concise. It improves support for backend operations. Scala programs are up to ten times shorter than Java applications. It avoids code that repeatedly appears to achieve a result that burdens the Java application.

  Example: A constructor class in Java looks like this:

```
class Peek
{
    // class data members
    String name;
    int id;

    // constructor would initialize data members
with
    // values of passed arguments while the object
of that class is created.
    Geek(String name, int id)
```

```
    {
        this.name = name;
        this.id = id;
    }
}
```

- Scala assists us in managing complexity by allowing us to increase the amount of abstraction in the interfaces we develop. Stings are low-level entities in Java, which are processed character by character in a loop. The Scala code treats the exact string as higher-level character sequences that may be searched. Scala allows for the creation of frameworks and libraries.

  Example: In Java, identify the first capital letter.

```
// Function to find the string with the first
character of each word.
static char first(String str)
{
    for (int m = 0; m < str.length(); m++)
        if (Character.isUpperCase(str.charAt(m)))
            return str.charAt(m);
    return 0;
}
```

- Strings are low-level entities in Java programming, whereas they are high-level things in Scala _.isUpperCase is a function literal in Scala.
- It enables static typing, in which calculations are expressed as statements that affect the program's state at build time. It is a method that can enhance runtime efficiency. A static type system classifies variables and expressions based on the values they contain and compute. A hierarchical class type system similar to Java's allows us to parameterize types with generics, conceal details with abstract types, and mix types using intersections.
- Because it is built atop the JVM, it has access to all Java methods and fields, may inherit from Java classes, and implement Java interfaces. No specific terminology, detailed interface descriptions, or glue code is required. It takes Java types and dresses them up to make them appear prettier. The Scala compiler converts the program into a .class file containing the Bytecode the JVM may run. Scala can utilize all of the Java SDK classes. Scala allows the user to modify Java classes.

HELLO WORLD IN SCALA

The Hello World program is the most basic and first program we should try while learning a new computer language. This displays the message Hello World on the output screen. A simple program in Scala includes the following elements:

- Object
- Main Method
- Statements or Expressions

Example:

```
// program to print Hello World
object Peeks
{
    // the main Method
    def main(args: Array[String])
    {
        // prints the Hello World
        println("Hello World")
    }
}
```

Explanation:

- Peeks: object is the term that is used to construct objects. A class's instance is an object. The item is called "Peeks" in this case.
- def main(args: Array[String]): In Scala, the term def is used to declare the function, and "main" is the name of the Main Method. The command line parameters are represented by args: Array[String].
- println("Hello World"): println is a Scala method that displays the Output on the console.

HOW TO EXECUTE A SCALA PROGRAM

To utilize an online Scala compiler, follow these steps: We may use multiple online IDEs to run Scala applications without installing anything.

Using the Command Line: Check that we have the Java 8 JDK (also known as 1.8). In the command line, use the javac -version to ensure that we see javac 1.8.___ If version 1.8 or above is unavailable, install the

Java Development Kit. To begin, launch a text editor such as Notepad or Notepad++. In a text editor, write the code and save it with the (.scala) extension. Launch the command prompt and proceed through the steps on our machine.

```
// program to print Hello World
object Peeks
{
    // the main Method
    def main(args: Array[String])
    {
        // prints the Hello World
        println("Hello World")
    }
}
```

- Step 1: Compile the above file with scalac Hello.Scala will create Peeks after compilation. class file and the name of the class file are the same as the name of the object (Here Object name is Peeks).
- Step 2: Run the command with the object name scala Peeks. It will provide the outcome.

Scala IDE: IDEs such as IntelliJ IDEA and ENSIME make it simple to run Scala programs. Enter the code in the text editor and hit enter to run it.

UNIFORM ACCESS PRINCIPLE IN SCALA

In Scala, a programming concept known as the Uniform Access Principle is implemented; this asserts that the annotations used to get the property of a Class are similar for both methods and variables. Bertrand Meyer advocated for this principle. The concept essentially states that the notation used to access a class feature should be the same whether it is a method or an attribute.

Some points to consider:

- This Principle states that attributes and functions with no parameters can be accessed using the same syntax.
- A function declaration with no arguments can convert to "var" or vice versa.
- This Principle is more closely related to OOP.

Example:

```
// program for Uniform Access Principle

// Creation of an object
object Access
{

    // the main method
    def main(args: Array[String])
    {

        // Creating array
        val m : Array[Int] = Array(17, 28, 49, 20,
55)

        // Creating String
        val n = "PeeksforPeeks"

        // Accessing length of an array
        println(m.length)

        // Accessing length of a String
        println(n.length)
    }
}
```

We now understand that the length of an array is variable and the length of a string is a method in the Class "String," but we accessed both in the same way.

Example:

```
// program for Uniform Access Principle

// Creating object
object Access
{

    // the main method
    def main(args: Array[String])
```

```
    {

        // Creating a list
        val m = List(11, 43, 25, 57, 7, 30)

        // Creating a method
        def portal = {
        "Peeks" +"for" + "Peeks"
        }

        // Accessing size of a method
        println(portal.size)

        // Accessing size of a variable
        println(m.size)
    }
}
```

SCALA VS. JAVA

Java is a general-purpose computer language that is concurrent, class-based, and object-oriented, among other characteristics. Java programs are compiled into bytecode, which may execute on any Java virtual machine, independent of computer architecture.

Scala is a multi-paradigm, general-purpose programming language. It is an OOP language that also supports the FP technique. Scala has no concept of basic data since everything is an object. It is intended to convey general programming patterns in a polished, concise, and type-safe manner.

The following are some significant differences between Scala and Java:

Scala	Java
Scala is a cross between object-oriented and functional programming.	Java is a general-purpose, object-oriented programming language.
Because of the nested code, Scala is less understandable.	Java is easier to read.
The compilation of source code into byte code is a time-consuming operation.	The compilation of source code into byte code is a quick process.
Scala allows for operator overloading.	Operator overloading is not supported in Java.
Scala allows for slow evaluation.	Lazy evaluation is not supported in Java.

(Continued)

Scala	Java
Scala does not support backward compatibility.	Java is backward compatible, which implies that code written in the latest version will execute without issue in prior versions.
Scala treats any method or function as though it were a variable.	Java considers functions to be objects.
Scala code is written concisely.	The code in Java is written in long form.
Scala variables are immutable by default.	Java variables are mutable by default.
Scala is more object-oriented than Java and regards everything as an instance of the class.	Because of the presence of primitives and statics, Java is less object-oriented than Scala.
The static keyword does not exist in Scala.	Java includes the static keyword.
Method calls are used to perform all operations on entities in Scala.	In Java, operators are processed differently from method calls.

PYTHON VS. SCALA

Python is a high-level, interpreted, general-purpose dynamic programming language with a significant emphasis on code readability. Python needs less typing, has new libraries, allows faster prototyping, and has various other new features.

Scala is a high-level programming language. It is an OOP language alone. Scala's source code is written so its compiler can read Java classes.

The following are some critical distinctions between Python and Scala:

Python	Scala
Python is a dynamically typed programming language.	Scala is a statically typed programming language.
Python is a dynamically typed Object Oriented Programming language; therefore, we don't need to declare objects.	Because Scala is a statically typed Object Oriented Programming language, we must declare the type of variables and objects.
Python is simple to learn and apply.	Scala is easier to learn than Python.
At runtime, the interpreter is given more tasks.	Scala generates no extra work, making it ten times quicker than Python.
It determines the data types at runtime.	This is not the case in Scala, so it should be used instead of Python is useful for dealing with large volumes of data.
The Python community is far larger than the Scala community.	Scala is also well-supported by the community. However, it is inferior to Python.
Python enables heavyweight process forking but not true multithreading.	Scala features reactive cores and a variety of asynchronous libraries, making it a preferable alternative for concurrent implementation.

(Continued)

Python	Scala
Python's techniques are significantly more complicated because it is a dynamic programming language.	Because Scala is a statically typed language, testing is significantly easier.
Its popularity stems from its English-like syntax.	Scala has a considerably more significant role in scalable and concurrent systems.
Python simplifies the writing of code for developers.	Scala is simpler to learn than Python, but it is more difficult to write code in Scala.
Python has an interface to numerous OS system functions and libraries. There are several interpreters.	It is a compiled language, with all source code being compiled before execution.
When there is a modification to the existing code, the Python language is very prone to problems.	Scala does not have such an issue.
Python has machine learning, data science, and speech recognition packages (NLP).	Scala, on the other side, does not have such tools.
Python is suitable for small-scale tasks.	Scala is suitable for large-scale projects.
It does not support scalable features.	It offers scalable feature support.

THE DISTINCTION BETWEEN KOTLIN AND SCALA

Scala is an extraordinarily multi-paradigm language that may range from being a far superior version of Java to a less desirable version of Haskell. This means that Scala libraries and codebases often use various writing styles, and it may be time-consuming to learn how to deal with them. In addition, it makes it tougher to standardize a group. To distinguish, Kotlin reduces its superiority to Java, resulting in more dependable libraries and the avoidance of many of these concerns.

Kotlin's Java interoperability is dependable. Scala allows Java interoperability, but it is difficult to use that wrapper written for the most popular Java libraries. JetBrains is a significant sponsor of Kotlin since they are the creators of some of the most acceptable app applications. Scala no longer has the same degree of major sponsorship; in fact, TypeSafe changed its name to Lightbend and withdrew from Scala. Individuals having a background in Java may remember Kotlin more quickly. Kotlin is a cross-platform, statically written, general-purpose programming language that supports type deduction. Kotlin is supposed to be fully compatible with Java, and its standard library's JVM version is based on the Java Lesson Library; however, sort induction allows its language structure to be more compact and performs well with Android. Android is easy to configure (many lines to the gradle record in Android Studio).

Scala's compatibility with Android requires further customization and a few aspects don't carry over at that point. We are at the beginning of an IoT uprising, with Android at the forefront; thus, it is crucial to have a cutting-edge language and 100% compatibility with the most popular mobile platform. Scala is a general-purpose programming language that supports functional programming and a suitable architecture for inactive sorting. Scala was concise, and many design decisions addressed Java's responses. Scala is a new computer program that blends the concepts of object-oriented and functional programming. Scala is so named because it is very scalable.

Scala might be a general-purpose programming language that underlies object-oriented and practical programming methods on a larger scale. Scala is influenced by the Java programming language and has the potential to be a robust static programming language. One of the most striking similarities between Scala and Java is that Scala may write in a manner similar to Java. It is also possible to use a subset of Java libraries inside Scala and several of Scala's third-party libraries.

Here are the main differences between Kotlin versus Scala.

Kotlin	**Scala**
An object-oriented dialect	Multi-paradigm programming language
There are fewer libraries, blogs, and direct, in other words, less quantified community support.	A larger community When compared to Kotlin, strengthens.
Kotlin Ordinarily, codes are concise and to the point.	Scala Codes are often larger.
Kotlin is a recognized Android dialect.	Scala isn't widely used for Android.
Kotlin Does not provide complete design coordination.	Complete assistance with pattern matching, macros, and higher-order forms
Practicality and consistency at the commercial level.	Extensive quantities of information pouring.
Kotlin is a free and open programming language.	It is a functional programming language hybrid.
It is a language that is statically typed.	Martin Odersky designed it.
It provides improved performance and a shorter runtime for all applications.	It also allows for object-oriented programming.
JetBrains introduced and launched Kotlin in 2016.	It can do higher-order functions.
It is employed in server-side applications.	It allows us to use all of the Java SDK's classes.

REPL IN SCALA

REPL stands for Read-Evaluate-Print-Loop and is an interactive command line interpreter shell. It only works for what it stands for. It first reads the expression provided as input on the Scala command line, then evaluates it and prints the result on the screen before returning to the reading loop. Previous results are automatically incorporated into the scope of the current expression as needed. At the prompt, the REPL reads expressions. In interactive mode, it then wraps them into an executable template before compiling and running the result.

REPL IMPLEMENTATION

- User code can encapsulate either an object or a class; the switch used is – Yrepl-class-based.
- Each line of input is compiled independently.
- Automatically created imports incorporate the dependencies on preceding lines.
- Scala's implicit import.
- An explicit import can use to regulate Predef.

Scala REPL may launch by entering the scala command in the console/terminal.

```
$scala
```

Let's look at how we can use Scala REPL to create two variables.

```
scala> val m, n = 4
m: Int = 4
n: Int = 4

scala> m + n
res0: Int = 8

scala>
```

In the first line, we set up two variables in Scala REPL. Scala REPL then printed these. We can see that it creates two variables of type Int with values internally. Then we ran the sum expression with two variables

specified. Scala REPL printed the total of expressions on the screen once again. Because it lacked a variable, it displayed it using its temporary variable solely with the prefix res. We may utilize these variables in the same way that we generated them.

We may retrieve further information about these temporary variables by executing the getClass method on them, as shown below.

```
scala> val m, n = 4
m: Int = 4
n: Int = 4

scala> m + n
res0: Int = 8

scala> res0.getClass
res1: Class[Int] = Int

scala>
```

We could conduct many tests using the Scala REPL on the run time, which would have taken a long time if we used an IDE.

SOME ADDITIONAL IMPORTANT REPL CHARACTERISTICS

- REPL's IMain is tied to $intp.
- The tab key is used for finishing.
- :load is used to load REPL input file.
- :javap is used to investigate class artifacts.
- -Yrepl-outdir is used to view class artifacts with the external tools.
- :power imports compiler components after entering compiler mode.
- :help returns a list of commands that can use to assist the user.

In this chapter, we covered Introduction and History, Setting up the environment in Scala, Hello World program, and Uniform Access Principle.

CHAPTER 2

Scala Basics

IN THIS CHAPTER

- Keywords and Identifiers
- Data Types and Variables
- Console and Identifiers
- Pattern Matching
- Comments and Command Line Argument
- Enumeration in Scala
- Ranges

In the previous chapter, we covered a Scala overview, and in this chapter, we will discuss the basics of Scala.

KEYWORDS IN SCALA

Keywords, often known as reserved words, are terms in a language that are used for internal processes or to indicate preset actions. As a result, these terms are not permitted to be used as variable names or objects. This will produce a compile-time error.

Example:

```
// Program to illustrate the keywords

//object, def, and var are valid keywords
```


DOI: 10.1201/9781003358527-2

```
object Main
{
    def main(args: Array[String])
    {
        var m = 10
        var n = 30
        var sum = m + n
        println("Sum of m and n is :"+sum);
    }
}
```

Scala includes the following keywords:

abstract	case	catch	class
def	do	finally	extends
false	final	else	for
lazy	if	implicit	import
forSome	match	new	Null
object	override	package	private
protected	throw	sealed	super
this	return	trait	Var
true	with	val	Try
while	type	yield	
–	:	=	=>
<–	<:	<%	>:
#	@		

Example:

```
// Program to illustrate the keywords

// class keyword is used to create new class def
keyword
// is used to create Function var keyword
// is used to create the variable
class PFP
{
    var name = "Shreya"
    var age = 19
    var branch = "Information technology"
    def show()
```

```
    {
    println("Hey my name: " + name + "and my age
is"+age);
    println("The Branch name: " + branch);
    }
}

// object keyword is used to define object new
keyword is used to
 create an object of the given class
object Main
{
    def main(args: Array[String])
    {
        var obj = new GFG();
        obj.show();
    }
}
```

IDENTIFIERS IN SCALA

Identifiers are used to identify objects in computer languages. An identifier in Scala can be a class name, method name, variable name, or object name.

Example:

```
class PFP{
    var m: Int = 19
}
object Main {
    def main(args: Array[String]) {
        var obj = new PFP();
    }
}
```

We have six Identifiers in the preceding program:

- PFP: Class name
- m: Variable name
- Main: Object name

- main: Method name
- args: Variable name
- obj: Object name

JAVA SCALA DEFINING RULES

A valid Scala identifier must follow certain constraints. If these criteria are not followed, we will obtain a compile-time error.

- Case matters with Scala identifiers.
- Scala does not support the usage of keywords as identifiers.
- Reserved words, such as $, cannot be used as identifiers.
- Scala only permitted the creation of Identifiers using the four types of identifiers listed below.
- The identification length is not limited, although it is best to keep it between 5 and 18 letters.
- Identifiers should not begin with numbers ([0-9]). "123peeks," for instance, is not an acceptable Scala identifier.

Example:

```
// program to demonstrate Identifiers

object Main
{

    // the main method
    def main(args: Array[String])
    {

    // Valid Identifiers
    var 'name' = "Rajat";
    var _age = 19;
    var Branch = "Computer Science";
    println("Name is:" +'name');
    println("Age is:" +_age);
    println("Branch is:" +Branch);
    }
}
```

SCALA IDENTIFIER TYPES

Scala recognizes four types of identifiers:

- **Alphanumeric identifiers:** Identifiers that begin with a letter (capital or small letter) or an underscore and are continued by letters, numerals, or underscores are known as alphanumeric identifiers.

  Valid alphanumeric identifiers include:

```
_PFP, peeks123, _1_Pee_23, Peeks
```

  Example:

```
// program to demonstrate Alphanumeric
Identifiers

object Main
{

    // the main method
    def main(args: Array[String])
    {

    // main, _name1, and Tuto_rial are the valid
alphanumeric identifiers
    var _name1: String = "PeeksforPeeks"
    var Tuto_rial: String = "Scala"

    println(_name1);
    println(Tuto_rial);
    }
}
```

- **Operator identifiers:** These are identifiers that contain one or more operator characters such as +, :, ?, ~, or #.

  Valid operator identifiers include:

```
+, ++
```

  Example:

```
// program to demonstrate Operator Identifiers

object Main
```

```
{

    // the main method
    def main(args: Array[String])
    {

    // main, m, n, and sum are valid
alphanumeric identifiers
    var m:Int = 20;
    var n:Int = 10;

    // Here, + is an operator identifier that is
used to add two values
    var sum = m + n;
    println("Display the result of +
identifier:");
    println(sum);
    }
}
```

- **Mixed identifiers:** Mixed identifiers have alphanumeric identifiers followed by an underscore and an operator identifier.

  Valid mixed identifiers include:

```
unary_+, sum_=
```

Example:

```
// program to demonstrate Mixed Identifiers

object Main
{

    // the main method
    def main(args: Array[String])
    {

    // num_+ is a valid mixed identifier
    var num_+ = 20;
    println("Display result of the mixed
identifier:");
    println(num_+);
    }
}
```

- **Literal identifiers:** These are identifiers that include an arbitrary string contained by back ticks (‘. . . .’).
  Valid mixed identifiers include:

```
'Peeks', 'name'
```

Example:

```
// Scala program to demonstrate
// Literal Identifiers

object Main
{

    // the main method
    def main(args: Array[String])
    {

    // 'name' and 'age' are the valid literal
identifiers
    var 'name' = "Rajat"
    var 'age' = 19
    println("Name is:" +'name');
    println("Age is:" +'age');
    }
}
```

SCALA DATA TYPES

A data type is a classification of data that tells the compiler what sort of value a variable has. For instance, if a variable has an int data type, it retains a numeric value. Scala data types are equivalent to Java in terms of length and storage. Data types are handled as objects in Scala; hence the initial letter of the data type is capitalized.

Scala data types are listed in the table below:

DataType	Default Value	Description
Boolean	False	True or False
Byte	0	8 bit signed value. The Range:-128 to 127
Short	0	16 bit signed value. The Range:-2^{15} to 2^{15}-1
Char	‘\u000’	16 bit unsigned unicode character. The Range:0 to 2^{16}-1
Int	0	32 bit signed value. The Range:-2^{31} to 2^{31}-1

(Continued)

DataType	Default Value	Description
Long	0L	64 bit signed value. The Range:-2^{63} to 2^{63}-1
Float	0.0F	32 bit IEEE 754 single Precision float
Double	0.0D	64 bit IEEE 754 double Precision float
String	null	Sequence of character
Unit	–	Coincides to no value.
Nothing	–	It is a subclass of all other types and has no value.
Any	–	It is a supertype of all the other types
AnyVal	–	It serves as a value types.
AnyRef	–	It serves as reference type.

Scala, unlike Java, does not have the idea of primitive type.

Example:

```
// program to illustrate the Datatypes
object Test
{
def main(args: Array[String])
{
    var m: Boolean = true
    var m1: Byte = 143
    var m2: Float = 3.45673f
    var m3: Int = 3
    var m4: Short = 29
    var m5: Double = 2.73846513
    var m6: Char = 'M'
    if (m == true)
    {
    println("boolean:peeksforpeeks")
    }
    println("byte:" + m1)
    println("float:" + m2)
    println("integer:" + m3)
    println("short:" + m4)
    println("double:" + m5)
    println("char:" + m6)
}
}
```

LITERALS IN SCALA

This section will go through the various sorts of literal used in Scala.

- **Literal integral:** These are often of the int or long type ("L" or "I" suffixed). Some examples of permissible integral literals are:

```
04
0
60
219
0xFFFFFFFF
0793L
```

- **Floating-point literals:** These are of the float type (the suffix "f" or "F" is used) and of the double type.

```
0.9
1e40f
3.14154f
1.0e100
.5
```

- **Boolean literals:** These are only true and false and are of the Boolean type.

- **Literals for the symbol:** Symbol is a case class in Scala. In symbol literal, a "Y" is the same as scala.

```
package scala
final case class Symbol private (name: String)
{
override def toString: String = "'" + name
}
```

- **Character literals:** A character literal in Scala is a single character surrounded by single quotes. Some characters are printable unicode characters as well as escape characters. Here are a few examples of valid literal:

```
'\b'
'a'
'\r'
'\u0027'
```

- **String literals:** In Scala, string literals are a sequence of characters surrounded by double quotes. Here are some examples of valid literal:

```
"welcome to \n peeksforpeeks"
"\\This is a tutorial of Scala\\"
```

- **Null values:** A null value is of scale in Scala. It is adaptable with any reference type since it is a null type. It is a reference value pointing to a special "null" object.
- **Multi-line literals:** In Scala, multi-line literals are a sequence of characters surrounded by triple quotes. Other control characters are acceptable in this new line. The following are some examples of valid multi-line literals:

```
"""Hey welcome to peeksforpeeks\n
this is a tutorial of \n
scala programing language"""
```

SCALA VARIABLES

Variables are nothing more than storage places. Every variable has a name and holds a known and unknown piece of information called value. A variable can be defined by its data type and name; a data type is in charge of allocating memory for the variable. Variables in Scala are classified into two types:

- Mutable Variables
- Immutable Variables

Let's take a closer look at each of these factors.

MUTABLE VARIABLE

Mutable variables allow us to modify the value of a variable after it has been declared. The var keyword is used to define mutable variables. Because data types are viewed as objects in Scala, the initial letter of the data type should capitalize.

Syntax:

```
var Variablename: Datatype = "value";
```

Example:

```
var name: String = "peekforpeeks";
```

In this case, the name is the variable's name, a string is the variable's data type, and peeksforpeeks is the value stored in memory.

Variable may also be defined as follows:

Syntax:

```
var variablename = value
```

Example:

```
var value = 40

//works without error
value = 32
```

The variable's name is used as the value here.

IMMUTABLE VARIABLES

These variables do not allow us to change their value after they have been declared. The val keyword is used to define immutable variables. Because data types are viewed as objects in Scala, the initial letter of the data type should capitalize.

Syntax:

```
val Variablename: Datatype =  "value";
```

Example:

```
val name: String = "peekforpeeks";
```

A name is the variable's name, a string is the variable's data type, and peeksforpeeks is the value stored in memory.

A variable may also be defined as follows:

Syntax:

```
val variablename = "value"
```

Example:

```
val value = 40
```

```
//it will give error
value = 32
```

Here, the value is the variable's name.

SCALA VARIABLE NAMING RULES

- The variable name should write in lowercase.
- A variable name can contain a letter, a numeric, and two special characters (the underscore (_) and the dollar symbol ($).
- The keyword or reserved term must not appear in the variable name.
- The variable name must begin with the alphabet letter.
- White space is not permitted in variable names.

Note that while Scala enables multiple assignments, we can only use them with immutable variables.

Example:

```
val(name1:Int, name2:String) = pair
(2, "peekforpeeks")
```

SCALA VARIABLE TYPE INFERENCE

Inference based on variable type inference is supported in Scala. Variable type inference assigns values to variables without describing their data types; the Scala compiler automatically determines which values correspond to which data types.

Example:

```
var name1=40;
val name2="peeksforpeeks";
```

In this case, name1 is an int by default, and name2 is a string by default.

println, printf, and readLine IN SCALA

Console provides functions for showing the specified values on the terminal; for example, we may post to the display using print, println, and printf. The scala.io.StdIn function is also used to read the data from the Console. It is even beneficial in developing interactive programs.

Let us go through it in depth and then look at some instances.

- **println:** It is used to write values to the console and compute a trailing newline. We can pass it any type as an argument.
- **print:** print is the same as println, but it does not calculate any trailing lines. It moves the data to the start of the line.
- **printf:** This is useful for writing format strings and inserting extra arguments.

Example:

```
// program of print functions

// Creation of an object
object PfP
{

    // the main method
    def main(args: Array[String])
    {

        // Applying console with the println
        Console.println("PeeksfoPeeks")

        // Display the output with no
        // the trailing lines
        print("CS")
        print("_portal")

        // Used for newline
        println()

        // Display the format string
        printf("Age = %d", 24)
    }
}
```

- **readLine():** readLine() is a function that takes input from the keyboard in the form of a String pattern.

Example:

```
// program of readLine() method

// Creation of an object
object PfP

{
// the main method
def main(args: Array[String])

{
// Applying loop
while (true) {

    // Reads line from the Console
    val result = scala.io.StdIn.readLine()

    // Display string that user gives
    printf("Enter String: %s", result)

    //prints the newline
    println()
    }
}
}
```

The while loop, in this case, is indefinite, and after providing user inputs, the variable will hold that string. If we offer any further input, the variable will contain that input.

PATTERN MATCHING IN SCALA

Pattern matching is a method of detecting the presence of a specified pattern in a given sequence of tokens. It is the most commonly used Scala feature. It is a method for comparing a value to a pattern. It is comparable to the Java and C switch statements.

Instead of a switch statement, the term "match" is used here. To make "match" available to all objects, it is always declared in Scala's root class. This might include a list of options. Each alternative will begin with the case keyword. Each case statement contains a pattern and one or more expressions are evaluated if the given pattern is matched. The arrow character (=>) is used to differentiate the pattern from the phrases.

First example:

```
// program to illustrate pattern matching

object PeeksforPeeks {

    // the main method
    def main(args: Array[String]) {

    // calling the test method
    println(test(1));
    }

// method containing the match keyword
def test(x:Int): String = x match {

    // if the value of x is 0,
    // this case will execute
    case 0 => "Hello, Everyone"

    // if the value of x is 1, this case will
execute
    case 1 => "Are we learning Scala?"

    // if x doesn't match any sequence, then this
case will execute
    case _ => "Good Bye"
}
}
```

Explanation: In the above code, if the value of x passed in the test method call corresponds to any of the situations, the expression within that case is evaluated. We are passing 1 here; therefore case 1 will be considered. case_ => is the default case, which is run if x is not 0 or 1.

Second example:

```
// program to illustrate the pattern matching

object PeeksforPeeks {

    // the main method
    def main(args: Array[String]) {
```

```
    // calling the test method
    println(test("Peeks"));
    }

// method containing match keyword
def test(x:String): String = x match {

    // if the value of x is "P1",
    // this case will be executed
    case "P1" => "PFP"

    // if the value of x is "P2",
    // this case will execute
    case "P2" => "Scala Tutorials"

    // if x does not match any sequence,
    // then this case will execute
    case _ => "Default Case Execute"
}
}
```

IMPORTANT NOTE

- At least one case clause must be present for each matching keyword.
- If none of the other instances match, the final "_" case will be executed. Cases are sometimes known as alternatives.
- There is no break statement in pattern matching.
- Pattern matching always yields some results.
- Match blocks are expressions rather than statements. This implies that they look at the body of the case that matches. This is a crucial aspect of functional programming.
- Pattern matching may be used for value assignment and comprehension in addition to matching blocks.
- With the first match policy, pattern matching may match any type of data.
- Each case statement returns the value, and the entire match statement functions to deliver a matched value.
- Using "|," we may test several values in a single line.

SCALA COMMENTS

In our code, comments are entities that the interpreter/compiler ignores. We often utilize them to explain the code and conceal code specifics. This indicates that comments will not include in the code. It will not be run; rather, it will use to explain the code thoroughly.

Scala comments, in other terms, are statements that the compiler or interpreter does not execute. The comments can use to explain or offer information about a variable, class, method, or sentence. This can also be used to conceal software code.

There are three sorts of comments in Scala:

- Singleline comments
- Multi-line comments
- Documentation comments

Each kind will be explained in detail, including syntax and examples:

SINGLELINE COMMENTS

When we only require one line of a remark in Scala, that is, when we just want to create a singleline comment, we may use the characters '//' before the comment. These characters will remark the line.

Syntax:

```
//Comments here(Only the text in this line is
considered a comment)
```

Example:

```
// This is the singleline comment

object MainObject
{
    def main(args: Array[String])
    {
        println("Singleline comment above")
    }
}
```

MULTILINE COMMENTS

A multiline comment can be used if our comment exceeds more than one line. We surround the comment with the letters '/*' and '*/'. We insert text between these characters, which becomes a comment.

Syntax:

```
/*Comment starts
continue
continue
.
.
.
.
.
Comment ends*/
```

Example:

```
// program to show the multiline comments

object MainObject
{
    def main(args: Array[String])
    {
        println("Multiline comments below")
    }

    /*Comment line1
    Comment line2
    Comment line3*/
}
```

DOCUMENTATION COMMENTS

A documentation comment is used to facilitate easy access to documentation. The compiler uses these comments to document the source code. The syntax for making a documentation comment is as follows:

Syntax:

```
/**Comment start
*
```

```
*tags are used in order to specify the parameter
*or method or heading
*HTML tags can also use
*such as
*
*comment ends*/
```

Example:

```
// program to show the Documentation comments

object MainOb
{
    def main(args: Array[String])
    {
        println("Documentation comments ")
    }

    /**
    * This is peek for peeks
    * peeks coders
    *
    */
}
```

To declare such a comment, write the letters '/**', type something, or press. As a result, every time we press enter, the IDE will be marked with a '*'. After one of the carets(*), write '/' to terminate a comment.

SCALA COMMAND LINE ARGUMENT

Command-Line Arguments are the arguments passed to the main() procedure by the user or programmer. The main() function is the program's starting point. The main() function receives a string array.

Syntax:

```
def main(args: Array[String])
```

The args array, made accessible to us implicitly when we extend App, is used to retrieve our Scala command-line arguments. Here's an illustration.

First example: Print all of the specified items.

```
// Program on the command line argument
object CMDExample
{
    // the main method
    def main(args: Array[String])
    {
        println("Command Line Argument Example");

        // We pass anything at runtime
        // that will print on the console
        for(arg<-args)
        {
            println(arg);
        }
    }
}
```

Save the program CMDExample.scala first, then open CMD/Terminal and navigate to the directory where we saved our Scala program.

To compile and run the above program on the terminal, use the following commands:

```
Compile: scalac CMDExample.scala
Execute: scala CMDExample Welcome To PeeksforPeeks!
```

Second example:

```
// Program on the command line argument
object CMDExample
{
    // the main method
    def main(args: Array[String])
    {
        println("Command Line Argument Example");

        // We pass anything at runtime
        // that will print on the console
        println(args(0));
        println(args(2));
    }
}
```

SCALA ENUMERATION

In computer languages, enumerations are used to express a collection of named constants. For further information on enumerations, see enumeration (or enum) in C and enum in Java. Scala includes an enumeration class that we may modify to construct our enumerations.

Enumeration declaration in Scala:

```
// simple scala program of the enumeration

// Creation of an enumeration
object Main extends Enumeration
{
    type Main = Value

    // Assigning the values
    val first = Value("Action")
    val second = Value("Horror")
    val third = Value("Romance")
    val fourth = Value("Comedy")

    // the main method
    def main(args: Array[String])
    {
        println(s"The Main Movie Genres = ${Main.
values}")
    }
}
```

Important enumeration points:

- Unlike Java or C, there is no enum keyword in Scala.
- Scala includes an Enumeration class that we may modify to construct our enumerations.
- Every enumeration constant represents an enumeration object.
- The evaluation's val members are specified as enumeration values.
- Many functions were inherited when we expanded the enumeration class. Identification is one of them.
- We can change the members.

Printing a specific enumeration element:

```
// program Printing particular element of enumeration

// Creation of ab enumeration
object Main extends Enumeration
{
    type Main = Value

    // Assigning the values
    val first = Value("Action")
    val second = Value("Horror")
    val third = Value("Romance")
    val fourth = Value("Comedy")

    // the main method
    def main(args: Array[String])
    {
        println(s"Third value = ${Main.third}")
    }
}
```

Printing a specific enumeration element:

```
// program of Printing default ID of any element in
enumeration

// Creation of an Enumeration
object Main extends Enumeration
{
    type Main = Value

    // Assigning the Values
    val first = Value("Action") // ID = 0
    val second = Value("Horror") // ID = 1
    val third = Value("Romance") // ID = 2
    val fourth = Value("Comedy") // ID = 3

    // the main Method
    def main(args: Array[String])
    {
        println(s"ID of the third = ${Main.third.id}")
    }
}
```

Enumeration values that match:

```
// program of Matching values in the enumeration

// Creation of an Enumeration
object Main extends Enumeration
{
    type Main = Value

    // Assigning the Values
    val first = Value("ACtion")
    val second = Value("Horror")
    val third = Value("Romance")
    val fourth = Value("Comedy")

    // the main Method
    def main(args: Array[String])
    {
        Main.values.foreach
        {
            // Matching values in the Enumeration
            case d if ( d == Main.third ) =>
            println(s"Favourite type of the Movie =
$d")
            case _ => None
        }
    }
}
```

Changing the value's default IDs:

The values are written in the order specified by the ID. These ID values can be any integer.

These IDs do not have to be in any specific sequence.

```
// program of Changing default IDs of values

// Creation of an Enumeration
object Main extends Enumeration
{
    type Main = Value

    // Assigning the Values
```

```
    val first = Value(0, "Action")
    val second = Value(-1, "Horror")
    val third = Value(-3, "Romance")
    val fourth = Value(4, "Comedy")

    // the main Method
    def main(args: Array[String])
    {
        println(s" The Movie Genres = ${Main.values}")
    }
}
```

VARIABLE SCOPE IN SCALA

The variable declaration defines the variable's name that will keep in memory, and memory may be accessed using this variable name. Scala variables have three forms of scope.

- Fields
- Method Parameters
- Local Variables

Let's go through each one in depth.

FIELDS

If we declare these variables with the appropriate access modifiers, we may access them from any method within the object and from outside the object. A field can be changeable or immutable, and it can define with "var" or "val."

Example:

```
// Scala program of field scope for the Scala
variable

// class created with the field
// variables m and n.
class disp
{
    var m = 20.3f
    var n = 8f
```

```
    def windet()
    {
        println("Value of m : "+m)
    }
        println("Value of n : "+n);
}

object Example
{
    // the main method
    def main(args:Array[String])
    {
        val Example = new disp()
        Example.windet()
    }
}
```

The preceding example creates a disp class with field variables x and y. These variables may be accessible within a method and called from an object by generating a reference to them.

METHOD PARAMETERS

When we call a method, we use these variables to send a value within the method. They can be accessed both inside and outside the method if the object is referenced from outside the method. These variables can change at any time using the term "val."

Example:

```
// program of Method scope for the Scala variable

// class created with Method variables k1 and k2.
class rect
{
    def mult(k1: Int, k2: Int)
    {
        var result = k1 * k2
    println("The Area is: " + result);
    }
}

object Area
```

```
{
    // the main method
    def main(args:Array[String])
    {
        val su = new rect()
        su.mult(15, 20)
    }
}
```

In the preceding example, a rect class is built with a mult function that accepts two method argument variables, k1 and k2. The area object is created, and the rect method is called by giving the values of variables k1 and k2 to it.

LOCAL VARIABLES

These variables are declared within a method and are only available within it. These variables can be both changeable and immutable by using the "var" and "val" keywords.

Example:

```
// program of Method scope for the Scala variable

// class created with the Local
// variables k1 and k2.
class Area
{
    def mult()
    {
        var(k1, k2) = (5, 70);
        var k = k1 * k2;
        println("The Area is: " + s)
    }
}

object Test
{
    // the main method
    def main(args:Array[String])
    {
        val ex = new Area()
        ex.mult()
    }
}
```

The above example demonstrates the class Area with local variables s1, s2, and k within function mult. Outside of the procedure, these variables are inaccessible.

RANGES IN SCALA

Scala defines the Range as an orderly series of evenly separated Integers. It is useful in providing more strength with fewer methods; therefore, operations are very fast.

Here are some key points:

- The for loops can use the Ranges for iteration.
- It may be achieved by the use of several specified methods such as until, by, and to.
- It is defined by three constants, which are:start, end, and increment value.

Syntax:

```
val range = Range(a, b, c)
```

Where x represents the lower limit, y represents the higher limit, and z represents the increment.

Example:

```
// program for Ranges

// Creation of object
object PFP
{

    // the main method
    def main(args: Array[String])
    {

        // applying the range method
        val a = Range(4, 12, 1)
```

```
        // Display the given range
        println(a)

        // Display the starting value
        // of a given range
        println(a(0))

        // Display the last value
        // of given range
        println(a.last)
    }
}
```

Operations Performed on Ranges

- If we want a range that includes the final value, we may use the till method. The until and Range methods serve the same function.

Example:

```
// program for Ranges

// Creation of object
object PFP
{

    // the main method
    def main(args: Array[String])
    {

        // applying the range method
        val a = Range(0, 12, 3)

        // applying the until method
        val b = 0 until 12 by 3

        // Displays true if both methods are
equivalent
        println(a == b)
    }
}
```

- The Range's upper bound can be made inclusive.

  Example:

```
// program for Ranges

// Creation of object
object PFP
{

    // the main method
    def main(args: Array[String])
    {

        // applying the range method
        val a = Range(2, 9)

        // Including the upper bound
        val b = a.inclusive

        // Displays all elements of the range
        println(b)
    }
}
```

- We may use the to method to get a range of integer values; both to and inclusive Ranges are similar.

  Example:

```
// Scala program for Ranges

// Creating object
object GFG
{

    // Main method
    def main(args: Array[String])
    {

        // applying the range method
        val a = Range(2, 9)
```

```
        // Including the upper bound
        val b = a.inclusive

        // applying 'to' method
        val c = 2 to 9

        // Displays true if both methods are
equal
        println(b == z)
    }
}
```

This chapter covered Keywords and Identifiers, Data Types and Variables, Console, and Identifiers. We also talked about Pattern Matching, Comments and Command Line Arguments, Enumeration, and Ranges.

CHAPTER 3

Scala Control Statements

IN THIS CHAPTER

- Decision Making
- Loops
- Break Statement in Scala
- Literals
- yield Keyword
- Type Inference

In the previous chapter, we covered Scala basics, and in this chapter, we will discuss control statements.

MAKING DECISIONS (if, if-else, Nested if-else, if-else if) IN SCALA

Making decisions in programming is analogous to making decisions in real life. When a condition is met, a piece of code is performed in decision making. Control flow statements are another name for them. Scala uses control statements to direct program execution flow based on particular criteria. These trigger the execution flow to progress and branch based on changes in a program's state.

DOI: 10.1201/9781003358527-3

Scala's conditional statements are as follows:

- if
- if-else
- Nested if-else
- if-else if Ladder

if Statement

Across all decision-making statements, the "if" statement is the simplest. In this statement, the piece of code is run once the specified condition is true, and it is not performed if the condition is false.

Syntax:

```
if(condition)
{
    // Code executed
}
```

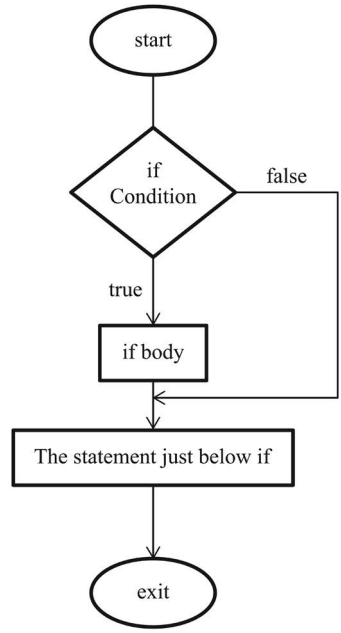
After examination, the condition will be either true or false. The if statement takes boolean values; if the value is true, the block of statements behind it is executed.

If we do not include the curly brackets "{and}" after if(condition), the if statement will regard the preceding statement as within its block by default.

Example:

```
if(condition)
   statement1;
   statement2;
.........
// if the condition is true if block
// will consider only statement1 inside its block.
```

Flowchart:



Statement of if.

Example:

```
// program to illustrate if statement
object Test {

    // the main Method
    def main(args: Array[String]) {

        // taking variable
        var m: Int = 60

        if (m > 40)
        {

            // statement will execute as m > 40
            println("HelloEveryone")
        }
    }
    }
```

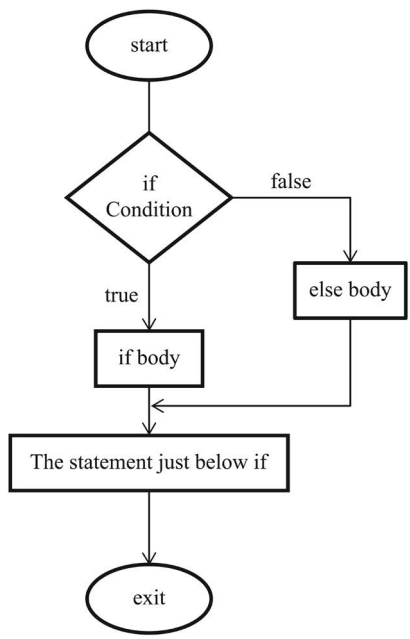
if-else Statement

The if statement alone indicates that if a condition is true, a block of statements will be executed; if the condition is false, the block of statements will not be executed. But what if the condition is false and we want to do else statement. The else statement follows. When the condition is false, we may use the else statement in conjunction with the if statement to run a block of code.

Syntax:

```
if (condition)
{
    // Executes this block if condition is true
}

else
{
    // Executes this block if condition is false
}
```

Flowchart:



Statement of if.

Example:

```
//program to illustrate the if-else statement
object Test {

    // the main Method
    def main(args: Array[String]) {

        // taking variable
        var m: Int = 650

        if (m > 798)
        {

            // This statement will not
            // execute as m > 798 is false
            println("HelloEveryone")
        }

        else
        {

            // statement will execute
            println("Sudo Placement")
        }
    }
    }
```

Nested if-else Statement

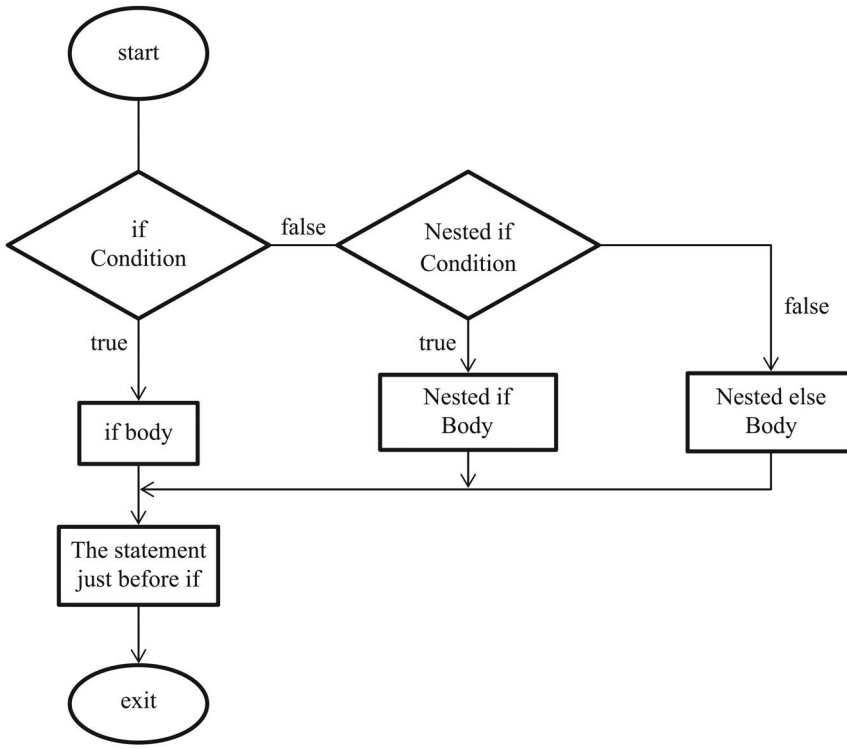
An if statement that targets another if-else expression is referred to as a nested if. An if-else statement that is nested indicates that it is included within an if statement or an else statement. We may nest if-else statements within if-else statements in Scala.

Syntax:

```
// Executes when the condition1 is true
if (condition1)
{

   if (condition2)
   {
```

```
       // Executes when the condition2 is true
    }

  else
  {

     // Executes when the condition2 is false
  }

}

// Executes when the condition1 is false
else
{

   if (condition3)
   {

      // Executes when the condition3 is true
   }

  else
  {

     // Executes when the condition3 is false
  }

}
```

Flowchart:



Statement of nested-if-else

Example:

```
// program to illustrate nested if-else statement
object Test {

    // the main Method
    def main(args: Array[String]) {

        // taking the three variables
        var x: Int = 70
        var y: Int = 40
        var z: Int = 100
```

```
    // condition1
    if (x > y)
    {
        // condition2
        if(x > z)
        {
            println("x is largest");
        }

        else
        {
            println("z is largest")
        }

    }

    else
    {

        // condition3
        if(y > z)
        {
            println("y is largest")
        }

        else
        {
            println("z is largest")
        }
    }
}
}
```

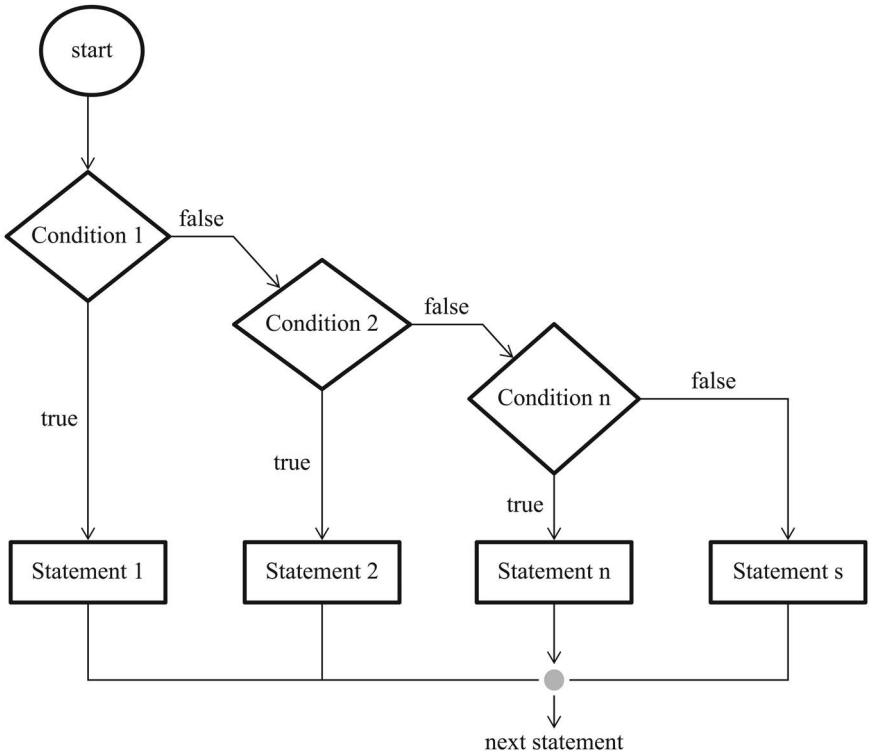
if-else if Ladder

A user can select from several alternatives here. The if statements are performed in the order listed. When one of the if conditions are met, the statement associated with that if is performed, and the rest of the ladder is skipped. If none of the requirements are met, the last else expression is performed.

Syntax:

```
if(condition_1)
{

     // this block will execute when condition1 is
true
}

else if(condition2)
{

    // this block will execute when the condition2
is true
}
.
.
.

else
{

      // this block will execute when none of
condition is true
}
```

Flowchart:



Statement of if-else-if.

Example:

```
//program to illustrate if-else-if ladder
object Test {

    // the main Method
    def main(args: Array[String]) {

        // Taking variable
        var value: Int = 60

        if (value == 30)
        {
```

```
            // print the "value is 30" when
            // the above condition is true
            println("Value is 30")
        }

        else if (value == 20)
        {

            // print "value is 20" when the
            // above condition is true
            println("Value is 20")
        }

        else if (value == 50)
        {

            // print "value is 50" when
            // the above condition is true
            println("Value is 50")

        }

        else
        {

            // print "No Match Found"
            // when all the condition is false
            println("No Match Found")
        }
    }
    }
```

LOOPS IN SCALA (while, do..while, for, Nested Loops)

In computer languages, looping is a feature that allows the execution of a set of instructions/functions repeatedly while some condition is true. Loops make the job of the coder easier. Scala supports many loop types to handle conditional situations in programs. Scala's loops are as follows:

- while Loop
- do..while Loop

- for Loop
- Nested Loops

while Loop

A while loop often accepts a condition in the form of parenthesis. If the condition is True, the program contained within the body of the while loop is executed. A while loop is used when we do not understand how many times we want the loop to run but know the loop's termination condition. Because the condition is verified before performing the loop, it is also known as an entry controlled loop. The while loop is similar to a repeating if statement.

Syntax:

```
while (condition)
{
    // Code to execute
}
```

The while loop begins with a condition check. If true, the loop body statements are performed; else, the first sentence after the loop is executed. As a result, it is also known as an entry control loop.

The statements in the loop body are performed once the condition is determined to be true. Normally, the statements include an updated value for the variable under consideration for the following iteration.

The loop stops when a condition is met, signaling the end of its life cycle.

Example:

```
// program to illustrate the while loop
object whileLoopDemo
{

    // the main method
    def main(args: Array[String])
    {
        var m = 1;

        // Exit when x becomes greater than 4
        while (m <= 4)
```

```
        {
            println("Value of m: " + m);

            // Increment the value of m for
            // next iteration
            m = m + 1;
        }
    }
}
```

Infinite while Loop

A while loop can continue indefinitely, which means it has no end condition. In other words, some circumstances always remain true, causing the while loop to continue endlessly, or it never finishes.

Example: The following software will print the provided statement indefinitely while displaying the runtime error Killed (SIGKILL) in the online IDE.

Example:

```
// program to illustrate Infinite while loop
object infinitewhileLoopDemo
{

    // the main method
    def main(args: Array[String])
    {
        var m = 1;

        // this loop will never terminate
        while (m < 5)
        {
            println("HelloEveryone")
        }
    }
}
```

do..while Loop

A while loop is identical to a do…while loop. The only distinction is that the do..while loop is run at least once. After the initial execution, the condition is verified. When we want the loop to execute at least once, we use a

do..while loop. Because the condition is verified after performing the loop, it is also called the exit controlled loop.

Syntax:

```
do {

    // statements to Execute

    } while(condition);
```

Example:

```
// program to illustrate the do..while loop
object dowhileLoopDemo
{

    // the main method
    def main(args: Array[String])
    {
        var m = 20;

        // using do..while loop
        do
        {
            print(a + " ");
            m = m - 1;
        }while(m > 0);
    }
}
```

for Loop

While the functionality of a for loop is identical to that of a while loop, the syntax is different. When the number of times loop statements are to be run is known ahead of time, for loops are chosen. We will cover other versions of the "for loop in Scala" in future posts. It is essentially a repetition control structure that lets the programmer construct a loop that must execute a specific number of times.

Example:

```
// program to illustrate for loop
object forloopDemo {
```

```
    // the main Method
    def main(args: Array[String]) {

        var m = 0;

        // for loop execution with the range
        for(m <- 1 to 7)
        {
            println("Value of m is: " + m);
        }
    }
    }
```

Nested Loops

The nested loop is a loop that contains a loop within a loop. It can have a for loop within a for loop or a while loop within a while loop. It is also feasible for a while loop to include a for loop and vice versa.

Example:

```
// program to illustrate the nested loop
object nestedLoopDemo {

    // the main Method
    def main(args: Array[String]) {

        var m = 5;
        var n = 0;

        // outer while loop
        while (m < 7)
        {
        n = 0;

        // inner while loop
        while (n < 7 )
        {

            // printing values of a and b
            println("Value of m = " +m, " n = "+n);
            n = n + 1;
        }
```

```
        // new line
        println()

        // incrementing value of m
        m = m + 1;

        // displaying the updated value of m
        println("The Value of m Become: "+m);

        // new line
        println()
        }

    }
    }
```

SCALA FOR LOOP

For loops are also referred to as for-comprehensions in Scala. A for loop is a loop control model that enables us to design a loop that will run numerous times. The loop allows us to complete n number of steps in a single line.

Syntax:

```
for(w <- range){
    // Code...
    }
```

Here, w is a variable; the <- operator is referred to as a generator since it is used to create individual values from a range, and the range is the value that contains the starting and ending values. m to n or m until n can use to denote the range.

for Loop Using to

When we need to display the values 0 to n in for loop, we may use to. In other terms, when we use for loop, it contains both the start and finish values, as shown in the following program, and it outputs from 0 to 10, rather than 0 to 9 as in until.

Example:

```
// program to demonstrate how to create for loop
using to
```

```
object Main
{
    def main(args: Array[String])
    {
        println("Value of k is:");

        // Here, for loop, starts from 0 and ends
at 10
        for( k <- 0 to 10)
        {
            println(k);
        }
    }
}
```

for Loop Using until

When we want to display the value from 0 to n–1, we may utilize till in the for loop. In other words, until the for loop excludes the last value, as illustrated in the following program, it prints just from 0 to 9, not 0 to 10.

Example:

```
// demonstrate how to create for loop using until
object Main
{
    def main(args: Array[String])
    {
        println("Value of k is:");

        // for loop starts from the 0 and ends at
10
        for( k <- 0 until 10)
        {
            println(k);
        }
    }
}
```

MULTIPLE VALUES IN for-loop

Multiple ranges can also be used in a single for-loop. A semi-colon separates these ranges (;). Let me discuss this with an instance. In the following example, we combine two separate ranges into a single loop: m <- 0 to 4; n<- 9 until 10.

Example:

```
// demonstrate how to create multiple ranges in
for loop
object Main
{
    def main(args: Array[String])
    {

    // for loop with the multiple ranges
        for( m <- 0 to 4; n<- 9 until 10 )
        {
            println("The Value of m is :" +m);
            println("The Value of n is :" +n);
        }
    }
}
```

USING for-loop WITH COLLECTIONS

In Scala, we may utilize the for-loop with collections such as List. It gives a quick way to iterate across the collections.

Syntax:

```
for(x <- List){
    // Code..
    }
```

Example:

```
// demonstrate how to use for loop with collection
object Main
{
    def main(args: Array[String])
    {
        var rank = 0;
        val ranklist = List(11, 12, 13, 14, 15,
16, 17, 18, 19, 20);

        // For loop with the collection
        for( rank <- ranklist){
```

```
            println("The Author rank is : " +rank);
        }
    }
}
```

USING for-loop WITH filters

For-loop in Scala allows us to filter some entries from a specified collection using one or more if expressions.

Syntax:

```
for(x<- List
if condition1; if condition2; if condition3;......)
{
// code..
}
```

Example:

```
// demonstrate how to use for loop with the
filters
object Main
{
    def main(args: Array[String])
    {
        var rank = 0;
        val ranklist = List(11, 12, 13, 14, 15,
16, 17, 18, 19, 20);

        // For loop with the filters
        for( rank <- ranklist
        if rank < 8; if rank > 2 )
        {
            println("The Author rank is : " +rank);
        }
    }
}
```

Explanation: In the preceding example, the for loop filters the supplied collection using two filters. These filters exclude rankings that are less than 8 and larger than 2.

USING for-loop WITH yield

```
var output = for{ x<- List
    if condition1; if condition2;
    }
    yield x
```

Example:

```
// show how to use for loop with yields
object Main
{
    def main(args: Array[String])
    {
        var rank = 0;
        val ranklist = List(11, 12, 13, 14, 15,
16, 17, 18, 19, 20);

        // For loop with the yields
        var output = for{ rank <- ranklist
                    if rank > 5; if rank != 9 }
                    yield rank

        // Display the result
        for (rank <- output)
        {
            println("The Author rank is : " +
rank);
        }
    }
}
```

Explanation: In the above instance, the result is a variable containing a collection of all rank values. And the for loop only displays Author's ranks that are more than 5 but less than 9.

SCALA while AND do while Loop

In computer languages, looping is a feature that allows the execution of a set of instructions/functions repeatedly while some condition is true. Loops make the job of the coder easier. Scala supports several loop types; however, this chapter focuses on the while and do-while loops.

while Loop

While programming, we may encounter situations that need us to repeat until and unless a condition is fulfilled. The while loop is used in these instances. A while loop often accepts a condition in the form of parenthesis. If condition is True, the code within the while loop's body is run. The while loop can be used when we don't know how many repetitions we want the loop to run but know the loop's termination condition. The circumstance that causes the loop to terminate is known as the breaking condition.

Syntax:

```
while (condition)
{
    // Code to execute
}
```

Example: While loop execution

```
// program of while loop

// Creation of object
object PFP
{
    // the main method
    def main(args: Array[String])
    {
        // the variable declaration (assigning
5 to m)
        var m = 5

        // loop execution
        while (m > 0)
        {
            println("m is : " + m)
            m = m - 1;
        }
    }
}
```

Example: Finding an element in an array

```
// program of while loop

// Creation of object
object PFP
{
// the main method
def main(args: Array[String])
{
    // variable declaration (assigning 5 to m)
    var m = Array("do_while", "for", "while")
    var index = 0

    // loop execution
    while (index < m.length)
    {
        if(m(index) == "while")
            println("The index of while is " +
index)
        index = index + 1
    }
}
}
```

do while Loop

A do…while loop is identical to while loop. The only difference is that the do..while loop is executed at least once. After the initial execution, the condition is verified. When we want the loop to execute at least once, we use a do..while loop. Because the condition is verified after the loop is also known as the exit controlled loop. The while loop condition is inserted at the top of the loop. Because the condition in the do while loop is placed at the end, all statements in the do while loop are executed at least once.

Syntax:

```
do {

    // statements to Execute

    } while(condition);
```

Example: do while loop execution

```
// program of do-while loop

// Creating object
object PFP
{
    // the main method
    def main(args: Array[String])
    {
        // the variable declaration (assigning 5 to
m)
        var m = 5;

        // loop execution
        do
        {
            println("m is : " + m);
            m = m - 1;
        }
        while (m > 0);
    }
}
```

Example: Running a loop till we find a string in the Array

```
// program for do-while loop

// Creation of object
object PFP
{
    // the main method
    def main(args: Array[String])
    {
        // Declaration of an array
        var m = Array("hello", "This", "is",
"Computertutorials", "bye")
        var str = "bye"
        var x = 0

        // loop execution
        do
```

```
        {
            println("program is saying " + m(x));
            x = x + 1;
        }
        while (m(x) != str);
    }
}
```

SCALA BREAK STATEMENT

To halt the program's loop execution, we utilize a break statement in Scala. The Scala programming language lacks the idea of a break statement (in versions above 2.8). Instead, it includes a break function, which is used to interrupt the execution of a program or a loop. Importing the scala.util.control.breaks._ package allows us to utilize the break function.

Syntax:

```
// import the package
import scala.util.control._

// create Breaks object
val loop = new breaks;

// loop inside the breakable
loop.breakable{

// Loop starts
for(..)
{
// code..
loop.break
}
}
```

OR

```
import scala.util.control.Breaks._
breakable
{
 for(..)
{
 code
```

```
 break
}
}
```

Example:

```
// program to illustrate the implementation of
break

// Importing the break package
import scala.util.control.Breaks._
object MainObject
{

// the main method
def main(args: Array[String])
{

    // breakable is used to prevent an exception
    breakable
    {
        for (m <- 1 to 12)
        {
            if (m == 8)

                // terminate the loop when the
value of m is equal to 8
                break
            else
            println(a);
        }
    }
}
}
```

BREAK IN Nested Loop

The break technique may also be used in nested loops.

Example:

```
// program to illustrate the implementation of the
break in the nested loop
```

```
// Importing the break package
import scala.util.control._

object Test
{

// the main method
def main(args: Array[String])
{
    var numb1 = 0;
    var numb2 = 0;
    val m = List(15, 20, 25);
    val n = List(30, 35, 40);

    val outloop = new Breaks;
    val inloop = new Breaks;

    // breakable is used to
    // prevent from exception
    outloop.breakable
    {
        for (numb1 <- m)
        {

            // print list m
            println(" " + num1);

            inloop.breakable
            {
            for (numb2 <- n)
            {

            //print list n
            println(" " + num2);

            if (numb2 == 35)
            {

            // inloop is break when
            // numb2 is equal to 35
            inloop.break;
            }
```

```
            }

        // inloop breakable
            }
        }

    // outloop breakable
    }
}
}
```

Explanation: In the above example, the starting value of numb1 and numb2 are both 0. Now, the outer for loop begins and prints 15 from the m list, followed by the inner for loop, which begins and prints 30, 35 from the n list. The inner loop terminates when the controls reach the numb2 == 35 condition, likewise for 20 and 25.

LITERALS IN SCALA

Literal/constant refers to any constant value that may give to a variable. The literals are a set of symbols used to describe a constant value in the code. In Scala, literals are classified as Character literals, String literals, Multiline String literals, Boolean literals, Integer literals, and Floating point literals.

LITERALS TYPES

1. **Integer literals:** When a suffix L or l is appended to the end of an Integer literal, it is of type Int or Long. Integer numbers are represented by the types Int and Long.

   Note:

   - It should note that the type Int has a range ranging from –231 to 230.
   - The Long type has a range of from –263 to 262.
   - When an Integer literal contains a number outside of this range, a compile time error occurs.

   The literals for integers are supplied in two ways:

   - **Decimal literals:** The allowable digits range from 0 to 9.

     ```
     val m = 47
     ```

- **Hexa-decimal literals:** The allowable digits range from 0 to 9, while the characters range from a to f. We can use both uppercase and lowercase letters.

```
// hexa-decimal number should be prefix with 0X
or 0x.
val x = 0xFFF
```

Example:

```
// program of integer
// literals

// Creation of an object
object integer
{

    // the main method
    def main(args: Array[String])
    {

        // decimal-form literal
        val m = 46

        // Hexa-decimal form the literal
        val n = 0xfF

        // Display the results in
        // the integer form
        println(m)
        println(n)
    }
}
```

2. **Floating point literals:** When a suffix F or f is added at the end, this type of literal becomes Double as well as Float, and we may further select the Double type by suffixing with d or D.

```
val x = 2.43159
```

Example:

```
// program of floating point literals
```

```
// Creation of object
object double
{

    // the main method
    def main(args: Array[String])
    {

        // the decimal-form literal
        val m = 4.256

        // It is also the decimal form of the
literal
        val n = 0213.34

        // Display the results
        println(m)
        println(n)
    }
}
```

3. **Character literals:** Character literals are either unicode characters that are printable or escape sequences.

```
val m = 'b'
//character literal in the single quote.

val m = '\u0051'
//uni-code representation of the character
literal,
//This uni-code represents Q.

val m = '\n'
//Escape sequence in the character literals
```

Example:

```
// program of character literal

// Creation of object
object literal
{
```

```
    // the main method
    def main(args: Array[String])
    {
        // Creating a character literal
        // in single quote
        val m = 'b'

        // uni-code representation of
        // character literal
        val n = '\u0051'

        // Escape sequence in the character
literals
        val o = '\n'

        // Display the results
        println(m)
        println(n)
        println(o)
    }
}
```

4. **String literals:** String literals are a series of characters enclosed in double quotes. String Interpolation allows for the seamless handling of String literals.

```
val m = "PfP"
```

Example:

```
// program of literals

// Creation of object
object literal
{

    // the main method
    def main(args: Array[String])
    {
        // Creation of a string
        // literal
        val m = "HelloEveryone"
```

```
        // Display the string literals
        println(m)
    }
}
```

5. **Multiline string literals:** Multiline string literals are also collections of characters that span many lines.

```
val m = """PfP"""
```

Example:

```
// program of multi-line string literals

// Creation of object
object literal
{

    // the main method
    def main(args: Array[String])
    {
        // Creating multiple
        // line string literal
        val m ="""HelloeEveryone
        is a
        computer graphics
        portal"""

        // Display the multiple lines
        println(m)

    }
}
```

6. **Boolean literals:** Boolean literals accept just two values, true and false, which are Boolean members.

```
val m = true
```

Example:

```
// program of Boolean literals

// Creation of object
```

```
object PfP
{

    // the main method
    def main(args: Array[String])
    {

        // Assigning the true
        val m = true

        // Assigning the false
        val n = false

        // Display the results
        println(m)
        println(n)
    }
}
```

yield KEYWORD IN SCALA

The yield keyword will produce a result when the loop iterations are completed. The for loop employed a buffer internally to hold iterated results, and after all, iterations were completed, it returned the end result from that buffer. It does not work in the same manner as an imperative loop works. The type of the returned collection is the same as the type we were iterating through, so a Map produces a Map, a List yields a List, and so on.

Syntax:

```
var result = for{ var m <- List}
yield m
```

The curly brackets have been used to keep the variables and conditions, and the outcome is a variable where all of the values of m are preserved in the form of a collection.

Example:

```
// program to illustrate yield keyword

// Creation of an object
object PFP
```

```
{
    // the main method
    def main(args: Array[String])
    {
        // Using the yield with for
        var print = for( m <- 1 to 10) yield m
        for(n<-print)
        {
            // Printing the result
            println(n)
        }
    }
}
```

In the preceding example, the for loop used with a yield statement generates a list sequence.

Example:

```
// program to illustrate yield keyword

// Creation of an object
object PFP
{
    // the main method
    def main(args: Array[String])
    {
        val m = Array( 8, 3, 1, 6, 4, 5)

        // Using the yield with for
        var print=for (n <- m if n > 4) yield e
        for(k<-print)
        {
            // Printing result
            println(k)
        }
    }
}
```

In the preceding example, the for loop combined with a yield statement creates an array. Because we say yield e, it's an Array[n1, n2, n3,. . . .]. e <- a is our generator, and if (e > 4) may be a guard that filters out numbers that don't appear to be bigger than 4.

TYPE INFERENCE IN SCALA

Scala Type Inference specifies the type of variable unnecessary, provided that type mismatches are handled. With type inference, we can spend less time writing out things that the compiler already knows. Because the Scala compiler can frequently infer the type of an expression, we don't need to specify it explicitly.

Let's start with the syntax for declaring immutable variables in Scala.

```
val variablename : Scala_datatype = value
```

Example:

```
// program of type interference
object Peeks
{

    // the main method
    def main(args: Array[String])
    {
        // prints double value
        val m : Double = 9.793
        println(m)
        println(m.getClass)

    }
}
```

The getClass function is used in the above example to output the variable's type to the console. The variable "m" in the preceding example is of the type double.

Scala, on the other hand, discovers the variable type without the user specifying it.

Example:

```
// program of type interference
object Peeks {

    // the main method
    def main(args: Array[String])
    {
```

```
        // type inference
        println("The Scala Data Types")
        val number = 5
        val bignumber = 100000000L
        val smallnumber = 2
        val doublenumber = 3.50
        val floatnumber = 3.50f
        val stringofcharacters = "This is a string
of the characters"
        val byte = 0xc
        val character = 'B'
        val empty = ()

        println(number)
        println(bignumber)
        println(smallnumber)
        println(doublenumber)
        println(floatnumber)
        println(stringofcharacters)
        println(byte)
        println(character)
        println(empty)

    }
}
```

It is worth noting that no specific data type is defined for the variables listed above.

SCALA FUNCTION TYPE INFERENCE

Now we'll look at how type inference works in Scala for functions.

Let's start with how functions are declared in Scala.

Syntax:

```
def functionname ([parameterlist]) : [returntype]
= {
    // function body

  }
```

Example:

```
// program of the multiply two numbers
object Peeks
{

    // the main method
    def main(args: Array[String])
    {

        // Calling the function
        println("The Product of two numbers is: "
+ Prod(6, 4));
    }


    // declaration and definition of the Product
function
    def Prod(x:Int, y:Int) : Int =
    {
        return x*y
    }
}
```

As we can see from the declaration, the specified return type in the preceding example is Int. Scala type inference automatically finds the function type without the user specifying it.

Example:

```
// program of the type interference
object Peeks
{

    def factorial(n: Int)= {

        var f = 1
        for(x <- 1 to n)
        {
            f = f * x;
        }
```

```
        f
    }

    // Driver-Code
    def main(args: Array[String])
    {
        println(factorial(6))
    }

}
```

The colon and return type are missing in the preceding example.

Also, in the preceding example, we removed the statement "return f" to "f" since we did not specify the return type.

The compiler displays the following error if "return f" is used instead of "f."

```
prog.scala:11: error: method factorial has return
statement; needs the result type
        return f
        ^
```

This demonstrates the potential of Scala type inference, but the compiler cannot infer a result type for recursive functions. The factorial function shown above can also implement recursively.

The factorial function is defined recursively below, with no type inference.

Example:

```
// program of using recursion
object Peeks {

    // the factorial function
    def factorial(n: Int): Int =
    {
        if (n == 0)
            return 1
        else
            return n * factorial(n-1)
    }
```

```
    // Driver-Code
    def main(args: Array[String])
    {
        println(factorial(6))
    }

}
```

Example: Consider Scala type inference.

```
// program of type interference
object Peeks
{

    // Defining function with the type interfrence
    def factorial(n: Int) =
    {
        if (n == 0)
            1
        else
            n * factorial(n-1)
    }

    // Driver-Code
    def main(args: Array[String])
    {
        println(factorial(6))
    }

}
```

This chapter covered Decision Making, Loops, Break statement in Scala, Literals, yield Keyword, and Type Inference.

CHAPTER 4

Scala OOP Concepts

IN THIS CHAPTER

- Class and Object in Scala
- Inheritance
- Operators
- Abstract Classes
- Singleton and Companion Objects
- Generic Classes
- Access Modifiers
- Type Casting
- Object Casting
- Object Equality
- Multithreading
- Constructors
- Extending a Class in Scala
- Polymorphism
- Field Overriding
- Abstract Type Members

DOI: 10.1201/9781003358527-4

- ➢ Final
- ➢ This Keyword

In the previous chapter, we covered Scala control statements, and in this chapter, we will discuss Object-Oriented Programming (OOP) concepts.

SCALA CLASS AND OBJECT

Classes and Objects are fundamental notions in OOP that center around real-world entities.

Class

A class is a user-defined blueprint or prototype used to build things. Alternatively, a class combines fields and methods (member functions that specify actions) into a single entity. In a class, the constructor is used to initialize new objects, fields are variables that supply the state of the class and its objects, and methods are used to implement the class's and its objects' actions.

Class Declaration

A class declaration in Scala begins with the class keyword and is followed by the class identifier (name). However, several optional properties can utilize with class definition depending on the application requirements. In general, class declarations can have the following components in the following order:

- Class keyword: The type class is declared using the class keyword.
- Classname: The classname should start with a capital letter (capitalized by convention).
- SuperClass (if any): If there is one, the name of the class's parent (superClass), preceded by the keyword extends. A class can only have one parent that it can extend (subClass).
- Traits (if any): A comma-separated list of the class's characteristics, if any, preceded by the term extends. A class can implement many traits.
- Body: The class's body is encircled by (curly braces).

Syntax:

```
class Classname{
    // the methods and fields
    }
```

It should note that the class's default modifier is public.

Example:

```
// program to illustrate how to create class

// Name of class is Smartphone
class Smartphone
{

    // the Class variables
    var number: Int = 16
    var nameofcompany: String = "Apple"

    // the Class method
    def Display()
    {
        println("The Name of the company : "
+ nameofcompany);
        println("The Total number of Smartphone
generation: " + number);
    }
}
object Main
{

    // the main method
    def main(args: Array[String])
    {

        // the Class object
        var obj = new Smartphone();
        obj.Display();
    }
}
```

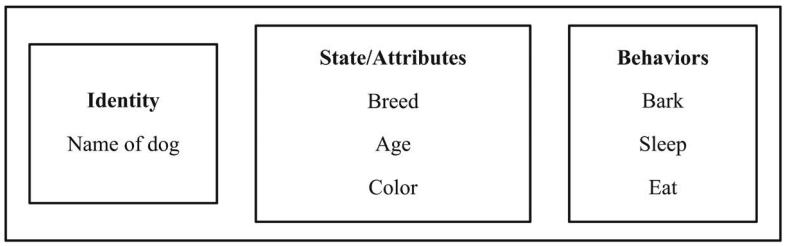
OBJECTS

It is a fundamental unit of OOP that represents real-world things. A typical Scala application generates a large number of objects, which interact via executing methods. An item is made up of:

- State: State is represented through an object's characteristics. It also reflects an object's attributes.

- Behavior: It is expressed through an object's methods. It also represents an object's interaction with other objects.
- Identity: It provides an object with a unique name and allows one thing to communicate with other objects.

Consider Dog as an object, and refer to the diagram below to learn about its identification, state, and behavior.

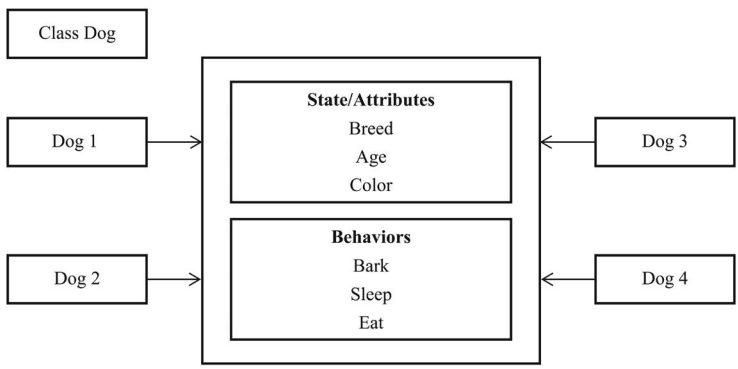


Objects in OOP.

Objects are items found in the actual world. A graphics application, for example, may have objects such as "circle," "square," and "menu." Objects in an online shopping system may include "shopping cart," "customer," and "product."

Defining Objects (Also Called Instantiating a Class)

When a class's object is generated, the class is said to be instantiated. All instances share the characteristics and behavior of the class. However, the values of those attributes, that is, the state, are specific to each object. A single class can have an unlimited number of instances.



Declaration of Objects.

The new keyword in Scala is used to construct a class object. In Scala, the syntax for constructing an object is:

Syntax:

```
var obj = new Dog();
```

Scala also has a companion objects feature that allows us to construct an object without using the new keyword.

Creating an Object

The new operator creates a class by allocating memory and returning a pointer to that memory. The new operator is also called the class constructor.

Example:

```
// program to illustrate Initialization of an
object

// Class with the primary constructor
class Dog(name:String, breed:String, age:Int,
color:String )
{
        println("My name:" + name + " my breed:" +
breed);
        println("I am: " + age + " and my color:"
+ color);

}
object Main
{

    // the main method
    def main(args: Array[String])
    {

        // the Class object
        var obj = new Dog("bruno", "papillon", 4,
"brown");
    }
}
```

Explanation: There is just one constructor in this class. We can identify a constructor because the body of a class is the body of the constructor in Scala, and the parameter-list precedes the class name. The Dog class's constructor accepts four parameters. The following sentence offers values for those parameters as "Bruno," "papillon," 4, and "brown":

```
var obj = new Dog("bruno", "papillon", 4, "brown");
```

Anonymous Object

Anonymous objects are objects that are created but lack a reference; we can construct an anonymous object if we do not wish to reuse it.

Example:

```
// program to illustrate how to create Anonymous
object

class PFP
{
    def display()
    {
        println("Welcome, Everyone");
    }
}
object Main
{

    // the main method
    def main(args: Array[String])
    {

        // Creating Anonymous object of the PFP
class
        new PFP().display();
    }
}
```

SCALA INNER CLASS

Inner class refers to the division of one class into another. This feature allows the user to logically organize classes that are only used in one location, increasing encapsulation and producing more understandable and

manageable code. The idea of inner classes in Scala differs from that of Java. The inner class, like the outer class in Java, is a member of the outer class, but in Scala, the inner class is connected to the outer object.

Syntax:

```
class Outerclass{
    class Innerclass{
    // Code…
    }
    }
```

Example:

```
// program to demonstrate how to create inner
class

// Outer class
class Peek
{

    // Inner-class
    class P1
    {
        var a = 0
        def method()
        {
            for(a<-0 to 4)
            {
                println("Welcome to the inner
class: P1");
            }
        }
    }
}
object Main
{
    def main(args: Array[String])
    {

        // Creating object of outer and
        // inner class Here, P1 class is bounded
with the object of Peek class
```

```
        val obj = new Peek();
        val o = new obj.P1;
        o.method();
    }
}
```

Explanation: Peek is the outside class in the preceding example, while P1 is the inner class. To construct the object of the inner class, we must first create the object of the outer class, which is obj. Because the inner class is tied to the object of the outer class, obj is prefixed with P1 class and creates the object o of P1 class.

How to Make a Class within an Object and an Object Inside a Class

We may also embed a class within an object or an object inside a class in Scala. Let's look at an example. In the next example, we first create an object within a class and then access the object's method using the new keyword followed by the class name, object name, and method name, as seen in the second comment:

```
new outerclass().innerobject.method;
```

Second, we construct a class within an object and access the methods contained within the class using the new keyword followed by the object, class, and method names, as illustrated in the following statement:

```
new outerobject.innerclass().method;
```

Example:

```
// program to illustrate how to create an object
inside class, Or
// class inside an object

// Class inside Object
class outerclass
{
    object innerobject
    {
        val q = 0;
        def method()
```

```
        {
            for(q <- 0 to 3)
            {
                println("The object inside a class
example")
            }
            println()
        }
    }
}

// Object inside Class
object outerobject
{
    class innerclass
    {
        val s = 0;
        def method()
        {
            for(s <- 0 to 2)
            {
                println("The class inside an
object example")
            }
        }
    }
}

object Main
{

    // the main method
    def main(args: Array[String])
    {

        // Object inside a class
        new outerclass().innerobject.method;

        // Class inside an object
        new outerobject.innerclass().method;
    }
}
```

SCALA INHERITANCE

OOP relies heavily on inheritance. The Scala mechanism allows one class to inherit another class's characteristics (fields and methods).

Important terms to remember:

- SuperClass: A superClass is a class with inherited characteristics (a base class or a parent class).
- SubClass: A subClass is a class that inherits from another class (or a derived class, extended class, or child class). In addition to the superClass's fields and methods, the subClass can add its own.
- Reusability: Inheritance supports the idea of "reusability," which means that if we want to construct a new class and there is already a class that has part of the code that we require, we may derive our new class from the old class. We are utilizing the old class's fields and functions by doing so.

HOW TO UTILIZE INHERITANCE IN SCALA

The keyword for inheritance is extends.

Syntax:

```
class child_classname extends parent_classname {
    // Methods and fields
    }
```

Example:

```
// program to illustrate implementation of
inheritance

// Base class
class Peeks1{
    var Name: String = "Drishti"
}

// Derived class Using the extends keyword
class Peeks2 extends Peeks1
{
    var Articleno: Int = 150

    // the Method
    def details()
```

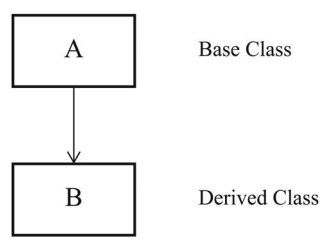
```
    {
    println("The Author name: " +Name);
    println("The Total numbers of articles: "
+Articleno);
    }
}

object Main
{

    // Driver code
    def main(args: Array[String])
    {

        // Creation of object of derived class
        val ob = new Peeks2();
        ob.details();
    }
}
```

Explanation: In the preceding example, Peeks1 is the base class, and Peeks2 is a derived class that is derived from Peeks1 through the extends keyword. When we create the object of the Peeks2 class in the main method, a duplicate of all the methods and fields of the base class is allocated memory in this object. As a result, we may access the base class members by using the object of the derived class.

INHERITANCE TYPE

Scala supports the various forms of inheritance listed below.

- **Single inheritance:** A derived class inherits the characteristics of just one base class under single inheritance. Class A acts as the base class for the derived class B in the illustration below.

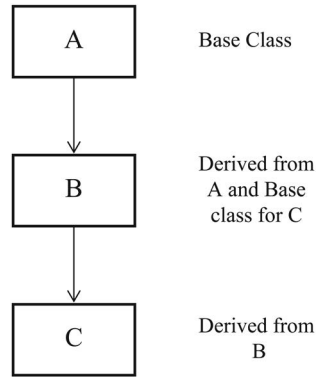


Single inheritance.

Example:

```
// program to illustrate Single inheritance

// Base class
class Parent
{
    var Name: String = "Drishti"
}

// Derived class
// Using the extends keyword
class Child extends Parent
{
    var Age: Int = 19

    // Method
    def details()
    {
    println("Name is: " +Name);
    println("Age is: " +Age);
    }
}

object Main
{

    // Driver code
    def main(args: Array[String])
    {

        // Creating object of derived class
        val ob = new Child();
        ob.details();
    }
}
```

- **Multilevel inheritance:** A derived class will inherit the base class, and the derived class will also function as the base class to some other class. The class A in the figure below consists of a base class for the derived class B, which acts as a base class for the derived class C.



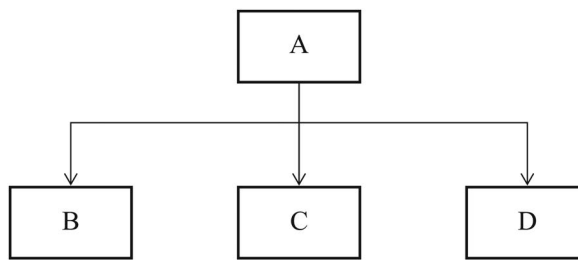
Multilevel inheritance.

Example:

```
// program to illustrate Multilevel inheritance

// Base class
class Parent
{
    var Name: String = "Shreya"
}

// Derived from the parent class
// Base class for the Child2 class
class Child1 extends Parent
{
    var Age: Int = 23
}

// Derived from the Child1 class
class Child2 extends Child1
{
    // Method
    def details(){
    println("Name is: " +Name);
    println("Age is: " +Age);
    }
}
```

```
object Main
{

    // Drived Code
    def main(args: Array[String])
    {

        // Creating object of derived class
        val ob = new Child2();
        ob.details();
    }
}
```

- **Hierarchical inheritance:** In this inheritance, one class acts as the superClass (base class) for several subClasses. In the diagram below, class A acts as the base class for the derived classes B, C, and D.



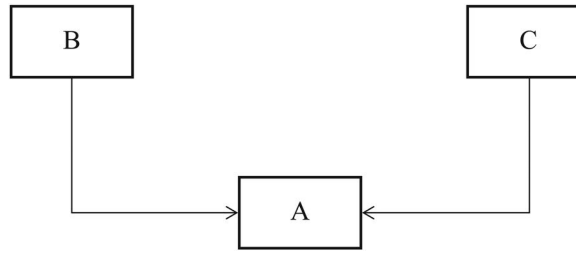
Hierarchical inheritance.

Example:

```
// program to illustrate Hierarchical
inheritance

// Base class
class Parent
{
    var Name1: String = "Shreya"
    var Name2: String = "Seema"
}

// Derived from parent class
```

```
class Child1 extends Parent
{
    var Age: Int = 23
    def details1()
    {
    println(" Name is: " +Name1);
    println(" Age is: " +Age);
    }
}

// Derived from the Parent class
class Child2 extends Parent
{
    var Height: Int = 154

    // Method
    def details2()
    {
    println(" Name is: " +Name2);
    println(" Height is: " +Height);
    }
}

object Main
{

    // Driver code
    def main(args: Array[String])
    {

        // Creation of objects of both derived
classes
        val ob1 = new Child1();
        val ob2 = new Child2();
        ob1.details1();
        ob2.details2();
    }
}
```

- **Multiple inheritance:** A class can have many superClasses and inherit traits from all parent classes. Multiple inheritance with classes is not supported in Scala, although it is possible using traits.



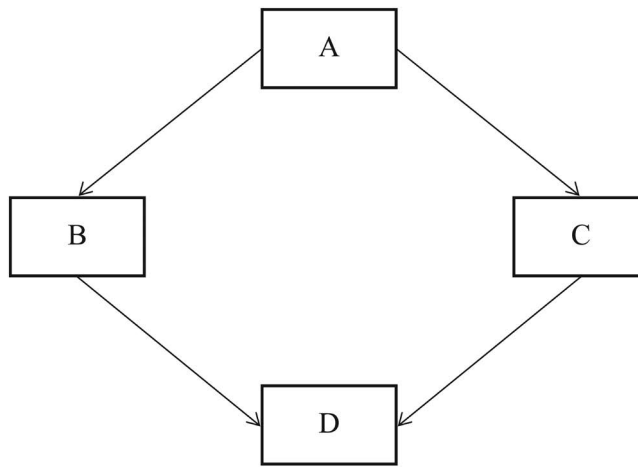
Multiple inheritance.

Example:

```
// program to illustrate multiple inheritance
using traits

// Trait 1
trait Peeks1
{
    def method1()
}

// Trait 2
trait Peeks2
{
    def method2()
}

// Class that implement both Peeks1 and Peeks2
traits
class PFP extends Peeks1 with Peeks2
{

    // method1 from Peeks1
    def method1()
    {
        println("Trait 1");
    }

    // method2 from Peeks2
    def method2()
    {
        println("Trait 2");
    }
}
```

```
object Main
{
    // Driver code
    def main(args: Array[String])
    {

        // Creating object of PFP class
        var obj = new PFP();
        obj.method1();
        obj.method2();
    }
}
```

- **Hybrid inheritance:** It combines two or more of the preceding inheritance categories. Because Scala does not enable multiple inheritance with classes, hybrid inheritance is likewise not feasible. Only characteristics in Scala allow us to accomplish hybrid inheritance.



Hybrid inheritance.

SCALA OPERATORS

An operator is a symbol that shows a method that must be carried out with one or more operands. Every programming language is built around operators. Operators allow us to do numerous actions on operands. Scala has several types of operators, which are as follows:

ARITHMETIC OPERATORS

These are used to carry out arithmetic and mathematical operations on operands.

- The addition (+) operator combines two operands. For instance, c+d.
- The subtraction (–) operator is used to subtract two operands. For instance, c–d.
- The multiplication (*) operator is used to multiply two operands. For instance, c*d.
- The division (/) operator divides the first and second operands. For instance, c/d.
- When first operand is divided by the second, the modulus(percent) operator returns the remainder. For instance, c % d.
- The exponent (**) operator returns the power of the operands. For instance, c**d.

Example:

```
// program to demonstrate Arithmetic Operators

object Arithop
{

def main(args: Array[String])
{
    // variables
    var c = 60;
    var d = 20;

    // Addition
    println("Addition of c + d = " + (c + d));

    // Subtraction
    println("Subtraction of c - d = " + (c - d));

    // Multiplication
    println("Multiplication of c * d = " + (c * d));
```

```
        // Division
        println("Division of c / d = " + (c / d));

        // Modulus
        println("Modulus of c % d = " + (c % d));

    }
    }
```

RELATIONAL OPERATORS

For comparing two values, relational operators or comparing operators are utilized. Let's go over them one by one:

- The Equal To(==) operator determines whether or not the two operands are equal. If this is case, it returns true. If not, it returns false. 5==5 will, for example, return true.
- The Not Equal To(!=) operator determines whether or not the two operands are equal. If it does not, it returns true. If not, it returns false. It is the boolean equivalent of the '==' operator. 5!=5 will, for example, return false.
- The Greater Than(>) operator determines if the first operand is greater than the second. If this is case, it returns true. If not, it returns false. 6>5 will, for example, yield true.
- The less than(<) operator determines if the first operand is less than the second. If this is case, it returns true. If not, it returns false. 6<5, for example, will return false.
- The Greater Than Equal To(>=) operator determines if the first and second operands are greater than or equal. If this is case, it returns true. If not, it returns false. 5>=5 will, for example, yield true.
- The operator Less Than Equal To(<=) determines if the first operand is less than or equal to the second operand. If this is case, it returns true. If not, it returns false. 5<=5 will, for instance, also yield true.

Example:

```
// program to demonstrate Relational Operators
object Relop
{
```

```
def main(args: Array[String])
{
    // variables
    var c = 60;
    var d = 20;

    // Equal to operator
    println("Equality of c == d is : " +
(c == d));

    // Not equal to operator
    println("Not Equals of c != d is : " +
(c != d));

    // Greater than operator
    println("Greater than of c > d is : " +
(c > d));

    // Lesser than operator
    println("Lesser than of c < d is : " +
(c < d));

    // Greater than equal to operator
    println("Greater than or Equal to of c >= d is
: " + (c >= d));

    // Lesser than equal to operator
    println("Lesser than or Equal to of c <= d is
: " + (c <= d));

}
}
```

LOGICAL OPERATORS

They are used to integrate two or more conditions/constraints or to supplement the evaluation of the original condition. These are as follows:

- When both of the conditions in question are met, the logical AND(&&) operator returns true. If not, it returns false. The "and" operator is an alternative to the && operator. c && d, for example, returns true when both c and d are true (i.e., non-zero).

- When one (or both) of the criteria in question are met, the logical OR(||) operator returns true. If not, it returns false. The "or" operator is an alternative to the || operator. For instance, c || d returns true if either c or d is true (i.e., non-zero). Naturally, it returns true if both c and d are true.
- Logical When the NOT(!) operator returns true, the condition in question is not met. If not, it returns false. The "not" operator is an alternative to the! operator. !true, for example, returns false.

Example:

```
// program to demonstrate
// Logical Operators
object Logop
{

def main(args: Array[String])
{

    // variables
    var c = false
    var d = true

    // logical NOT operator
    println("Logical Not of !(c && d) = " + !(c &&
d));

    // logical OR operator
    println("Logical Or of c || d = " + (c || d));

    // logical AND operator
    println("Logical And of c && d = " + (c &&
d));

}
}
```

ASSIGNMENT OPERATORS

It is used to assign the value to a variable. The assignment operator's left operand is a variable, while the assignment operator's right operand is a

value. However, the compiler will throw an error if the value on the right side is not of the same data-type as the variable on the left side.

The following are examples of assignment operators:

- The simple assignment operator is the equals symbol (=). This operator is used to assign the variable on the left the value on the right.
- The Add AND Assignment (+=) operator is used to combine the left and right operands and then assign the result to a variable on the left.
- The Subtract AND Assignment (−=) operator is used to subtract the left operand from the right operand and then assign the result to the left operand's variable.
- The Multiply AND Assignment (*=) operator is used to multiply the left operand by the right operand and then assign the result to the left operand variable.
- The Divide AND Assignment (/=) operator divides the left operand by the right operand and then assigns the result to a variable on the left.
- The Modulus AND Assignment (% =) operator is used to assign the modulus of the left operand to the right operand and then to the variable on the left.
- The Exponent AND Assignment (**=) operator is used to increase the power of the left operand and assign it to the variable on the left.
- The Left shift AND Assignment (<<=) operator performs a binary left shift of the left operand with the right operand and assigns it to the variable on the left.
- The Right shift AND Assignment (>>=) operator performs binary right shift of the left operand with the right operand and assigns it to the variable on the left.
- The Bitwise AND Assignment (&=) operator is used to execute Bitwise And on the left operand and assign it to the variable on the left.
- The Bitwise exclusive OR and Assignment (^=) operator is used to conduct Bitwise exclusive OR on the left operand and assign it to the variable on the left.

- The Bitwise inclusive OR and Assignment (|=) operator is used to conduct Bitwise inclusive OR on the left operand and assign it to the variable on the left.

Example:

```
// program to demonstrate
// Assignments Operators
object Assignop
{

def main(args: Array[String])
{

    // variables
    var x = 50;
    var y = 40;
    var z = 0;

    // simple addition
    z = x + y;
    println("simple addition: z= x + y = " + z);

    // Add AND assignment
    z += x;
    println("Add and assignment of z += x = " +
z);

    // Subtract AND assignment
    z -= x;
    println("Subtract and assignment of z -= x = "
+ z);

    // Multiply AND assignment
    z *= x;
    println("Multiplication and assignment of z *=
x = " + z);

    // Divide AND assignment
    z /= x;
    println("Division and assignment of z /= x =
" + z);
```

```
    // Modulus AND assignment
    z %= x;
    println("Modulus and assignment of z %= x = "
+ z);

    // Left shift AND assignment
    z <<= 3;
    println("Left shift and assignment of z <<=
3 = " + z);

    // Right shift AND assignment
    z >>= 3;
    println("Right shift and assignment of z >>=
3 = " + z);

    // Bitwise AND assignment
    z &= x;
    println("Bitwise And assignment of z &= 3 =
" + z);

    // Bitwise exclusive OR and assignment
    z ^= x;
    println("Bitwise Xor and assignment of z ^=
x = " + z);

    // Bitwise inclusive OR and assignment
    z |= x;
    println("Bitwise Or and assignment of z |=
x = " + z);
}
}
```

BITWISE OPERATORS

Scala has seven bitwise operators that act at the bit level or are used to execute bit by bit operations. The bitwise operators are as follows:

- Bitwise AND (&): Takes two operands and does AND on each bit of the two numbers. The AND only returns 1 if both bits are 1.
- Bitwise OR (|): Takes two operands and performs OR on each bit of the two integers. OR yields 1 if any of the two bits is 1.

- Bitwise XOR (^): Takes two operands and performs XOR on each bit of the two numbers. If two bits are different, the result of XOR is 1.
- Bitwise left Shift (<<): Takes two integers, left shifts the bits of the first operand, and the second operand determines the number of places to shift.
- Bitwise right shift (>>): Takes two values, right shifts the bits of the first operand, and the second operand determines the number of places to shift.
- Bitwise operations complement (^): This operator accepts a single number and performs an 8-bit complement operation.
- Bitwise shift right zero fill (>>>): The left operand is shifted right by the number of bits indicated by the right operand, and the shifted values are filled with zeros.

Example:

```
// program to demonstrate
// Bitwise Operators
object Bitop
{
def main(args: Array[String])
{
    // variables
    var x = 20;
    var y = 18;
    var z = 0;

    // Bitwise AND operator
    z = x & y;
    println("Bitwise And of x & y = " + z);

    // Bitwise OR operator
    z = x | y;
    println("Bitwise Or of x | y = " + z);

    // Bitwise XOR operator
    z = x ^ y;
    println("Bitwise Xor of x ^ y = " + z);
```

```
        // Bitwise once complement operator
        z = ~x;
        println("Bitwise Ones Complement of ~x = " + z);

        // Bitwise left shift operator
        z = x << 3;
        println("Bitwise Left Shift of x << 3 = " + z);

        // Bitwise right shift operator
        z = x >> 3;
        println("Bitwise Right Shift of x >> 3 = " + z);

        // Bitwise shift right zero fill operator
        z = x >>> 4;
        println("Bitwise Shift Right x >>> 4 = " + z);
    }
    }
```

SCALA OPERATORS PRECEDENCE

An operator is a way to depict a process that must be performed with one or more operands. Operators are the foundations of every programming language. Operator precedence determines which operator is performed first in an expression having several operators with differing precedence.

For example, 20 + 10 * 30 is computed as 20 + (10 * 30) rather than (20 + 10) * 30.

Associativity is utilized when two operators with the same precedence exist in an expression. Right-to-left or left-to-right associativity is possible. Because "*" and "/" have the same precedence and associativity is left to right, the phrase "100 / 20 * 20" is handled as "(100 / 10) * 20."

The table below shows operators with the highest precedence at the top and operators with the lowest precedence at the bottom.

Operator	Category	Associativity
()[]	Postfix	Left to Right
* / %	Multiplicative	Left to Right
! ~	Unary	Right to Left
+ –	Additive	Left to Right
< <= > >=	Relational	Left to Right
>> >>> <<	Shift	Left to Right

(Continued)

Operator	Category	Associativity
== !=	The Relational is equal to/ is not equal to	Left to Right
== !=	Equality	Left to Right
&	Bitwise AND	Left to Right
|	Bitwise inclusive OR	Left to Right
^	Bitwise exclusive OR	Left to Right
&&	Logical AND	Left to Right
||	Logical OR	Left to Right
= += -= *= /= %= >>= <<= &= ^= |=	Assignment	Right to left
,	Comma (separate expressions)	Left to Right

Example:

```
// program to show Operators Precedence
// Creation of object
object pfp
{
    // the main method
    def main(args: Array[String])
    {

        var a:Int = 40;
        var b:Int = 20;
        var c:Int = 35;
        var d:Int = 9;
        var e = 0

        // operators with highest precedence
        // will operate first
        e = a + b * c / d;

        println("The Value of a + b * c / d is : "
+ e )

    }
}
```

SCALA ABSTRACT CLASSES

Abstraction is the process of hiding internal information and displaying simply the functionality. An abstract class is used to accomplish

abstraction in Scala. The Scala abstract class operates similarly to the Java abstract class. The abstract keyword is used in Scala to create an abstract class. It has both abstract and non-abstract methods and cannot handle multiple inheritances. A class can extend just one abstract class.

Syntax:

```
abstract class classname
{
// code…
}
```

The abstract methods of an abstract class are those that do not have any implementation. In other terms, an abstract procedure is one that does not have anybody.

Syntax:

```
def functionname()
```

Example:

```
// program to illustrate how to create abstract
class

// Abstract class
abstract class myauthor
{

    // the abstract method
    def details()
}

// PFP class extends abstract class
class PFP extends myauthor
{
    def details()
    {
        println("Author name is: Shreya Sood")
        println("Topic name is: Abstract class in
the Scala")
    }
}
```

```
object Main
{
    // the main method
    def main(args: Array[String])
    {
        // objects of PFP class
        var obj = new PFP()
        obj.details()
    }
}
```

The following are some key points to remember regarding abstract classes in Scala.

- In Scala, like in Java, we are not permitted to make an instance of the abstract class. When we try to construct objects of the abstract class, the compiler throws an error, as demonstrated in the following program.

  Example:

```
// program to illustrate concept of abstract
class

// Abstract class
abstract class myauthor{

    // abstract method
    def details()
}

object Main {

    // the main method
    def main(args: Array[String]) {

        // the object of myauthor class
    var obj = new myauthor()
    }
}
```

- An abstract class in Scala can also include fields. These fields are accessible by abstract class methods as well as class methods that inherit abstract class. As shown in the program following.

Example:

```
// program to illustrate
// concept of abstract class

// Abstract class with the fields
abstract class Peek
{
    var name : String = "HelloEveryone"
    var tutorial: String = "Scala"
    def portal()
}

// PFP class extends abstract class
class PFP extends Peek
{

    // Abstract class method accessing
    // fields of abstract class
    def portal()
    {
        println("The Portal name: " + name)

    }

    // PFP class method accessing
    // fields of abstract class
    def tutdetails()
    {
        println("The Tutorial name: " +
tutorial)
    }
}

object Main
{

    // the main method
    def main(args: Array[String])
    {

        // objects of PFP class
        var obj = new PFP()
```

```
        obj.portal()
        obj.tutdetails()
    }
}
```

- An abstract class in Scala, like Java, can have a constructor, and an abstract class's constructor is invoked when an instance of an inherited class is created. As shown in the program below.

Example:

```
// program to illustrate
// concept of abstract class

// Abstract class with the constructor
// And constructor contain two arguments
abstract class myauthor(name: String,
                        topic: String)
{
    def details()
}

// PFP class extends abstract class
class PFP(name: String, topic: String) extends
                          myauthor(name, topic)
{
    def details()
    {
        println("Author name is: " + name)
        println("Topic name is: " + topic)
    }
}

object Main
{

    // the main method
    def main(args: Array[String])
    {

        // objects of PFP class
```

```
        var obj = new PFP("Shreya", "Abstract
class")
        obj.details()
    }
}
```

- Only non-abstract methods can be found in an abstract class. This enables us to construct classes that can only be inherited rather than instantiated. As shown in the program below.

Example:

```
// program to illustrate
// concept of abstract class

// Abstract class with
// the non-abstract method
abstract class myauthor
{

    // Non-abstract method
    def details()
    {
        println("Welcome to Club")
    }
}

// PFP class extends abstract class
class PFP extends myauthor{}

object Main
{

    // the main method
    def main(args: Array[String])
    {

        // objects of PFP class
        var obj = new PFP()
        obj.details()
    }
}
```

- An abstract class in Scala can have final methods (methods that cannot be overridden). The following program, for instance, builds and executes without problem. The final keyword is used in Scala to create the final method.

  Example:

```
// program to illustrate
// concept of abstract class

// Abstract class with final method
abstract class myauthor
{
    final def mymethod()
    {
        println("The Final method")
    }
}

// PFP class extends abstract class
class PFP extends myauthor{}

object Main
{

    // the main method
    def main(args: Array[String])
    {

        // objects of PFP class
        var obj = new PFP()
        obj.mymethod()
    }
}
```

When Should We Use Abstract Class in Scala?

A useful abstract class is:

- When we want to create a base class that requires constructor parameters.
- When our code is invoked from Java code.
- It should be noted that traits may also be utilized to accomplish abstraction.

SCALA COMPANION OBJECTS AND SINGLETON

SINGLETON OBJECT

Scala is a more object-oriented language than Java, hence there is no idea of a static keyword in Scala. Scala uses a singleton object instead of the static keyword. A Singleton object is one that defines just one object of a class. A singleton object serves as the starting point for our program's execution. If we do not construct a singleton object in our application, our code will compile but will not produce any output. So we needed a singleton object to retrieve our program's output. The object keyword is used to construct a singleton object.

Syntax:

```
object Name{
    // code…
    }
```

Important information regarding the singleton object:

- The method in the singleton object is available globally.
- We are not permitted to construct a singleton object instance.
- The primary constructor of a singleton object cannot accept parameters.
- A singleton object in Scala can extend classes and traits.
- A main method is always present in a singleton object in Scala.
- The method in the singleton object is called using the object's name (much like calling a static method in Java), hence no object is required to access this function.

First example:

```
// program to illustrate
// concept of the singleton object

class AreaOfRectangle
```

```
{

    // Variables
    var length = 50;
    var height = 30;

    // the method which gives area of the
rectangle
    def area()
    {
        var ar = length * height;
        println("Height of the rectangle:" +
height);
        println("Length of the rectangle:" +
length);
        println("Area of the rectangle:" + ar);
    }
}

// the singleton object
object Main
{
    def main(args: Array[String])
    {

        // Creating object of the AreaOfRectangle
class
        var obj = new AreaOfRectangle();
        obj.area();
    }
}
```

Second example:

```
// program to illustrate
// how to call method inside the singleton object

// Singleton object with the named as
Exampleofsingleton
object Exampleofsingleton
{
```

```
    // Variables of singleton object
    var str1 = "Welcome ! toSession";
    var str2 = "This is Scala programming
tutorial";

    // Method of the singleton object
    def display()
    {
        println(str1);
        println(str2);
    }
}

// Singleton object with the named as Main
object Main
{
    def main(args: Array[String])
    {

        // Calling method of the singleton object
        Exampleofsingleton.display();
    }
}
```

Explanation: In the above instance, we have two singletons, Exampleofsingleton and Main. The function display() in the exampleofsingleton object is now called in the Main object. Using the expression Exampleofsingleton.display(), we invoke the display() function of the Exampleofsingleton object and print the output.

COMPANION OBJECT

A companion object is one whose name is the same as the class name. Or in other words, when an object and a class share the same name, the object is referred to as the companion object, and the class is referred to as the companion class. The companion object is specified in the same source file as the class. A companion object has access to both the class's private methods and private fields.

Example:

```
// program to illustrate
// concept of the Companion object
```

```
// the Companion class
class ExampleofCompanion
{

    // Variables of the Companion class
    var str1 = "WelcomeToSession";
    var str2 = "Tutorial of the Companion object";

    // Method of the Companion class
    def show()
    {
        println(str1);
        println(str2);
    }
}

// the Companion object
object ExampleofCompanion
{
    def main(args: Array[String])
    {
        var obj = new ExampleofCompanion();
        obj.show();
    }
}
```

SCALA GENERIC CLASSES

Creating a Generic Class in Scala is quite similar to creating generic classes in Java. In Scala, Generic Classes are defined as classes that accept a type as a parameter. This class accepts a type as a parameter enclosed by square brackets, that is, [ ]. In Scala, these classes are expressly used to advance the collection classes. If we have two list types, A and B, then List[A] is a sub-type of List[B] if and only if type B is comparable to type A.

Some things to keep in mind:

- A, as in List[A], is the symbol for a type parameter of a simple type.
- In generic classes, the symbols for a type parameter of the second, third, fourth, and so on, types are B, C, D, and so on.
- In Scala Map, the symbol for a key is A, while the symbol for a value is B.
- A numeric value is represented by the symbol N.

Despite the fact that certain symbol conventions are specified, any symbol can be used for the type parameters.

Let's look at a few instances below.

Example:

```
// program of forming Generic classes

// Creation of an object
object PfP
{

    // the main method
    def main(args: Array[String])
    {

        // Class structure for the Generic types
        abstract class Divide[z]
        {
            // Defining method
            def divide(u: z, v: z): z
        }

        // Extending Generic class of the type
parameter Int
        class intDivide extends Divide[Int]
        {
            // method returning Int
            def divide(u: Int, v: Int): Int =
u / v
        }

        // Extending Generic Class of the type
parameter Double
        class doubleDivide extends Divide[Double]
        {
            // method returning Double
            def divide(u : Double, v : Double) :
Double = u / v
        }

        // Creation of objects and assigning
        // values to methods called
```

```
        val q = new intDivide().divide(35, 8)
        val r = new doubleDivide().divide(32.0, 6.0)

        // Display the output
        println(q)
        println(r)

    }
}
```

The abstract class Divide contains a type argument z, which is enclosed in square brackets to indicate that the class is generic; this type parameter z can accept any data type. We have built a method split within the Generic class, which contains two variables, u and v, with the data type z. As previously indicated, the class intDivide accepts integer values while the class doubleDivide accepts double kinds. As a result, the type parameter z was replaced with the data types Int and Double. Subtyping is feasible in Generic classes in this manner.

Example:

```
// program of using generic
// types for the numeric values
import Numeric._

// Creation of an object
object PfP
{

    // the main method
    def main(args: Array[String])
    {

        // Defining generic type for the numeric
        // values with the implicit parameter
        def addition[N](a: N, b: N)(implicit num:
Numeric[N]):

        // Using a method 'plus'
        N = num.plus(a, b)

        // Displays sum of two numbers
```

```
        println("The sum: "+addition(55, 10))
    }
}
```

SCALA ACCESS MODIFIERS

Scala Access Modifiers are used to specify the access field of members of packages, classes, or objects. To use an access modifier, add its keyword in the definition of package, class, or object members. These modifications will limit the members' access to certain portions of code.

In Scala, there seem to be three types of access modifiers:

- Private
- Protected
- Public

The following table lists the types of access modifiers:

Modifier	Class	Companion	SubClass	Package	World
No Modifier/Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No *	No
Private	Yes	Yes	No	No *	No

What is the meaning of companion in the above table?
It is a singleton object with the same name as the class.

1. **Private:** A private member may only be used within the defining class or through one of its objects.

   Example:

```
// program of private access modifier
class xyz
{
    private var m:Int = 345
    def display()
    {
        m = 8
        println(m)
    }
}
```

```
object access extends App
{
    // class xyz is accessible
    // because this is in same enclosing scope
    var n = new xyzc()
    n.display()
}
```

We defined a variable "a" private here, and it may now be accessible only within its defining class or through the class's object.

2. **Protected:** They can only be accessed by subClasses of the base class where the member is declared.

Example:

```
// program of the protected access modifier

class pfp
{
    // declaration of the protected member
    protected var a:Int = 234
    def display()
    {
        a = 9
        println(a)
    }
}

// class new1 extends by class pfp
class new1 extends pfp
{
    def display1()
    {
        a = 10
        println(a)
    }
}

object access extends App
{
    // class abc is accessible because this
    // is in same enclosing scope
    var e = new gpfp()
    e.display()
```

```
    var e1 = new new1()
    e1.display1()

}
```

Because new1 is a subClass of abc, we were allowed to modify protected variable a when we extended abc in class new1.

3. **Public:** Scala does not have a public keyword. When no modifier is supplied, the default access level corresponds to Java's public access level.S

Example:

```
// program of the protected access modifier

class pfp
{
    var a:Int = 234
}
object access extends App
{
    var e = new gfg()
    e.a = 666
    println(e.a)
}
```

SCALA CONSTRUCTORS

Constructors are used to set the state of an object. A constructor, like methods, includes a set of statements (i.e., instructions) performed when an object is created.

Scala has two kinds of constructors:

PRIMARY CONSTRUCTOR

When our Scala code just has one constructor, that constructor is referred to as the main constructor. Because the primary constructor and the class share the same body, we don't need to define a constructor explicitly.

Syntax:

```
class classname(Parameterlist){
    // Statements....
    }
```

Important details:

- Because the primary constructor and the class share the same body in the above syntax, anything declared in the body of the class except the method declaration is part of the primary constructor.

  Example:

```
// program to illustrate concept of primary
constructor

// Creation of primary constructor
// with parameterlist
class GFG(Aname: String, Cname: String,
Particle: Int)
{
    def display()
    {
        println("Author name is: " + Aname);
        println("Chapter name is: " + Cname);
        println("Total published articles is:" +
Particle);
    }
}

object Main
{
    def main(args: Array[String])
    {

        // Creating and initializing
        // object of PFP class
        var obj = new PFP("Shreya",
"Constructors", 240);
        obj.display();
    }
}
```

- There may be zero or more arguments in the main constructor.
- If we do not define a constructor in our Scala code, the compiler will generate one when we create an object of our class; this constructor is known as a default primary constructor. There are no parameters in it.

Example:

```
// program to illustrate
// the concept of default primary constructor

class PFP
{
    def display()
    {
        println("Welcome to Sessions");
    }
}

object Main
{
    def main(args: Array[String])
    {

        // Creating object of PFP class
        var obj = new PFP();
        obj.display();
    }
}
```

- If the constructor parameter-list parameters are defined with var, the values of the fields may vary. Scala creates getter and setter methods for that field as well.
- If the constructor parameter-list arguments are defined with val, the values of the fields cannot change. In addition, Scala produces a getter function for that field.
- If the arguments in the constructor parameter-list are not defined with val or var, the field's accessibility is severely limited. Scala generates no getter or setter methods for the field.
- If the arguments in the constructor parameter-list are specified using private val or var, no getter and setter methods are generated for that field. As a consequence, members of that class have access to these fields.
- Only a primary constructor in Scala is permitted to call a superClass constructor.

- We may make a primary constructor private in Scala by inserting a private keyword between the class name and the constructor parameter-list.

  Syntax:

```
// private constructor with the two argument
class PFP private(name: String, class:Int){
    // code..
    }

    // private constructor without the argument
    class PFP private{
    // code...
    }
```

- In Scala, we may specify default values in the constructor declaration.

  Example:

```
// program to illustrate concept of primary
constructor

// Creation of primary constructor with the
default values
class PFP(val Aname: String = "Shreya",
        val Cname: String = "Constructors")
{
    def display()
    {
        println("Author name is: " + Aname);
        println("Chapter name is: " + Cname);

    }
}

object Main
{
    def main(args: Array[String])
    {
        // Creating object of PFP class
        var obj = new PFP();
        obj.display();
    }
}
```

AUXILIARY CONSTRUCTOR

Auxiliary constructors are constructors other than the primary constructor in a Scala program. We can include as many auxiliary constructors as we like in our code, but there is only one major constructor.

Syntax:

```
def this(.......)
```

Important details:

- We can have numerous auxiliary constructors in a piece of code, but they must have separate signatures or parameter lists.
- Every auxiliary constructor must use one of the constructors already specified.
- The invoke constructor can be a primary or an auxiliary constructor that occurs before the calling constructor in the code.
- The constructor call utilizing this must include the auxiliary constructor's first sentence.

Example:

```
// program to illustrate
// the concept of the Auxiliary Constructor

// Primary constructor
class PFP( Aname: String, Cname: String)
{
    var no: Int = 0;;
    def display()
    {
        println("Author name is: " + Aname);
        println("Chapter name is: " + Cname);
        println("Total number of articles is: " +
no);

    }

    // the Auxiliary Constructor
    def this(Aname: String, Cname: String, no:Int)
```

```
    {

        // Invoking the primary constructor
        this(Aname, Cname)
        this.no=no
    }
}

object Main
{
    def main(args: Array[String])
    {

        // Creation of object of PFP class
        var obj = new PFP("Ankita", "Constructor",
37);
        obj.display();
    }
}
```

PRIMARY CONSTRUCTOR IN SCALA

Constructors are used to set the state of an object. A constructor, like methods, includes a set of statements (i.e., instructions). Statements are performed when an object is created. When our Scala program has only one constructor, that constructor is referred to be the primary constructor.

Because the primary constructor and the class share the same body, we don't need to define a constructor explicitly.

Syntax:

```
class classname(Parameterlist) {
    // Statements...
    }
```

Some key points concerning Scala's primary constructor are as follows:

- There can zero or more arguments for the main constructor.
- The parameters of the parameter-list are specified using var within the constructor, and their values are subject to change. Scala creates getter and setter methods for that field as well.

- When the parameters of the parameter-list are specified using val within the constructor, the value cannot change. In addition, Scala only produces a getter function for that field.
- When the parameters in a parameter-list are declared in the constructor without using val or var, the visibility of the field is very compact, and Scala does not produce any getter and setter methods for that field.
- When the parameters in a parameter-list are specified using private val or var in the constructor, no getter and setter methods are generated for that field. As a consequence, members of that class have access to these fields.
- Only a primary constructor in Scala is permitted to call a superClass constructor.

Let's break it down with some examples.

First example: A primary constructor with a set of parameters.

```
// program to illustrate concept of primary
constructor

// Creation of a primary constructor
// with the parameter-list
class PFP(Lname: String, Tname: String, article:
Int)
{
    def show()
    {
        println("Language name is: " + Lname);
        println("Topic name is: " + Tname);
        println("Total published articles is:" +
article);
    }
}

// Creation of object
object Main
{
    // the main method
    def main(args: Array[String])
```

```
    {

        // Creating and initializing
        // object of PFP class
        var obj = new PFP("Scala", "Constructors",
16);
        obj.show();
    }
}
```

In the preceding example, Lname, Tname, and article are major constructor parameters, and show the function that prints values.

Second example: A parameter-list primary constructor.

```
// program to illustrate concept of the default
primary constructor

class PFP
{
    def show()
    {
        println("Welcome to Sessions");
    }
}

// Creation of object
object Main
{
    // the main method
    def main(args: Array[String])
    {

        // Creation of object of PFP class
        var obj = new PFP();
        obj.show();
    }
}
```

When we generate an object of our class, the compiler will automatically produce a primary constructor, which is known as a default primary constructor.

Third example: Default values for the primary constructor.

```
// program to illustrate concept of primary
constructor

// Creating primary constructor with the default
values
class PFP(val Lname: String = "Scala",
        val Tname: String = "Constructors")
{
    def show()
    {
        println("Language name is: " + Lname);
        println("Topic name is: " + Tname);

    }
}

// Creating object
object Main
{
    // the main method
    def main(args: Array[String])
    {
        // Creating object of PFP class
        var obj = new PFP();
        obj.show();
    }
}
```

Fourth example: Using the private keyword to make the primary constructor private.

```
// program to illustrate concept of the primary
constructor
// by using private keyword
class PFP private
{
    // Define the method
    override def toString = "Welcome to Sessions."
}

// Creating object of the class PFP
```

```
object PFP
{
    // Creation of object
    val pfp = new PFP
    def getObject = pfp
}

object SingletonTest extends App
{

// this won't compile
// val pfp = new PFP
// this works
val pfp = PFP.getObject
println(pfp)
}
```

AUXILIARY CONSTRUCTOR IN SCALA

Constructors are used to set the state of an object. A constructor, like methods, includes a set of statements (i.e., instructions). Statements are performed when an object is created. Auxiliary Constructors are constructors that are not the primary constructor in a Scala Program. In our Scala class, we may add as many auxiliary constructors as we like, but there is only one primary constructor.

Auxiliary constructors are declared in the class as methods using the keyword this. Multiple auxiliary constructors can be described, but their argument lists must be distinct.

Syntax:

```
def this(.......)
```

Let's look at few instances to assist us grasp Auxiliary constructors.

First example: Only one Auxiliary Constructor is used.

```
// program to illustrate concept of Auxiliary
Constructor

// Primary constructor
class PFP( Lname: String, Tname: String)
```

```
{
    var no: Int = 0;;
    def show()
    {
        println("Language name is: " + Lname);
        println("Topic name is: " + Tname);
        println("Total number of articles is: " +
no);

    }

    // the Auxiliary Constructor
    def this(Lname: String, Tname: String, no:Int)
    {

        // Invoking the primary constructor
        this(Lname, Tname)
        this.no = no
    }
}

// Creation of object
object Main
{
    // the main method
    def main(args: Array[String])
    {

        // Creating object of PFP class
        var obj = new PFP("Scala", "Constructor",
5);
        obj.show();
    }
}
```

As we see in the above instance, only one auxiliary constructor is utilized, and the primary constructor is executed in that auxiliary constructor. After creating an object of the PFP class (obj), the display() function will be invoked, and the result will print.

Second example: Using several Auxiliary Constructors.

```
// Scala program to illustrate concept of more than
concept
```

```
// Auxiliary Constructor

// the Primary constructor
class Company
{
    private var Cname = ""
    private var Employee = 0

    // Creation of function
    def show()
    {
        println("Language name is: " + Cname);
        println("Total number of employee is: " +
Employee);
    }

    // auxiliary constructor
    def this(Cname: String)
    {
        // Calls the primary constructor
        this()
        this.Cname = Cname
    }

    // Another auxiliary constructor
    def this(Cname: String, Employee: Int)
    {
        // Calls the previous auxiliary constructor
        this(Cname)
        this.Employee = Employee
    }
}

// Creation of object
object Main
{
    // the main method
    def main(args: Array[String])
    {
        // the Primary constructor
        val c1 = new Company
        c1.show()
```

```
        // First auxiliary constructor
        val c2 = new Company("PeeksForPeeks")
        c2.show()

        // Second auxiliary constructor
        val c3 = new Company("PeeksForPeeks", 42)
        c3.show()

    }
}
```

As we see from the code above, two auxiliary constructors with distinct arguments are produced. Auxiliary constructor is called primary constructor, and another auxiliary constructor is called auxiliary constructor that was previously declared.

Some vital points about Auxiliary Constructor are as follows:

- In the same class, we can have one or more auxiliary constructors, but they must have independent signatures or parameter-lists.
- Each auxiliary constructor must invoke one of the previously defined constructors, either the primary or preceding auxiliary constructor.
- The invoke constructor might be a primary or previous auxiliary constructor that appears before the calling constructor in the code.
- The this keyword must be in the first statement of the auxiliary constructor.

IN SCALA, CALLING A SUPERCLASS CONSTRUCTOR

Constructors are used to create an object's state in Scala and are run when the object is created. There is just one primary constructor, and all other constructors must eventually chain into it. When we specify the extends component of a subClass declaration in Scala, we control the superClass constructor that its primary constructor invokes.

With one constructor: A call to a superClass constructor instance

Example:

```
// program to show calling a super class
constructor
```

```
// Primary constructor
class PFP (var message: String)
{
    println(message)
}

// Calling super class constructor
class Subclass (message: String) extends PFP
(message)
{
    def display()
    {
        println("Subclass constructor called")
    }
}

// Creation of object
object Main
{
    // the main method
    def main(args: Array[String])
    {

        // Creation of object of Subclass
        var obj = new Subclass("Peeksforpeeks");
        obj.display();
    }
}
```

- In the preceding example, the subClass is specified to call the PFP class's primary constructor, which is a single argument constructor that accepts message as a parameter. When constructing a subClass in Scala, one has control over the SuperClass constructor, which is invoked by the SubClass's primary constructor when the extends section of the SubClass declaration is defined.
- **With numerous builders:** If the superClass has many constructors, any of those constructors can be invoked using the SubClass's primary constructor. For instance, in the following code, the superClass's double argument constructor is called by the SubClass's primary constructor via the extends clause by declaring the particular constructor.

Example:

```
// Scala program to show
// calling a specific super class constructor

// Primary constructor (1)
class PFP (var message: String, var num: Int)
{

    println(message+num)

    // Auxiliary constructor (2)
    def this (message: String)
    {
        this(message, 0)

    }

}

// Calling super class constructor with 2
arguments
class Subclass (message: String) extends PFP
(message, 3000)
{
    def display()
    {
        println("Subclass constructor called")
    }
}

// Creating object
object PFP
{
    // Main method
    def main(args: Array[String])
    {

        // Creating object of Subclass
        var obj = new Subclass("Article count ");
        obj.display();
    }
}
```

We may call the single argument constructor here, and the default value for the additional parameter is 0.

Example:

```
// program to illustrate
// calling the specific super class constructor

// the Primary constructor (1)
class PFP(var message: String, var num: Int)
{

    println(message + num)

    // the Auxiliary constructor (2)
    def this (message: String)
    {
        this(message, 0)

    }

}

// Calling superclass constructor with the 1
arguments
class Subclass (message: String) extends PFP
(message)
{
    def display()
    {
        println("Subclass constructor called")
    }
}

// Creation of object
object PFP
{
    // the main method
    def main(args: Array[String])
    {

        // Creation of object of the Subclass
```

```
        var obj = new Subclass("Article Count");
        obj.display();
    }
}
```

SCALA CLASS EXTENDING

When extending a class in Scala, the user can create an inherited class. In Scala, we utilize the extends keyword to extend a class. In Scala, there are two limitations to extending a class:

To override a method in Scala, use the override keyword.

The primary constructor is the only one that may provide parameters to the base constructor.

Syntax:

```
class base_classname extends derived_classname
{
    // Methods and fields
}
```

Example:

```
// program of extending a class

// Base class
class Peeks1
{
    var Name: String = "drishtisood"
}

// Derived class
// Using the extends keyword
class peeks2 extends Peeks1
{
    var Article_no: Int = 40

    // Method
    def details()
    {
        println("Author name is: " + Name);
        println("Total numbers of articles is: "
+ Article_no);
```

```
    }
}

// Creation of object
object PFP
{

    // Driver code
    def main(args: Array[String])
    {

        // Creation of object of the derived class
        val ob = new Peeks2();
        ob.details();
    }
}
```

Peeks1 is the base class in the above example, and Peeks2 is the derived class that is derived from Peeks1 through the extends keyword. When we create the object of the Peeks2 class in the main method, a duplicate of all the methods and fields of the base class is allocated memory in this object. As a result, we may access the base class members by using the object of the derived class.

Example:

```
// program of extending a class

// Base class
class Parent
{
    var Name1: String = "peek1"
    var Name2: String = "peek2"
}

// Derived from parent class
class Child1 extends Parent
{
    var Age: Int = 25
    def details1()
    {
        println(" Name is: " + Name1)
```

```
        println(" Age is: " + Age)
    }
}

// Derived from the Parent class
class Child2 extends Parent
{
    var Height: Int = 156

    // Method
    def details2()
    {
        println(" Name is: " + Name2)
        println(" Height is: " + Height)
    }
}

// Creating object
object PFP
{

    // Driver code
    def main(args: Array[String])
    {

        // Creation of objects of both derived
classes
        val ob1 = new Child1();
        val ob2 = new Child2();
        ob1.details1();
        ob2.details2();
    }
}
```

Parent is the base class in the preceding example. Child1 and Child2 are derived classes from Parent through the extends keyword. When we construct the objects of the Child1 and Child2 classes in the main method, a duplicate of all the methods and fields of the base class is stored in memory in this object.

Example:

```
// program of extending a class

// Base class
class Bicycle (val gearVal:Int, val speedVal: Int)
```

```
{
    // Bicycle class has two fields
var gear: Int = gearVal
var speed: Int = speedVal

// Bicycle class has two methods
def applyBreak(decrement: Int)
{
    gear = gear - decrement
    println("new gear value is: " + gear);
}
def speedUp(increment: Int)
{
    speed = speed + increment;
    println("new speed value is: " + speed);
}
}

// Derived class
class MountainBike(override val gearVal: Int,
                   override val speedVal: Int,
                   val startHeightVal : Int)
                   extends Bicycle(gearVal,
speedVal)
{
    // MountainBike subclass adds one more field
var startHeight: Int = startHeightVal

// MountainBike subclass adds one more method
def addHeight(newVal: Int)
{
    startHeight = startHeight + newVal
    println("new startHeight is: " + startHeight);
}
}

// Creating object
object PFP
{
    // the main method
    def main(args: Array[String])
    {
```

```
        val bike = new MountainBike(20, 10, 25);

        bike.addHeight(12);
        bike.speedUp(7);
        bike.applyBreak(7);
    }
}
```

CASECLASS AND CASEOBJECT IN SCALA

A caseClass is a standard class with the addition of a capability for representing unchangeable data. It is also beneficial in pattern matching. It has been specified with a modifier case; as a result of this case keyword, we may receive certain benefits to avoid doing portions of code that must include in many places with little or no modification. As seen below, a minimum caseClass requires the keyword caseClass, an identifier, and a parameter list that may be empty.

Syntax:

```
Case class class_Name(parameters)
```

Take note that the CaseClass contains a default apply() function that handles object construction.

CASEOBJECT EXPLANATION

A caseObject is similar to an object in that it contains more properties than a regular Object. It combines caseClasses with object classes. A caseObject has certain additional characteristics over a regular object.

Two significant characteristics of the caseObject are listed below:

- It can be serialized.
- It includes a hashCode implementation by default.

Example:

```
// program of the case class and case Object
case class employee (name:String, age:Int)
object Main
{
    // the main method
```

```
    def main(args: Array[String])
    {
        var c = employee("Shreya", 26)

        // Display the both Parameter
        println("Name of employee is " + c.name);
        println("Age of employee is " + c.age);
    }
}
```

Some advantages of caseClass/Object are as follows:

- One of the most significant advantages of CaseClass is that the Scala Compiler affixes a method with the name of the class and the same number of arguments as provided in the class description, allowing us to create instances of the CaseClass even in the absence of the keyword new.

Example:

```
// program of case class and case Object
// affix a method with name of the class
case class Book (name:String, author:String)
object Main
{
    // the main method
    def main(args: Array[String])
    {
        var Book1 = Book("Data Structure and
Algorithm", "cormen")
        var Book2 = Book("Computer Structure",
"Tanenbaum")

        // Display strings
        println("Name of Book1 is " + Book1.
name);
        println("Author of Book1 is " + Book1.
author);
        println("Name of Book2 is " + Book2.
name);
        println("Author of Book2 is " + Book2.
author);
    }
}
```

- The second advantage is that the Scala compiler by default affixes val or var to all constructor parameters, so we won't be able to reassign a new value to them once that class object is constructed. This is why, even in the absence of val or var, caseClass's constructor parameters will turn out to be class members, which is not feasible for regular classes.
- The Scala compiler additionally adds a copy() function to the caseClass, which is used to produce a replica of the same object with or without modifying any parameters.

  Example: To produce a copy of the same instance without changing the parameters.

```
// program of the case class To create
// duplicate of the same instance
case class Student (name:String, age:Int)
object Main
{
    // the main method
    def main(args: Array[String])
    {
        val s1 = Student("Shreya", 26)

        // Display the parameter
        println("The Name is " + s1.name);
        println("The Age is " + s1.age);
        val s2 = s1.copy()

        // Display the copied data
        println("Copy Name: " + s2.name);
        println("Copy Age: " + s2.age);
    }
}
```

- In this case, we generated a new object s2 by copying the s1 object without changing any of its characteristics.

  Example: To make a copy of the same object with different properties.

```
// program of the case class same object
// with the changing attributes
case class Student (name:String, age:Int)

object Main
{
    // the main method
```

```
    def main(args: Array[String])
    {
        val s1 = Student("Shreya", 26)

        // Display parameter
        println("The Name is " + s1.name);
        println("The Age is " + s1.age);
        val s2 = s1.copy(age = 25)

        // Display copied and changed attributes
        println("The Copy Name is " + s2.name);
        println("The Change Age is " + s2.age);
    }
}
```

- Scala Compiler adds toString, equals methods, companion object with apply and unapply methods by default, so we don't need the new keyword to generate a CaseClass object.

POLYMORPHISM SCALA

Polymorphism refers to any data's capacity to be processed in more than one way. The term implies the meaning itself since poly means many and morphism denotes kinds. Polymorphism is implemented in Scala via virtual functions, overloaded functions, and overloaded operators. Polymorphism is a crucial topic in OOP languages. Polymorphism is most commonly employed in OOP when a parent class reference is used to refer to a child class object. We will explore how to represent any function in various kinds and forms. A person can simultaneously have several roles in life, which is an example of polymorphism. A woman is a mother, a wife, an employee, and a daughter all at the same time. So the same individual must have numerous traits but must implement each according to the circumstance and conditions. Polymorphism is regarded as an important aspect of OOP. In Scala, the function can use to arguments of several kinds, or the type itself can include instances of various types.

Polymorphism is classified into two types:

- **Subtyping:** A subClass's instance can be provided to a base class via subtyping.
- **Generics:** Type parameterization creates instances of a function or class.

Here are a few examples:

First example:

```
// program to shows the usage of
// many functions with same name
class example
{

    // This is the first function with name fun
    def func(a:Int)
    {
        println("The First Execution:" + a);
    }

    // This is the second function with name fun
    def func(a:Int, b:Int)
    {
        var sum = a + b;
        println("The Second Execution:" + sum);
    }

    // This is the first function with name fun
    def func(a:Int, b:Int, c:Int)
    {
        var product = a * b * c;
        println("The Third Execution:" + product);
    }
}

// Creation of object
object Main
{
    // the main method
    def main(args: Array[String])
    {
        // Creation of object of example class
        var ob = new example();
        ob.func(140);
        ob.func(30, 60);
        ob.func(20, 15, 9);
    }
}
```

Second example:

```
// program to illustrate polymorphism concept
class example2
{
    // Function1
    def func(vehicle:String, category:String)
    {
        println("The Vehicle is:" + vehicle);
        println("The Vehicle category is:" +
category);
    }

    // Function2
    def func(name:String, Marks:Int)
    {
        println("The Student Name is:" + name);
        println("The Marks obtained are:" +
Marks);
    }

    // Function3
    def func(a:Int, b:Int)
    {
        var Sum = a + b;
        println("The Sum is:" + Sum)
    }

}

// Creation of object
object Main
{
    // the main method
    def main(args: Array[String])
    {
        var A = new example2();
        A.func("swift", "hatchback");
        A.func("honda-city", "sedan");
        A.func("Ashok", 85);
        A.func(20, 30);
    }
}
```

VALUE CLASSES IN SCALA

Value classes are a novel method that aids in avoiding the allocation of run-time objects. AnyVal defines value classes. Value classes are predefined and correspond to the basic type of Java-like languages.

Double, Float, Long, Int, Short, Byte, Char, Unit, and Boolean are the nine predefined value types.

A value class cannot redefine equals or hashCode. Value classes are primarily used to improve speed and memory management.

Let's look at few examples to grasp value classes better.

First example:

```
// program to illustrate the value class

// Creation of a value class and extend with
AnyVal
case class C(val name: String) extends AnyVal

// Creation of object
object pfp
{
    // the main method
    def main (args: Array[String])
    {
        // Creating instance of the ValueClass
        val c = new C("PeeksForPeeks")
        c match
        {
            // new C instantiated here
            case C("PeeksForPeeks") =>
println("Matched with PeeksForPeeks")
            case C(x) => println("Not matched with
PeeksForPeeks")
        }
    }
}
```

In the preceding code, a value class is defined using the caseClass, and AnyVal is used to define the value class (C). The value class has one string argument. If we provide the same string as in the case statement, this will return true else false.

Second example:

```
// program to illustrate value class

// Creation of the value class and extend with
AnyVal
class Vclass(val a: Int) extends AnyVal
{
    // Defining the method
    def square() = a*a
}

// Creation of object
object pfp
{
    // the main method
    def main (args: Array[String])
    {
        // creation of the instance of the
ValueClass
        val v = new Vclass(6)
        println(v.square())
    }
}
```

As we can see in the above instance, a value class was formed, and the representation is an int. The above code includes a definition in the value class Vclass. Vclass is a user-defined value class that encompasses a square method and wraps the Int argument. To invoke the square method, make an object of the Vclass class like follows: new Vclass = val v (6).

Some value class restrictions are as follows:

- It is possible that a value class does not contain specialized type arguments. There may be no specialized type parameters.
- There may be no nested or local classes, characteristics, or objects in a value class.
- A value class cannot redefine equals or hashCode.
- Lazy vals, vars, and vals cannot be members of a value class. Its members can only be defs.
- A value class cannot be extended by any other class.

FIELD OVERRIDING IN SCALA

Overriding is a feature in any object-oriented computer language that allows a subClass to offer a customized implementation of a method or field that is already supplied by one of its super-classes. Overriding is more plainly specified in Scala than in Java, as both methods and fields can be overridden, although a few constraints must follow.

Overriding Rules for the Field

The field overriding rules are as follows:

- One of the most significant requirements is that when overriding the fields of the super-class in the sub-classes, we must use the keyword override or override notation; otherwise, the compiler will raise an error and stop the program's execution.
- To do a Field Overriding, we must override variables specified using simply the val keyword in both the super-class and sub-classes.
- Because we can read and write var, field overriding cannot override it.

Example:

```
// program of Field Overriding

// Creation of class
class Shapes
{

    // Creation of a variable with val keyword
    val description:String = "shape"
}

// Creating a subclass
class shape1 extends Shapes
{

    // Overriding field using the
    // 'override' keyword
    override val description:String ="It is a
circle."
```

```
    // Defining the method
    def display()
    {

        // Display the output
        println(description)
    }
}

// Creation of a subclass
class shape2 extends Shapes
{

    // overriding field using the
    // 'override' keyword
    override val description:String ="It is a
square."

    // Defining the method
    def display()
    {

        // Display the output
        println(description)
    }
}

// Creation of object
object PfP
{

    // the main method
    def main(args:Array[String])
    {

        // Creation of instances for all
        // sub-classes
        var x = new shape1()
        var y = new shape2()

        // Calling methods
        x.display()
```

```
        y.display()

    }
}
```

As we can see from the following code, val is used in both the super class and the subClass; therefore overriding the field was possible; otherwise, an error would have been thrown. The field in the superClass is val, which is overridden by the override keyword in the sub-classes.

Example:

```
// program of Field Overriding

// Creation of class
class Shapes
{

    // Creation of a variable with val keyword
    val description:String = "shape"
}

// Creation of a subclass
class shape1 extends Shapes
{

    // Overriding field using the
    // 'override' keyword
    override var description:String ="It is a
circle."

    // Defining the method
    def display()
    {

        // Display the output
        println(description)
    }
}

// Creating a subclass
class shape2 extends Shapes
```

```
{

    // overriding field using the
    // 'override' keyword
    override var description:String ="It is a
square."

    // Defining the method
    def display()
    {

        // Display the output
        println(description)
    }
}

// Creation of object
object PfP
{

    // the main method
    def main(args:Array[String])
    {

        // Creation of instances for all
        // sub-classes
        var x = new shape1()
        var y = new shape2()

        // Calling the methods
        x.display()
        y.display()

    }
}
```

It is the same instance as earlier, but now issues are identified because var is used in the sub-classes to override the fields, which is not possible because, as stated above, val cannot be overridden by the var.

Example:

```
// program of Field Overriding

// Creation of class
class Animals
{

    // Creation of a variable with the 'var'
keyword
    var number:Int = 2
}

// Creation of a subclass
class Cat extends Animals
{

    // Overriding field using the 'override'
keyword
    override var number:Int = 4

    // Defining the method
    def show()
    {

        // Display the output
        println("We have " + number + " cats.")
    }
}

// Creation of object
object PfP
{

    // the main method
    def main(args:Array[String])
    {

        // Creation of instance of
        // the sub-class
        var cat = new Cat()
```

```
        // Calling the method
        cat.show()

    }
}
```

ABSTRACT TYPE MEMBERS IN SCALA

If a member of a class or trait lacks a complete definition in the class, that member is said to be abstract. These abstract members are always implemented in any subClasses of the class that defines them. Many programming languages provide this type of declaration, and it is an important feature of object-orientated programming languages. Scala also allows us to specify such methods, as illustrated in the following example:

```
abstract class Samples{
    def contents: Array[String]
    def width: Int = a
    def height: Int = b
  }
```

Thus, in the preceding Samples class, we specified three methods: contents, width, and height. The implementation of the final two methods is already defined, however, no implementation is mentioned in the first method, contents. As a result, this method is an abstract member of the Samples class. It's worth noting that a class containing abstract members must also be declared abstract. The abstract keyword in front of the class indicates that the class will almost certainly have an abstract member with no implementation.

Another example of how to build abstract class members:

```
abstract class Examples{
    type T
    def transform(m: T): T
    val initial: T
    var current: T
 }
```

The abstract class is declared in the above example, which specifies an abstract type T, an abstract method transform, an abstract value initial, and an abstract value current.

The following is an instance of an abstract type member:

```
// program of the abstract type member

// Declarion of an abstract class
abstract class vehicle (name:String)
{
    // This is abstract member
    // with the undefined implementation
    val category: String

    // function that is used to print
    // value of the abstract member
    def cartype{ println(category) }
    override def toString = s" Vehicle type is
$category"
}

// Now extend classes bike, car,
// truck and bus and provide values for variable type

class car (name:String) extends vehicle (name)
{
    // assigning value of abstract member as car
    val category = "car"
}
class bike (name:String) extends vehicle (name)
{
    // assigning the value of the
    // abstract member as bike
    val category = "bike"
}
class bus (name:String) extends vehicle (name)
{
    // assigning value of the
    // abstract member as bus
    val category = "bus"
}
class truck (name:String) extends vehicle (name)
{
    // assigning value of the
    // abstract member as truck
    val category = "truck"
}
```

```
object AbstractFieldsDemo extends App
{
    // assigning name as Honda in the abstract
    // class where category value is car
    val car = new car("Honda")

    // assigning the name as Yamaha in the abstract
    // class where the category value is bike
    val bike = new bike("Yamaha")

    // assigning name as Tata in the abstract
    // class where category value is bus
    val bus = new bus("Tata")

    // assigning name as Ashok_Leyland in the
    // abstract class where category value is truck
    val truck = new truck("Ashok_Leyland")

    // function implementation
    // cartype for the object car
    car.cartype

    // function implementation
    // cartype for the object bus
    bus.cartype

    // function implementation
    // cartype for the object truck
    truck.cartype

    // function implementation
    // cartype for the object bus
    bike.cartype
    println(car)
    println(bus)
    println(truck)
    println(bike)
}
```

The trait vehicle in the above example has an abstract val category, a simple concrete method named cartype, and an override of the toString function. The classes vehicle is then extended by the classes car, bike, truck, and bus, which offer values for the field category.

In the preceding code, we can see how the abstract member's undefined implementation is being utilized to assign values and alter the assigned values for each object of a different sort. In this instance, we saved many category values for various vehicle types.

As a result, the abstract data members are those with an uncertain implementation.

SCALA TYPE CASTING

A type casting is essentially a conversion from one type to another. Casting from one type to another is commonly utilized in Dynamic Programming Languages such as Scala. The asInstanceOf[] function is used in Scala to cast types.

asInstanceof method applications:

- This viewpoint is necessary for manifesting beans from an application context file.
- It is also used to cast numbers.
- It may also use in more complicated programming, such as communicating with Java and passing an array of Object objects to it.

Syntax:

```
obj1 = obj.asInstanceOf[class];
where,
obj1 is the object to which casted instance of obj
is returned, obj represents the object to be
casted, and class represents the name of the class
into which obj is to be casted.
```

Only object of an extended(child) class can cast to be an object of its parent class, not the other way around. If class A extends class B, the object of class A may cast to be an object of class B, but the object of class B cannot convert to be an object of class A. This method notifies the compiler that the value is of the given type. An exception is triggered during runtime if the value/object given is incompatible with the type or class specified.

Example:

```
// program of type casting
object PFP
```

```
{
    // Function to display the name, value and
    // classname of a variable
    def display[A](y:String, x:A)
    {
        println(y + " = " + x + " is of type " +
                              x.getClass.getName);
    }

    // the main method
    def main(args: Array[String])
    {
        var i:Int = 60;
        var f:Float = 7.0F;
        var d:Double = 92.4;
        var c:Char = 'p';

        display("i", i);
        display("f", f);
        display("d", d);
        display("c", c);

        var i1 = i.asInstanceOf[Char]; //Casting
        var f1 = f.asInstanceOf[Double]; //Casting
        var d1 = d.asInstanceOf[Float]; //Casting
        var c1 = c.asInstanceOf[Int]; //Casting

        display("i1", i1);
        display("f1", f1);
        display("d1", d1);
        display("c1", c1);
    }
}
```

SCALA OBJECT CASTING

It is required to utilize the asInstanceOf method when casting an Object (i.e., instance) from one type to another. This function is specified in the Scala class hierarchy's root, Class Any (like Object class in Java). The asInstanceOf method of Class Any relates to the concrete value members and is used to cast the recipient Object.

SCALA OBJECT EQUALITY

Comparing two values for equality is common in programming languages. We construct an equals method for a Scala class so that we may compare object instances. The equality function in Scala represents object identity, however, it is rarely used.

Scala provides three distinct equality techniques:

- The equals Method
- The == and != Methods
- The ne and eq Methods

Note that eq functions similarly to the == operator in Java, C++, and C#, but not in Ruby. The ne method is the inverse of eq, which is identical to !(x eq y). The == operator in Java, C++, and C# checks for reference equality rather than value equality. Ruby's == operator, on the other hand, checks for value equality. In Scala, however, == tests for value equality.

Let's look at an example.

```
// program of Equals

// Creation of a case class of Subject
case class Subject (LanguageName:String,
TopicName:String)

// Creation of object
object PFP
{
    // the main method
    def main(args: Array[String])
    {
        // Creation of objects
        var x = Subject("Scala", "Equality")
        var y = Subject("Scala", "Equality")
        var z = Subject("Java", "Array")

        // Display the true if instances are equal
else false
        println(x.equals(y))
        println(x.equals(z))
```

```
        println(y == z)

    }
}
```

- equals Method: The equals method is used to test for value equality. If both x and y have the same value, the expression if x equals y is true. They do not have to refer to the same instance. As a result, the equals method in Java and the equals method in Scala work identically.
- The == and != Methods: While == is an operator in many languages, Scala reserves The == equality for all natural equality. It's a Scala method that's declared as final in Any. This will put value equality to the test. If both x and y have the same value, then x == y.
- The ne and eq methods will use to test reference equivalence. If both x and y point to the same address in memory or reference the same object, x eq y is true. These techniques are applicable solely to AnyRef.

If two objects are equal as per equals method, then applying the hash code method to each must have the same numeric result. Equals (and, ==) has the same behavior as eq by default, but we can change it by overriding the equals method in the classes we define. Scala treats == as though it were declared in class Any: as follows.

The following is an example of the equals method and its corresponding hashCode method:

Example:

```
// program to illustrate how
// the hashCode() and equals() methods work

// Creation of class
class Subject (name: String, article: Int)
{
    // Defining the canEqual method
    def canEqual(a: Any) = a.isInstanceOf[Subject]

    // Defining equals method with the override
keyword
```

```
    override def equals(that: Any): Boolean =
        that match
    {
        case that: Subject => that.canEqual(this)
&&
                    this.hashCode == that.hashCode
        case _ => false
    }

    // Defining the hashcode method
    override def hashCode: Int = {
        val prime = 33
        var result = 1
        result = prime * result + article;
        result = prime * result +
                (if (name == null) 0 else name.
hashCode)
        return result
    }
}

// Driver code
object PFP
{
    // the main method
    def main(args: Array[String])
    {

        // Creating the Objects of Peek class.
        // Subject p1 = new Subject("aa", 1);
        val p1 = new Subject("Scala", 29)
        val p2 = new Subject("Scala", 29);


        // Comparing above created Objects.
        if(p1.hashCode() == p2.hashCode())
        {

            if(p1.equals(p2))
                println("Both the Objects are
equal. ");
            else
```

```
                println("Both the Objects are not
equal. ");

        }
        else
            println("Both the Objects are not
equal. ");
    }
}
```

In the preceding example, a modified version of a hashCode function created by Eclipse for a comparable Java class. It also use the canEqual technique. With the equals method implemented, we can use == to compare instances of a Subject.

MULTITHREADING IN SCALA

The process of operating many threads at the same time is referred to as multithreading. It enables us to carry out different tasks individually.

WHAT EXACTLY ARE THREADS IN SCALA?

Threads are little subprocesses that use little memory. A multi-threaded application comprises two or more threads that may operate concurrently, and each thread can do a distinct job at the same time, making the most use of the available resources, which is especially important when our system (computer) has several CPUs. Scala uses multithreading to create concurrent applications.

Scala threads may be generated using two mechanisms:

- Extending Thread class
- Extending Runnable Interface

Thread Creation by Extending Thread Class

We design a class that extends the Thread class. This class overrides the Thread class's run() function. A thread begins its life inside the run() procedure. To begin thread execution, we construct an instance of our new class and use the start() function. Start() calls the run() function on the Thread object.

```
// code for thread creation by extending Thread class
class MyThread extends Thread
```

```
{
    override def run()
    {
        // Displayingthread that is running
        println("Thread " + Thread.currentThread().
getName() +
                                        " is running.")
    }
}

// Creation of object
object PFP
{
    // the main method
    def main(args: Array[String])
    {
        for (y <- 1 to 6)
        {
            var th = new MyThread()
                        th.setName(y.toString())
                            th.start()
        }
    }
}
```

Thread Creation by Extending the Runnable Interface

We build a new class that extends the Runnable interface and overrides the run() function. Then we build a Thread object by sending the newly generated class to the constructor. We then invoke the start() function on this object.

```
// code for thread creation by implementing Runnable
Interface
class MyThread extends Runnable
{
    override def run()
    {
        // Displaying thread that is running
        println("Thread " + Thread.currentThread().
getName() +
                                        " is running.")
    }
}
```

```
// Creation of object
object MainObject
{
    // the main method
    def main(args: Array[String])
    {
        for (y <- 1 to 6)
        {
            var th = new Thread(new MyThread())
                        th.setName(y.toString())
                            th.start()
        }
    }
}
```

Threads do not have to run in any particular order. All threads operate continuously and independently of one another.

THREAD LIFE CYCLE IN SCALA

A Scala Thread goes through several state changes between the time it is created and when it is terminated. These are the stages of a Scala Thread's life. It has the five states listed below.

- New: When the Thread is first formed, it is in this condition.
- Runnable: This is the state in which the Thread has been generated but has yet to begin running.
- Running: The Thread is in this condition because it is completing its duty.
- Blocked (or Waiting): This is the condition of a thread that is still alive but unable to run owing to a lack of input or resources.
- Terminated: When a thread's run() function returns an error, it is said to be terminated.

SCALA FINAL

Final is a keyword in Scala used to place restrictions on superClasses or parent classes in various ways. We can use the final keyword in conjunction with variables, methods, and classes.

The following are examples of using the final keyword in Scala.

1. **Scala final variable:** A Scala final variable is defined once and then utilized as a constant throughout the program. In the following example, variable area is defined as final and also initialized when declared in superClass shapes. If we wish to access or edit the variable area from the derived class Rectangle, we cannot since the keyword final applies the variable area limitation.

   Scala final variables can initialize in the following ways:

   - While declaring
   - In static block
   - In Constructor

```
// program of using the final variable
class Shapes
{
    // define the final variable
    final val area:Int = 80
}
class Rectangle extends Shapes
{
    override val area:Int = 120
    def display()
    {
        println(area)
    }
}

// Creation of object
object PFP
{
    // the main method
    def main(args:Array[String])
    {
        var m = new Rectangle()
        m.display()
    }
}
```

2. **Scala final methods:** This is the final method CalArea in the parent class (Shapes) indicates that methods in a child class cannot be overridden (Rectangle).

```
// program of using the final method
class Shapes
{
    val height:Int = 0
    val width :Int =0

    // Define the final method
    final def CalArea(){
    }
}
class Rectangle extends Shapes
{
    override def CalArea()
    {
        val area:Int = height * width
        println(area)
    }
}

// Creation of object
object PFP
{
    // the main method
    def main(args:Array[String])
    {
        var m = new Rectangle()
        m.CalArea()
    }
}
```

3. **Scala final classes:** If a Scala class is final, it cannot inherit from derived class. The final keyword will introduce inheritance restrictions. If the class Shapes is final, then all of its members are also final and cannot be utilized in derived class.

```
// program of using the final class
final class Shapes
{
    // Final variables and functions
```

```
    val height:Int = 0
    val width :Int =0
    final def CalArea()
    {
    }
}
class Rectangle extends Shapes
{
    // Cannot inherit the Shapes class
    override def CalArea()
    {
        val area:Int = height * width
        println(area)
    }
}

// Creation of Object
object PFP
{
    // the main method
    def main(args:Array[String])
    {
        var m = new Rectangle()
        m.CalArea()

    }
}
```

SCALA THIS KEYWORD

Keywords are terms used in a language to describe predetermined actions or internal processes. When we wish to introduce the current object for a class, we utilize this keyword. We may then refer to instance variables, methods, and constructors by using the dot operator (.). This keyword is also used in conjunction with auxiliary constructors.

Let's look at few examples to grasp this keyword better.

```
// program to illustrate this keyword
class Addition(x:Int)
{
    // using this keyword
    def this(x:Int, y:Int)
```

```
    {
        this(x)
        println(x + " + " + y + " = " + { x + y })
    }
}

// Creation of object
object PFP
{
    // the main method
    def main(args:Array[String])
    {
        var add = new Addition(18, 14)
    }

}
```

CONTROLLING VISIBILITY OF CONSTRUCTOR FIELDS IN SCALA

The visibility of Constructor Fields in the Scala language is controlled and regulated by declaration. These can be declared in the following formats:

- Declared as val
- Declared as var
- Declared without var and val
- Add Private to the fields

We'll go through all of the following approaches in further depth, with the assistance of several examples:

- **When a var field is defined:** When a field is specified as var, the Scala language creates both Getter and Setter modes for that variable. This indicates that the field's value can always modify.
- **When the field is specified to be val:** If the field is defined as val, the value of the fields assigned at the beginning cannot be modified and stays fixed. In this scenario, Scala only supports the getter function.
- **When a field is specified without the variables val and var:** If field is declared without var and val, its visibility is severely limited because

Scala does not support setter and getter functions. The visibility of the field is reduced.

- **Including the keyword Private:** In addition to the var and val modes, we may use the term "private." This makes the field accessible in the same way as C++ does. This disables the methods getter and setter, and the field is ordinarily accessible using the class's member functions.

We covered object-oriented programming ideas, including syntax and examples, in this chapter.

CHAPTER 5

Scala String and Packages

IN THIS CHAPTER

- String
- String Interpolation
- StringContext
- Regular Expressions
- StringBuilder and concatenation
- Packages
- File Handling

In the previous chapter, we covered Scala control statements, and in this chapter, we will discuss strings and packages.

SCALA STRING

A string is a series of characters. String objects in Scala are immutable, meaning they cannot modify once created.

DOI: 10.1201/9781003358527-5

SCALA STRING CREATION

In Scala, there are two ways to construct a string:

- When compiler encounters a string literal, it generates a string object str.

```
var str = "Hello! PFP"
or
val str = "Hello! PFP"
```

- Before meeting the string literal, a String type is specified.

Syntax:

```
var str: String = "Hello! PFP"
or
val str: String = "Hello! PFP"
```

Use the StringBuilder class if we need to add to the original string.

Example:

```
// program to illustrate how to
// create the string
object Main
{

    // str1 and str2 are the two different strings
    var str1 = "Hello! PFP"
    val str2: String = "PeeksforPeeks"
    def main(args: Array[String])
    {

        // Display both the strings
        println(str1);
        println(str2);
    }
}
```

DETERMINE THE LENGTH OF THE STRING

An accessor method is used to find information about an object. In Scala, a length() method is an accessor function used to determine a given string's length. In other terms, the length() function returns the number of characters in the string object.

Syntax:

```
var len1 = str1.length();
```

Example:

```
// program to illustrate how to
// get a length of the given string
object Main
{

    // str1 and str2 are the two strings
    var str1 = "Hello! PFP"
    var str2: String = "PeeksforPeeks"

    // the main function
    def main(args: Array[String])
    {

        // Get length of str1 and str2 strings
        // using length() function
        var LEN1 = str1.length();
        var LEN2 = str2.length();

        // Display the both strings with their
length
        println("String 1 is:" + str1 + ",
Length :" + LEN1);
        println("String 2 is:" + str2 + ",
Length :" + LEN2);
    }
}
```

CONCATENATING STRINGS IN SCALA

A string concatenation occurs when a new string is formed by joining two existing strings. Scala has a concat() method for concatenating two strings; this method returns a new string generated by concatenating two strings. To concatenate two strings, use the "+" operator.

Syntax:

```
str1.concat(str2);
```

Example:

```
// program to illustrate how to concatenate
strings
object Main
{

    // str1 and str2 are the two strings
    var str1 = "Welcome, PeeksforPeeks "
    var str2 = " to Session"

    // the main function
    def main(args: Array[String])
    {

        // concatenate the str1 and str2 strings
        // using concat() function
        var Newstr = str1.concat(str2);

        // Display the strings
        println("String 1:" +str1);
        println("String 2:" +str2);
        println("New String :" +Newstr);

        // Concatenate the strings using '+'
operator
        println("This is the lessons" +
                    " of the Scala language" +
                    " on PFP portal");
    }
}
```

CREATING FORMAT STRING

When we need to format a number or values in a string, we'll utilize the printf() or format() methods. Aside from these methods, the String class also has a format() function that returns a String object rather than a PrintStream object.

Example:

```
// program to illustrate how to
// Creation of format string
object Main
```

```
{

    // two strings
    var A_name = "Shreya"
    var Ar_name = "Scala|Strings"
    var total = 150

    // the main function
    def main(args: Array[String])
    {

        // using the format() function
        println("%s, %s, %d".format(A_name,
Ar_name, total));
    }
}
```

Some useful string methods are as follows:

Function	Description
char charAt(int index)	This function returns the character at the provided index.
String replace(char ch1, char ch2)	This method returns a new string with the element of ch1 replaced by the element of ch2.
String[] split(String reg)	This method separates the string based on regular expression matches.
String substring(int i)	This method generates a new string that is a substring of the provided string.
String trim()	This method returns a string with the beginning and ending whitespace removed.
boolean startsWith(String prefix)	This function determines whether or not the provided text begins with the specified prefix.

STRING INTERPOLATION IN SCALA

String interpolation is the substitution of identified variables or expressions in a given String with respected values. String interpolation makes it simple to parse String literals. To use this Scala functionality, we must follow a few rules:

- Strings must begin with the characters s / f /raw.
- Variables in the String must prefix with "$."
- Expressions must enclose in curly braces ({, }), and a prefix of "$" is added.

Syntax:

```
// c and d are defined
val str = s"Sum of $c and $d is ${c+d}"
```

STRING INTERPOLATOR TYPES

- **s Interpolator:** We may access variables, object fields, function calls, and so on within the String.

  First example: Variables and expressions.

```
// program for s interpolator

// Creation of object
object PFP
{
    // the main method
    def main(args:Array[String])
    {

        val c = 40
        val d = 20

        // without s interpolator
        val str1 = "The Sum of $c and $d is
${c+d}"

        // with s interpolator
        val str2 = s"The Sum of $c and $d is
${c+d}"

        println("str1: "+str1)
        println("str2: "+str2)
    }
}
```

  Second example: Function call.

```
// program for s interpolator

// Creation of object
object PFP
```

```
{
    // adding the two numbers
    def add(c:Int, d:Int):Int
    =
    {
        c+d
    }

    // the main method
    def main(args:Array[String])
    {

        val c = 40
        val d = 20

        // without s interpolator
        val str1 = "The Sum of $c and $d is
${add(c, d)}"

        // with s interpolator
        val str2 = s"the Sum of $c and $d is
${add(c, d)}"

        println("str1: " + str1)
        println("str2: " + str2)
    }
}
```

- **f Interpolator:** This interpolation aids in the easy formatting of numbers.

  Format Specifiers explains how to format specifiers function.

Example: Printing upto 2 decimal place:

```
// program for f interpolator

// Creation of object
object PFP
{
    // the main method
    def main(args:Array[String])
    {

        val x = 32.6
```

```
        // without the f interpolator
        val str1 = "The Value of x is $x%.2f"

        // with the f interpolator
        val str2 = f"The Value of x is $x%.2f"

        println("str1: " + str1)
        println("str2: " + str2)

    }
}
```

- **raw Interpolator:** String Literal should begin with the word "raw." Escape sequences are treated the same as any other character in a String by this interpolator.

  Example: Printing an escape sequence.

```
// program for raw interpolator

// Creation of object
object PFP
{
    // the main method
    def main(args:Array[String])
    {

        // without the raw interpolator
        val str1 = "Hello\nEveryone"

        // with the raw interpolator
        val str2 = raw"Hello\nEveryone"

        println("str1: " + str1)
        println("str2: " + str2)
    }
}
```

StringContext IN SCALA

StringContext is a class used in string interpolation that allows end users to add variable references directly into processed String literals. This class includes raw, s, and f methods as interpolators by default. The Linear

Supertypes, in this case, are Serializable, java.io. Serializable, Product, Equals, AnyRef, and Any.

- An example of using the s-method as an interpolator.

  Example:

```
// program of StringContext

// Creation of object
object Main
{

    // the main method
    def main(args: Array[String])
    {

        // Assigning the values
        val name = "PeeksforPeeks"
        val articles = 29

        // Applying the StringContext with
s-method
        val result = StringContext("We have
written ",
                    " articles on ",
".").s(articles, name)

        // Display the output
        println(result)

    }
}
```

  In this case, the StringContext.s method extracts the constant sections, translates the escape sequences inside them, and combines them with the values of the specified expression parameters.

- Creating our own interpolator: To provide our own String interpolator, we must create an implicit class that will connect a method to the StringContext class.

  Example:

```
// program of StringContext for creating our own
string interpolator
```

```
// Creation of object
object Main
{

// the main method
def main(args: Array[String])
{
    // Using the implicit class with
StringContext
    implicit class Reverse (val x :
StringContext)
    {

        // Defining the method
        def revrs (args : Any*) : String =
        {

            // Applying the s-method
            val result = x.s(args : _*)

            // Applying the reverse method
            result.reverse
        }
    }

    // Assigning the values
    val value = "PeeksforPeeks"

    // Displays reverse of the stated string
    println (revrs"$value")
}
}
```

SCALA REGULAR EXPRESSIONS

Regular Expressions describe a common pattern used to match a succession of input data, making it useful in Pattern Matching in various computer languages. Scala Regex is the common word for regular expressions in Scala.

Regex is a class imported from the package scala.util.matching.Regex that is widely used in search and text processing. To recast a string into a Regular Expression, use the r() function with the specified string.

Example:

```
// program for Regular Expressions

// Creation of object
object PfP
{

    // the main method
    def main(args: Array[String])
    {

        // Applying the r() method
        val portal = "PeeksforPeeks".r
        val CS = "PeeksforPeeks is a CS portal."

        // Displays first match
        println(portal findFirstIn CS)
    }
}
```

To generate a pattern, we invoked the function r() on the specified string to generate an instance of the Regex class. In the preceding code, the function findFirstIn() is used to find the first match of the Regular Expression. Use the findAllIn() function to find all the expressions' matching words.

```
// program for Regular Expressions
import scala.util.matching.Regex

// Creation of object
object PfP
{

    // the main method
    def main(args: Array[String])
    {

        // Applying the Regex class
        val x = new Regex("Shreya")
        val myself = "My name is Shreya Sood."
```

```
        // replaces first match with String given below
        println(x replaceFirstIn(myself, "Ridhi"))
    }
}
```

We may even utilize the Regex constructor instead of the r() function. The replaceFirstIn() function is used to replace the first match of the specified string, while the replaceAllIn() method is used to replace all matches.

Example:

```
// program for Regular Expressions
import scala.util.matching.Regex

// Creation of object
object PfP
{

    // the main method
    def main(args: Array[String])
    {

        // Applying the Regex class
        val Peeks = new Regex("(G|g)fG")
        val y = "PfP is a CS portal. I like pfP"

        // Displays all matches separated
        // by separator
        println((Peeks findAllIn y).mkString(", "))
    }
}
```

We utilized the mkString function to concatenate all the matches separated by a separator, and a pipe (|) is used in the above code to search for both upper and lower case in the provided text. As a result, both the upper and lower case of the specified string is returned here.

Scala Regular Expression Syntax

Java inherits several features from Perl, whereas Scala inherits the Scala regex syntax from Java. The following is a list of metacharacter syntax:

Subexpression	Matches
^	It is used to match the line's starting point.
$	It is used to match the line's ending point.
.	It is used to match any one character other than the newline.
[…]	It is used to match any one character included within the brackets.
[^…]	It is used to match any one scharacter that is not contained inside the brackets.
\\A	It is used to find the beginning of the intact string.
\\z	It is used to find the end of the intact string.
\\Z	It matches the end of the whole string, omitting any new lines that may appear.
re*	It is used to match zero or more occurrences of the preceding expressions.
re+	It corresponds to one or more of the preceding expressions.
re?	It matches either zero or one occurrence of the preceding expression.
re{ n}	It is used to match exactly n instances of the preceding expression.
re{ n, }	It is used to find n or more instances of the preceding expression.
re{ n, m}	It matches at least n and no more than m occurrences of the preceding expression.
q|r	It may use to match either q or r.
(re)	It is used to group Regular expressions and remember the matched text.
(?: re)	It also groups the regular expressions but does not remember the matched text.
(?> re)	It is used to match a self-sufficient pattern in the absence of backtracking.
\\w	It is used to match word characters.
\\W	It is used to match non-word characters.
\\s	It is used to match white spaces that are similar to [tnrf].
\\S	It's used to fill in non-white spaces.
\\d	It matches digits, such as [0-9].
\\D	It's used to find non-digits.
\\G	It is used to match the moment at which the final match is over.
\\n	It is used to occupy group number n for back-reference.
\\b	It matches the word frontiers when they are outside the brackets and the backspace when they are inside the brackets.
\\B	It is used to find non-word frontiers.
\\n, \\t, etc.	It is used to match newlines, tabs, and so forth.
\\Q	It is used to escape (quote) each character till \\E.
\\E	It is used in quotes that begin with \\Q.

SCALA StringBuilder

A String object is immutable, meaning it cannot modify once generated. StringBuilder is useful in instances when we need to make repetitive changes to a string. To add input data to the internal buffer, StringBuilder is used. Using methods on the StringBuilder, we can conduct various operations. This operation includes adding, inserting, and deleting data.

Important details:

- The StringBuilder class is useful for effectively extending mutable strings.
- StringBuilder instances are used in the same way as Strings.
- Because Scala Strings are immutable, we can use StringBuilder to create changeable Strings.

The StringBuilder Class Performs Operations

- **Character adding:** This procedure is useful for character appending.

  Example:

```
// program to append a character

// Creation of object
object PFP
{

    // the main method
    def main(args: Array[String])
    {

        // Creation of StringBuilder
        val x = new StringBuilder("Author");

        // Appending the character
        val y = (x += 's')

        // Displays string after
        // appending character
        println(y)
    }
}
```

- **String appending:** This operation is useful for string appending.

  Example:

```
// program to append a String

// Creation of object
object PFP
{

    // the main method
    def main(args: Array[String])
    {

        // Creation of StringBuilder
        val x = new StringBuilder("Authors");

        // Appending the String
        val y = (x ++= " of PeeksforPeeks")

        // Displays string after
        // appending the string
        println(y)

    }
}
```

- **Appending String representation of a number:** The number can be of any type, such as Integer, Double, Long, Float, etc.

  Example:

```
// program to append String representation of
the number

// Creation of object
object num
{

    // the main method
    def main(args: Array[String])
```

```
    {

        // Creation of StringBuilder
        val x = new StringBuilder("The Number of
Contributors : ");

        // Appending String representation of
the number
        val y = x.append(700)

        // Displays string after appending the
number
        println(y)
    }
}
```

- **Resetting StringBuilder's content:** It is useful for resetting the content by making it empty.

Example:

```
// program to reset the content

// Creation of object
object PFP
{

    // the main method
    def main(args: Array[String])
    {

        // Creation of StringBuilder
        val x = new StringBuilder("Hello")

        // Resetting content
        val y = x.clear()

        // Display the empty content
        println(y)
    }
}
```

- **Delete operation:** This operation is useful for removing characters from the StringBuilder's content.

  Example:

```
// program to perform delete operation

// Creation of object
object delete
{

    // the main method
    def main(args: Array[String])
    {

        // Creation of StringBuilder
        val q = new StringBuilder("Computer
Networking")

        // Deleting the characters
        val r = q.delete(1, 4)

        // Displaying the string after
        // deleting some characters
        println(r)
    }
}
```

- **Insertion operation:** This operation is useful for inserting Strings.

  Example:

```
// program to perform insertion operation

// Creating object
object insert
{

    // the main method
    def main(args: Array[String])
    {
```

```
        // Creation of StringBuilder
        val q = new StringBuilder("PfP CS
portal")

        // inserting the strings
        val r = q.insert(4, "is a " )

        // Display the string after
        // insertion of required string
        println(r)
    }
}
```

- **Converting StringBuilder to a String:** This operation converts StringBuilder to a String.

  Example:

```
// program of Converting StringBuilder to a
String

// Creation of object
object builder
{

    // the main method
    def main(args: Array[String])
    {

        // Creation of StringBuilder
        val q = new
StringBuilder("GeeksforGeeks")

        // Applying conversion operation
        val r = q.toString

        // Display the String
        println(r)
    }
}
```

SCALA STRING CONCATENATION

A string is a character sequence. String objects in Scala are immutable, which means they cannot be modified once generated. A string concatenation occurs when a new string is formed by joining two existing strings. Scala has a concat() method for concatenating two strings; this method returns a new string generated by concatenating two strings. We may also use the "+" operator to concatenate two strings.

Syntax:

```
str1.concat(str2);

Or

"str1" + "str2";
```

The following is an example of concatenating two strings.

Using the concat() method: The parameter is appended to the string using this method.

Example:

```
// program to illustrate how to concatenate
strings
object PFP
{

    // str1 and str2 are the two strings
    var str1 = "Welcome, PeeksforPeeks "
    var str2 = " to Portal"

    // the main function
    def main(args: Array[String])
    {

        // concatenate the str1 and str2 strings
        // using the concat() function
        var Newstr = str1.concat(str2);

        // Display the strings
        println("String 1 is:" +str1);
```

```
        println("String 2 is:" +str2);
        println("New String is:" +Newstr);

        // Concatenate strings using the '+'
operator
        println("This is the sessions" +
                    " of Scala programming" +
                    " on PFP portal");
    }
}
```

SCALA PACKAGES

In Scala, a package is a technique for encapsulating a collection of classes, subpackages, traits, and package objects. It just gives namespace for storing our code in multiple files and folders. Packages are a simple method to organize our code and avoid name conflicts between members of various packages. Providing access control to package members such as private, protected, and package-specific regulating scope limits the access of members to other packages, whereas members with no modifier can use within any other package with some reference.

PACKAGE DECLARATION

Packages are specified in the first statement of a Scala file.

Syntax:

```
package packagename
// Scala classes
// traits
// objects…
```

A package can define in several ways:

- **Chained methods**

```
package a.b.c
// members of c
```

- **Nesting packages**

```
package a{
    // members of a {as required}
    package b{
```

```
            // members of b{as required}
            package c{
               // members of c{as required}
            }
         }
      }
```

HOW PACKAGE FUNCTIONS

Packages link together data in a single file or act as data encapsulation; when a file is saved, it is stored under the default package or the package name specified at the start of the file. Package names and directory structure are inextricably linked. If a package's name is college.student.cse, for example, there will be three directories: college, student, and cse. As a result, cse is present in the student and the student is present at college.

```
college
     +student
          +cse
```

The goal is to ensure that files in directories are simple to find when using the packages.

Domain name package naming standards are in reverse order, as in org.peeksforpeeks.practice, org.peeksforpeeks.contribute.

ADDING PACKAGE MEMBERS

A package can have any number of members, including classes, subclasses, traits, objects containing the primary method, and subpackages. Unlike Java packages, we may declare packages in separate scala files, which means that different scala files can create for the same package.

Example:

```
// the file named as faculty.scala
package college
class faculty{
   def faculymethod(){}
}


// the file named as student.scala
// containing main method
```

```
// using college package name again
package college
class student
{
    def studentmethod(){}
}

// Creation of object
object Main
{

    // the main method
    def main(args: Array[String])
    {
        val stu= new student()
        val fac= new faculty()
        // faculty class can access while
        // in the different file but in the same
package.
    }
}
```

USING PACKAGES

Packages can utilize in a variety of ways within a program. Import clauses in Scala are more versatile than in Java. Import clauses, for instance, can be used anywhere in the program as an independent statement by using the keyword import, something Java does not allow.

```
// base.scala
// bb directory
package bb

// creation of a class
class peek
{
    private var id=0
    def method()
    {
        println("welcome to scala class")
        println("id="+id)
    }
}
```

The following is an instance of a Package utilizing import clauses.

```
// main.scala
// aa directory
package aa

// Creation of object
object Main
{
    // the main method
    def main(args: Array[String])
    {
        // importing in the main method
        import bb.peek

        // using member injected using the import
statement
        val obj = new peek()
        obj.method();
    }
}
```

PACKAGE OBJECTS IN SCALA

The primary goal of a package is to make files modularized and easy to manage. So we maintain project files in distinct folders or directories based on the namespace defined, but there are instances when we want specific variables, definitions, classes, or objects to be available to the entire package. However, definitions, data variables, and type aliases cannot store directly in a package. To do this, we have package objects, which allow the common data to reside at the top level of the package. Scala version 2.8 added package objects.

Syntax:

```
package projectx
package src
package main
object 'package'
{
  // using the backticks
  val m
  // members
}
```

Example:

```
package language.PFP

object PFP
{
val scala = "scala"
val java = "java"
val csharp = "csharp"
}

object demo
{
def main( args : Array[String])
{
    println(PFP.scala)
    println(PFP.csharp)
    var totalfees= tax + fees
    println(totalfees)
}
}
```

Objects in this package are as follows:

- They have the ability to extend other classes and/or mixin traits.

```
package object main extends demovars
{
  val m = a // from demovars
  val n = b // from demovars
  // members
}
```

- Instead of specifying implicit in companion objects, they are an excellent way to keep them.
- We may reduce the number of imports in our code by using package objects.

```
package object main
{
  val m = demovars.a // from demovars
  val n = demovars.b // from demovars
  // members
}
```

- Only one package object with the associated directory directory/package name is permitted for each package.

```
+src
   +eatables
        +food.scala
        +package.scala // package object for the
eatables folder
  +drinkables
        +drinks.scala
        +package.scala // package object for the
drinkable folder
```

SCALA CHAINED PACKAGE CLAUSES

Chained packages are a method of resolving the visibility of package members. According to Martin Odersky, this was added in Scala 2.8. Assume we have the following code.

Let's crack the code and figure out what's going on here.

```
package x.z
object m {
  n  //object n
}
object b{
  m  //object m
}
```

Alternatively, we may write the above code as follows.

```
// but this is not good and short way of writing the
package clauses
// let's just stick to first style.
package x{
    package z{
       object m {
       }
       object n{
       }
   }
 }
```

Objects m and n are specified in a directory z, which is contained within a directory x. Members m and n of package z are visible, but not members x. We may alternatively put these objects in various Scala files, as given below:

```
// m.scala
package x.z
object m {
  n //object n still visible
}

// n.scala
package x.z
object n{
  m // object m still visible
}

// files would create like this
+x
  +z
    +m.scala
    +n.scala
```

SCALA FILE HANDLING

File Handling is a method of storing the retrieved data in a file. Scala includes packages allowing us to create, read, and write files. We borrow java.io._ from Java to write to a file in Scala since the Scala standard library lacks a class. We might also include java.io.File and java.io.PrintWriter in our imports.

Creating a new file:

- java.io is generating a new file. The file provides classes and interfaces that allow the JVM to access files, file systems, and attributes.
- File(String pathname) translates the supplied string to an abstract path name, generating a new file object.

Writing the file:

- Writing to the file. java.io PrintWriter contains all of the printing techniques found in PrintStream.

The implementation for making a new file and writing into it is shown below:

```
// program of File handling
import java.io.File
import java.io.PrintWriter

// Creation of object
object Peeks
{
    // the main method
    def main(args:Array[String])
    {
        // Creation of a file
        val file_Object = new File("xyz.txt" )

        // Passing the reference of file to
printwriter
        val print_Writer = new
PrintWriter(file_Object)

        // Writing to file
        print_Writer.write("Hello, This is Peeks For
Peeks")

        // Closing printwriter
        print_Writer.close()
}
}
```

The string "Hello, This is Peeks For Peeks" is written to the text file abc.txt.

Scala does not provide a class for writing files, but it does give a class for reading files. This is the Source class. To read files, we utilize its partner object. To read the contents of this file, we use the fromFile() function of the Source class, which takes the filename as an argument.

The implementation for reading every character from a file is shown below:

```
// program of File handling
import scala.io.Source
```

```
// Creation of object
object PeeksScala
{
    // the main method
    def main(args : Array[String])
    {
        // file name
        val fname = "xyz.txt"

        // creates the iterable representation
        // of source file
        val fSource = Source.fromFile(fname)
        while (fSource.hasNext)
        {
            println(fSource.next)
        }

        // closing the file
        fSource.close()
    }
}
```

We may use the getLines() function to read specific lines rather than the entire file at once.

The implementation for reading each line from a file is shown below:

```
// file handling program to Read each
// line from the single file
import scala.io.Source

// Creation of object
object pfpScala
{
    // the main method
    def main(args:Array[String])
    {
        val fname = "xyz.txt"
        val fSource = Source.fromFile(fname)
        for(line<-fSource.getLines)
        {
            println(line)
        }
```

```
        fSource.close()
    }
}
```

This chapter covered String Interpolation, StringContext, Regular Expressions, StringBuilder, and concatenation. Moreover, we discussed Packages and File Handling.

CHAPTER 6

Scala Methods

IN THIS CHAPTER

- String
- String Interpolation
- StringContext
- Regular Expressions
- StringBuilder and concatenation
- Packages
- File Handling

In the previous chapter, we covered Scala strings and packages, and in this chapter, we will discuss Scala Methods.

SCALA FUNCTIONS – BASICS

A function is a group of statements that execute a certain activity. The code can be divided into separate functions, understanding that each function must do a certain purpose. Functions simplify some common and repetitive tasks into a single function so that instead of rewriting the same code for different inputs, we can just call the function. Scala is thought to be a functional programming language, hence they are significant. It makes debugging and modifying the code easy. First-class values are Scala functions.

DOI: 10.1201/9781003358527-6

The distinction between Scala functions and methods is that a function is an object that may be put in a variable. A method, on the other hand, always belongs to a class with a name, signature bytecode, and so on. A method is essentially a function that is a member of some object.

DECLARATION AND DEFINITION OF FUNCTIONS

In theory, function declaration and definition consist of six components:

- def keyword: In Scala, the "def" keyword is used to declare a function.
- functionname: It must be a legitimate name in lower case. In Scala, function names can contain characters like as +, ~, &, –, ++, \, /, and so on.
- parameterlist: In Scala, a comma-separated list of input parameters, followed by their data type, is declared within the enclosing parentheses.
- returntype: When defining a function, the user must provide the arguments' return type, and the function's return type is optional. If no return type is specified for a function, the default return type is Unit, which is equivalent to void in Java.
- =: In Scala, a function can create with or without the = (equal) operator. The function will return the desired value if the user invokes it. If they do not utilize it, the function will not return any value and will operate in the same manner as a subroutine.
- Method body: The method body is surrounded by braces {}. The code must be performed in order to carry out our planned tasks.

Syntax:

```
def functionname ([parameterlist]) : [returntype]
= {

    // function body

  }
```

Note: The method is declared abstract implicitly if the user does not use the equals sign and body.

CALLING A FUNCTION

In Scala, there are primarily two methods for calling the function. The first method is the standard method, which is as follows:

```
functionname(paramterlist)
```

In the second method, a user can invoke the function using the instance and dot notation as follows:

```
[instance].functionname(paramterlist)
```

Example:

```
object PeeksforPeeks {

    def main(args: Array[String]) {

        // Calling function
        println("The Sum is: " +
functionToAdd(7,4));
    }


    // declaration and definition of the function
    def functionToAdd(m:Int, n:Int) : Int =
    {

        var sum:Int = 0
        sum = m + n

        // returning value of sum
        return sum
    }
    }
```

EXAMPLES OF CURRYING FUNCTIONS IN SCALA

Currying is essentially a technique or procedure for modifying a function in Scala. This function converts a function with several parameters into a function with a single argument. It is commonly used in a variety of functional languages.

Syntax:

```
def functionname(argument1, argument2) = operation
```

Example:

```
// program add two numbers
// using the currying Function
object Curry
{
    // Define the currying function
    def add(m: Int, n: Int) = m + n;

    def main(args: Array[String])
    {
        println(add(22, 17));
    }
}
```

Here, we defined the add function, which accepts two inputs (m and n) and simply adds m and n and returns the result, which we call the main function.

Another Method for Declaring a Currying Function

Assume we need to convert this add function into a Curried function, which means we need to transform a function that takes two(multiple) parameters into a function that takes one(single) argument.

Syntax:

```
def functionname(argument1) = (argument2) =>
operation
```

Example:

```
// program add two numbers
// using the Currying function

object Curry
{
    // transforming function that
    // takes two(multiple) arguments into
```

```
    // function that takes one(single) argument.
    def add2(m: Int) = (n: Int) => m + n;

    // the main method
    def main(args: Array[String])
    {
        println(add2(22)(17));
    }
}
```

Here, we defined the add2 function, which accepts only one input m and returns a second function with the value of add2. The second function will likewise accept one argument, say n, and when called in main, it will take two parenthesis(add2()()), where the first is of the function add2 and the second is of the second function. It will yield the product of two numbers, m+n. As a consequence, we curried the add function, which means we converted the function that accepts two arguments into a function that takes one input and returns the outcome.

Partial Application Currying Function

The Partially Applied function is another method to use the Curried function. So, to understand it, let's look at a basic example. In the main function, we defined a variable sum.

Example:

```
// program add two numbers
// using the Currying function
object Curry
{
    def add2(m: Int) = (n: Int) => m + n;

    // the main function
    def main(args: Array[String])
    {
        // Partially Applied function.
        val sum = add2(27);
        println(sum(7));
    }
}
```

When assigning the function to the value, just one parameter is provided. The value is supplied as a second parameter, and the result is displayed when these arguments are added.

The following is another method to construct the currying function (syntax).

Syntax:

```
def functionname(argument1) (argument2) =
operation
```

Example:

```
// program add two numbers
// using the Currying function
object Curry
{
    // Curring the function declaration
    def add2(m: Int) (n: Int) = m + n;

    def main(args: Array[String])
    {
        println(add2(27)(7));
    }
}
```

The Partial Application method also changes for this syntax.

Example:

```
// program add two numbers
// using the Currying function
object Curry
{
    // Curring the function declaration
    def add2(m: Int) (n: Int) = m + n;

    def main(args: Array[String])
    {
    // Partially Applied function.
        val sum=add2(27)_;
        println(sum(7));
    }
}
```

SCALA ANONYMOUS FUNCTIONS

Scala has A function literal is another name for an anonymous function. An anonymous function is one which doesn't have a name. An anonymous function provides a lightweight function definition. It comes in handy when we need to write an inline function.

Syntax:

```
(n:Int, m:Int)=> n*m
Or
(_:Int)*(_Int)
```

In the first syntax, => is referred to as a transformer. The transformer changes the left-hand side of the symbol's parameter list into a new result using the expression on the right-hand side.

The _character in the above second syntax is known as a wildcard and is a shorthand approach to express a parameter that appears just once in the anonymous function.

ANONYMOUS PARAMETERIZED FUNCTIONS

A function value is created when a function literal is instantiated in an object. In other words, when we assign an anonymous function to a variable, we may use that variable as a function call. In the anonymous function, we may declare numerous parameters.

Example 1:

```
// program to illustrate an anonymous method
object Main
{
    def main(args: Array[String])
    {

        // Creation of anonymous functions
        // with the multiple parameters Assign
        // anonymous functions to the variables
        var myfc1 = (str1:String, str2:String)
=> str1 + str2

        // Anonymous function is created
        // using _ wildcard instead of
```

```
        // the variable name because str1 and
        // str2 variable appear only once
        var myfc2 = (_:String) + (_:String)

        // Here, variable invoke like a function
call
        println(myfc1("Peeks", "12Peeks"))
        println(myfc2("Peeks", "forPeeks"))
    }
}
```

Without parameters, anonymous functions

We can define an anonymous function with no parameters. We can provide an anonymous function as a parameter to another function in Scala.

Example 2:

```
// program to illustrate the anonymous method
object Main
{
    def main(args: Array[String])
    {

        // Creation of anonymous functions
        // without the parameter
        var myfun1 = () => {"Welcome to
PeeksforPeeks...!"}
        println(myfun1())

        // Function which contains anonymous
        // function as parameter
        def myfunction(fun:(String, String)=>
String) =
        {
            fun("Dog", "Cat")
        }

        // Explicit type declaration of anonymous
        // function in the another function
        val f1 = myfunction((str1: String,
                    str2: String) => str1 + str2)

        // Shorthand declaration using the wildcard
        val f2 = myfunction(_ + _)
```

```
        println(f1)
        println(f2)
    }
}
```

SCALA HIGHER ORDER FUNCTIONS

A function is referred to be a Higher Order Function if it takes other functions as parameters or returns another function as an output; that is, functions that operate on other functions are referred to as Higher Order Functions. It is important to note that this higher order function applies to functions and methods that take functions as parameters or return a function as a result. This is possible because the Scala compiler allows us to coerce methods into functions.

Some key items to remember regarding higher order functions:

- Higher order functions are conceivable because the Scala programming language considers functions as first-class values, which means that functions, like other values, may be supplied as parameters or returned as output, which is useful in providing an adaptable approach for constructing codes.
- It is useful in constructing function compositions, in which functions may create from other functions. Function composition is a form of composing in which a function demonstrates the use of two composed functions.
- It's also useful for writing lambda functions and anonymous functions. Anonymous functions are functions that do not have a name yet execute functions.
- It is even used to reduce the number of unnecessary lines of code in a program.

Example:

```
// program of higher order function

// Creation of object
object PfP
{

    // the main method
    def main(args: Array[String])
```

```
    {
        // Display the output by assigning
        // the value and calling functions
        println(apply(format, 34))

    }

    // higher order function
    def apply(m: Double => String, n: Double)
= m(n)

    // Defining the function for
    // format and using a
    // method toString()
    def format[R](z: R) = "{" + z.toString() + "}"

}
```

The apply function in this case contains another function m with the value n and performs the function m to n.

Example:

```
// program of higher order function

// Creation of object
object PfP
{
    // the main method
    def main(args: Array[String])
    {

        // Creation of a list of numbers
        val num = List(5, 6, 7)

        // Defining the method
        def multiplyValue = (n: Int) => n * 4

        // Creation of a higher order function
        // that is assigned to variable
        val result = num.map(n =>
multiplyValue(n))
```

```
        // Displays output
        println("Multiplied List is: " + result)
    }
}
```

Here, map is a higher order function that accepts another function as an argument, namely (n => multiplyValue(n)).

NAMED ARGUMENTS IN SCALA

When arguments pass through a function with named parameters in Scala, we may label the arguments with the parameter names. These named arguments are cross-referenced with the function's named parameters. Unnamed Parameters normally utilize Parameter Positions to call Functions or Constructors; however, these named parameters enable us to modify the order of the arguments passed to a function by simply swapping the order.

Syntax:

```
Function Definition : def createArray(length:int,
capacity:int);
Function calling : createArray(capacity=30,
length:20);
```

Precautions:

```
-> If some arguments are named, and others are
not, unnamed arguments come first
function(0, b = "1")

-> the Order Interchange is valid
function(b = "1", a = 0)

-> Not accepted, error: positional after the named
argument
function(b = "1", 0)

-> Not accepted, parameter 'a' specified twice as
'0' in the first position and again as a = 1
function(0, a = 1)
```

Note: If the m argument expression is of the form m = expr and m is not a method parameter name, the argument is handled as an assignment expression to some variable m.

Example:

```
// program using the Named arguments

// Creation of object
object PFP
{
    // the main method
    def main(args: Array[String])
    {
        // passed with the named arguments
        printIntiger(X = 8, Y = 9);
    }

    // Defining the method
    def printIntiger( X:Int, Y:Int ) =
    {
        println("The Value of X : " + X );
        println("The Value of Y : " + Y );
    }
}
```

In the above instance, we built the printIntiger function and then called it. When invoking the function, we utilize the names of the function parameters. We passed arguments X = 8 and Y = 9, where X and Y are parameter names.

Example:

```
// program using the Named arguments

// Creation of object
object PFP
{

    // the main method
    def main(args: Array[String])
```

```
    {
        // without the named arguments
        printName("peeks","for","peeks");

        // passed arguments according to order
        printName(first = "Peeks", middle="for",
                                last = "Peeks");

        // passed arguments with different order
        printName(last = "Peeks", first = "Peeks",
                                middle="for");
    }

    // Defining the function
    def printName( first: String, middle: String,
                                last: String ) =
    {
        println("Ist part of the name: " + first )
        println("IInd part of the name: " + middle )
        println("IIIrd part of the name: " + last )
    }
}
```

As seen in the preceding example, we constructed the printName function and then invoked it. When invoking the function, we utilize the names of the function parameters. We may get the same result by modifying the order of the inputs, such as printName(last = "Peeks," first = "Peeks," middle= "for").

FUNCTIONS CALL-BY-NAME IN SCALA

When arguments pass through a call-by-value function in Scala, the value of the passed-in expression or arguments is computed once before invoking the function. A call-by-Name function in Scala, on the other hand, calls the expression and recomputes the value of the passed-in expression every time it is retrieved within the function. Here are some examples of differences and syntax.

Call-by-Value

This approach makes advantage of in-mode semantics. Changes made to formal parameters are not returned to the caller. Any changes made to the formal parameter variable within the called function or method impact just the separate storage location and do not affect the real parameter in the calling environment. This technique is also known as a call-by-value method.

Syntax:

```
def callByValue(m: Int)


// program of function call-by-value

// Creation of object
object PFP
{
    // the main method
    def main(args: Array[String])
    {
        // Defined the function
        def ArticleCounts(i: Int)
        {
            println("Shreya did article " +
                    "on day one is 1 - Total = "
+ i)
            println("Shreya did article " +
                    "on day two is 1 - Total = "
+ i)
            println("Shreya did article "+
                    "on day three is 1 - Total = "
+ i)
            println("Shreya did article " +
                    "on day four is 1 - Total = "
+ i)
        }

        var Total = 0;

        // the function call
        ArticleCounts
```

```
        {
            Total += 1 ; Total

        }
    }
}
```

In this case, the total articles are not raised using the function call-by-value method in the preceding program.

Call-by-Name

A call-by-name mechanism sends a code block to the function call, which compiles, executes, and calculates the value. The message will be written first, followed by the value.

Syntax:

```
def callByName(m: => Int)
```

Example:

```
// program of the function call-by-name

// Creation of object
object main
{
    // the main method
    def main(args: Array[String])
    {
        // Defined the function call-by-name
        def ArticleCounts(i: => Int)
        {
            println("Shreya did articles " +
                    " on day one is 1 - Total = "
+ i)
            println("Shreya did articles "+
                    "on day two is 1 - Total = "
+ i)
            println("Shreya did articles " +
                    "on day three is 1 - Total = "
+ i)
            println("Shreya did articles " +
```

```
                    "on day four is 1 - Total = "
+ i)
        }

        var Total = 0;

        // calling the function
        ArticleCounts
        {
            Total += 1 ; Total
        }
}
}
```

The total number of articles will enhance in this case by employing the function call-by-name approach in the preceding program.

Another function-by-name program.

Example:

```
// program of the function call-by-name

// Creation of object
object PFP
{
    // the main method
    def main(args: Array[String])
    {
        def something() =
        {
            println("calling the something")
            1 // return value
        }

        // Defined the function
        def callByName(m: => Int) =
        {
            println("m1=" + m)
            println("m2=" + m)
        }

        // Calling the function
        callByName(something)
    }
}
```

CLOSURES IN SCALA

Scala Closures are functions that employ one or more free variables and whose return value is determined by these variables. The free variables are defined independently of the Closure Function and are not sent to it as arguments. The free variable distinguishes a closure function from a conventional function. A free variable is any variable that is not specified within the function and is not supplied as a function argument. A free variable does not have a valid value since it is not tied to a function. There are no values for the free variable in the function.

As an example, consider the following function:

```
def example(m:double) = m*p / 100
```

When we run the above code, we get the error beginning not found p. So now we give p a value outside of the function.

```
// defined value of p as 20
val p = 20

// define this closure.
def example(m:double) = m*p / 100
```

The above code may now execute since the free variable has a value. If we execute the functions as follows:

```
Calling function: example(20000)
Input: p = 20
Output: double = 2000.0
```

When the value of the free variable changes, how would the value of the closure function change?

So, in essence, the closure function takes the most recent state of the free variable and appropriately modifies the closure function's value.

```
Input: p = 20
Output: double = 2000.0

Input: p = 30
Output: double = 3000.0
```

Depending on the kind of free variable, a closure function can further characterize as pure or impure. If we give the free variable the type var, it will alter its value at any moment during the code, potentially modifying the value of the closure function. As a result, this closure is an impure function. On the other hand, if we define the free variable of type val, the variable's value remains constant, resulting in a pure closure function.

Example:

```
// Addition of the two numbers with Scala closure

// Creation of object
object PFP
{
    // the main method
    def main(args: Array[String])
    {
        println( "Final_Sum(1) value is = "
+ sum(1))
        println( "Final_Sum(2) value is= "
+ sum(2))
        println( "Final_Sum(3) value is= "
+ sum(3))
    }

    var a = 6

    // define the closure function
    val sum = (b:Int) => b + a
}
```

In the above code, the sum is a closure function. var a = 6 represents impure closure. The value of a is the same, but the value of b is different.

Example:

```
// closure program to print the string

// Creation of object
object PFP
{
    // the main method
```

```
    def main(args: Array[String])
    {

        var employee = 70

        // define the closure function
        val gfg = (name: String) => println
(s"The Company name is $name"+
                    s" and total no. of the
employees are $employee")

        gfg("peeksforpeeks")
    }
}
```

In the above example, pfp is a closure. The employee is a mutable variable that may change.

NESTED FUNCTIONS IN SCALA

A Nested Function is a function definition that is contained within another function. C++, Java, and other programming languages do not support it. We can call a function within a function in other languages, but it is not a nested function. Scala allows us to construct functions within functions, and functions defined within other functions are referred to as nested or local functions.

Syntax:

```
def FunctionName1( parameter_1, parameter_2, ...)
= {
    def FunctionName2() = {
    // code
    }
    }
```

SINGLE NESTED FUNCTION

Here's an instance of a single nested function that accepts two values as inputs and returns the Maximum and Minimum of those numbers.

Example:

```
// program of the Single Nested Function
object MaxAndMin
```

```
{
    // the main method
    def main(args: Array[String])
    {
        println("The Min and Max from 7, 9")
                maxAndMin(7, 9);
    }

    // Function
    def maxAndMin(m: Int, n: Int) = {

    // the Nested Function
    def maxValue() = {
        if(m > n)
        {
            println("Max is: " + m)
        }
        else
        {
            println("Max is: " + n)
        }
    }

    // the Nested Function
    def minValue() = {
        if (m < n)
        {
            println("Min is: " + m)
        }
        else
        {
            println("Min is: " + n)
        }
    }
    maxValue();
    minValue();
    }
}
```

In the above program, maxAndMin is a function, and maxValue is another inner function that returns the largest value between m and n. Similarly, minValue is another inner function that is likewise nested, and it returns the least value between m and n.

MULTIPLE NESTED FUNCTION

Here's an example of a multiple-nested function.

Example:

```
// program of the Multiple Nested Function
object MaxAndMin
{
    // the main method
    def main(args: Array[String])
    {
        fun();
    }

    // Function
    def fun() = {

        peeks();

        // The First Nested Function
        def peeks() = {
            println("peeks");

            pfp();

            // The Second Nested Function
            def pfp() = {
                println("pfp");

                peeksforpeeks();

                // The Third Nested Function
                def peeksforpeeks() = {
                    println("peeksforpeeks");
                }
            }
        }
    }
}
```

In the preceding code, fun is a function, and peeks, pfp, and peeksforpeeks are nested or local functions.

SCALA PARAMETERLESS METHOD

A parameterless method does not accept parameters and is distinguished by the absence of any empty parenthesis. A parameterless function should be invoked without parentheses. This allows for the change of def to val without requiring any changes to the client code, which is part of the uniform access concept.

Example:

```
// program to illustrate Parameterless method
invocation
class PeeksforPeeks(name: String, ar: Int)
{
    // parameterless method
    def author = println(name)
    def article = println(ar)

    // empty-parenthesis method
    def printInformation() =
    {
        println("User -> " + name + ", Articles ->
" + ar)
    }
}

// Creation of object
object Main
{
    // the main method
    def main(args: Array[String])
    {

        // Creation of object of Class
'Peeksforpeeks'
        val PFP = new PeeksforPeeks("Disha", 60)
        PFP.author   // calling the method without
parenthesis
    }
}
```

There are two main rules for utilizing the parameterless procedure. The first is when there are no parameters. The second is when the procedure

does not alter the changeable state. By designing methods using parentheses, one may prevent invocations of parameterless methods that seem like field selections.

Calling the parameterless method with parenthesis results in a Compilation error.

Example:

```
// program to illustrate Parameterless method
invocation
class PeeksforPeeks(name: String, ar: Int)
{
    // parameterless method
    def author = println(name)
    def article = println(ar)

    // empty-parenthesis method
    def printInformation() =
    {
        println("User -> " + name + ", Articles ->
" + ar)
    }
}

// Creation of object
object Main
{
    // the main method
    def main(args: Array[String])
    {

        // Creation of object of Class
'Peeksforpeeks'
        val PFP = new GeeksforGeeks("Disha", 60)
        PFP.author()     //calling the method
without parenthesis
    }
}
```

Note: It is able to reach an empty parenthesis method without parenthesis, although it is usually encouraged and recognized as a convention to call empty-parenthesis methods with parenthesis.

SCALA RECURSION

The Recursion is a method that divides the problem into smaller subproblems and calls itself for each of them. That is function calling itself. Instead of loops, we may employ recursion. Recursion avoids the changeable state associated with loops. Recursion is fairly prevalent in functional programming and gives a simple technique to explain many algorithms. In functional programming, recursion is seen as significant. Scala has strong support for recursion.

Let me explain with a simple factorial example.

Example:

```
// program of the factorial using recursion

// Creation of object
object PFP
{
    // the Function define
    def fact(n:Int): Int=
    {
        if(n == 1) 1
        else n * fact(n - 1)
    }

    // the main method
    def main(args:Array[String])
    {
        println(fact(4))
    }
}
```

The preceding code demonstrates a recursive approach to a factorial function, where the condition n == 1 causes the recursion to break.

Let us clarify by using gcd as an example.

Example:

```
// program of GCD using recursion

// Creation object
object PFP
```

```
{
    // the Function defined
    def gcd(x:Int, y:Int): Int=
    {
        if (y == 0) x
        else gcd(y, x % y)
    }

    // the main method
    def main(args:Array[String])
    {
        println(gcd(14, 17))
    }
}
```

The problem with recursion is that deep recursion might blow up the stack if we are not careful.

Let me illustrate this using an example:

Example:

```
// program of sum all numbers using recursion

// Creation of object
object PFP
{
    // the Function defined
    def sum(num: Int): Int=
    {
        if (num == 1)
            1
        else
            sum(num - 1) + num
    }

    // the main method
    def main(args:Array[String])
    {
        println(sum(66))
    }
}
```

The procedure sum will add all of the numbers together. Every time, we lower the number and add it to the result. Every time we use sum, it will leave the input value num on the stack, using memory. When we try to pass a huge input, such as sum(666666), the result is java.lang. StackOverflowError. This indicates that the stack has blown up.

Because the above example does not employ tail recursion, it is not an optimum strategy, especially if the beginning number n is very big.

TAIL RECURSION

Tail-recursive functions are preferred over non-tail-recursive functions because tail-recursion can optimize by the compiler. If the recursive call is the last thing done by the function, it is said to be tail recursive. It is not necessary to preserve a record of the prior condition.

Consider the following example:

```
// program of factorial using the tail recursion
import scala.annotation.tailrec

// Creation of object
object PFP
{
    // Function definition
    def factorial(n: Int): Int =
    {
        // Using the tail recursion
        @tailrec def factorialAcc(acc: Int, n: Int):
Int =
        {
            if (n <= 1)
                acc
            else
                factorialAcc(n * acc, n - 1)
        }
        factorialAcc(1, n)
    }

    // the main method
    def main(args:Array[String])
    {
        println(factorial(6))
    }
}
```

We may use the @tailrec annotation in the preceding code to ensure that our algorithm is tail recursive.

The compiler will complain if we use this annotation and our approach is not tail recursive. For example, if we try to apply this annotation on the factorial method shown above, we will get the below compile-time error.

Example:

```
// program of factorial with the tail recursion
import scala.annotation.tailrec

// Creation of object
object PFP
{
    // Function definition
    @tailrec def factorial(n: Int): Int =
    {
        if (n == 1)
            1
        else
            n * factorial(n - 1)
    }

    // the main method
    def main(args:Array[String])
    {
        println(factorial(6))
    }
}
```

```
Couldn't optimize @tailrec annotated method
factorial: it contains recursive call, not in the
tail position
```

SCALA TAIL RECURSION

Recursion is a method that divides a problem into smaller subproblems and then calls itself for each of them. That is, it merely indicates that the function calls itself. Because the compiler can optimize tail-recursion, tail-recursive functions perform better than non-tail-recursive functions. If the recursive call is the last thing done by the function, it is said to be tail recursive. It is not necessary to preserve a record of the prior condition.

In the code, a package import scala.annotation.tailrec will be utilized for the tail recursion function.

Syntax:

```
@tailrec
def FuntionName(Parameter_1, Parameter_2, ...):
type = …
```

Example:

```
// program of GCD using the recursion
import scala.annotation.tailrec

// Creation of object
object PFP
{
    // Function defined
    def GCD(n: Int, m: Int): Int =
    {
        // the tail recursion function defined
        @tailrec def gcd(x:Int, y:Int): Int=
        {
            if (y == 0) x
            else gcd(y, x % y)
        }
        gcd(n, m)
    }

    // the main method
    def main(args:Array[String])
    {
        println(GCD(14, 17))
    }
}
```

As seen in the above code, @tailrec is used for the gcd function, which is a tail recursion function. There is no need to retain track of the prior state of the gcd function while employing tail recursion.

Example:

```
// program of factorial using the tail recursion
import scala.annotation.tailrec
```

```
// Creation of object
object PFP
{
    // Function definition
    def factorial(n: Int): Int =
    {
        // Using the tail recursion
        @tailrec def factorialAcc(acc: Int, n:
Int): Int =
        {
            if (n <= 1)
                acc
            else
                factorialAcc(n * acc, n - 1)
        }
        factorialAcc(1, n)
    }

    // the main method
    def main(args:Array[String])
    {
        println(factorial(6))
    }
}
```

The @tailrec annotation in the preceding code confirms that our approach is tail recursive.

The compiler will complain if we use this annotation and our approach is not tail recursive. For example, if we try to apply this annotation on the factorial method shown above, we will get the following compile-time error.

Example:

```
// program of factorial with the tail recursion
import scala.annotation.tailrec

// Creation of object
object PFP
{
    // Function definition
    @tailrec def factorial(n: Int): Int =
```

```
    {
        if (n == 1)
            1
        else
            n * factorial(n - 1)
    }

    // the main method
    def main(args:Array[String])
    {
        println(factorial(6))
    }
}
```

PARTIALLY APPLIED FUNCTIONS IN SCALA

Partially applied functions are not applied to all of the arguments defined by the given function; for example, while calling a function, we can offer some of the parameters and the left arguments are supplied as needed. When we call a function, we may send less parameters to it, and it does not raise an exception when we do. We utilize hyphen (_) as a placeholder for parameters that are not given to the function.

Here are some key points:

- Partially applied functions are useful for reducing a function with many arguments to a function with only a few arguments.
- It may use to replace any number of function parameters.
- This strategy is more convenient for users to use.

Syntax:

```
val multiply = (x: Int, y: Int, z: Int) => x * y * z

// the less arguments passed
val k = multiply(1, 2, _: Int)
```

As we can see from the syntax above, we built a standard function multiply with three parameters, but we passed less arguments (two). As we can see, it does not throw an exception when the function is just partially applied.

Example:

```
// program of Partially applied functions

// Creation of object
object Applied
{

    // the main method
    def main(args: Array[String])
    {

        // Creation of a Partially applied function
        def Book(discount: Double, costprice:
Double)
        : Double =
        {

            (1 - discount/100) * costprice

        }

        // Applying the only one argument
        val discountedPrice = Book(27, _: Double)

        // Displays discounted price of the book
        println(discountedPrice(500))

    }
}
```

The discount is passed in argument, and the cost price part is left empty so that it may be passed later as required, allowing the decreased price to be computed indefinitely.

Here are a few more instances of partially applied functions:

1. Even if no of the defined parameters are used, a partially applied function can achieve.

   Example:

   ```
   // program of Partially applied functions

   // Creation of object
   object Applied
   {
   ```

```
    // the main method
    def main(args: Array[String])
    {

        // Creation of a Partially applied
function
        def Mul(x: Double, y: Double)
        : Double =
        {
            x * y
        }

        // Not applying any argument
        val r = Mul _

        // Display the Multiplication
        println(r(7, 5))
    }
}
```

2. Any number of arguments can be replaced using partially applied functions.

Example:

```
// program of Partially applied functions

// Creation of object
object Applied
{

    // the main method
    def main(args: Array[String])
    {

        // Creation of the Partially applied
function
        def Mul(c: Double, d: Double, d: Double)
        : Double =
        {
            c * d * e
        }
```

```
        // applying the some arguments
        val r = Mul(8, 7, _ : Double)

        // Display the Multiplication
        println(r(10))
    }
}
```

3. Currying may use in Partially applied functions to divide a function with several arguments into numerous functions, each with only one argument.

   Example:

```
// program of Partially applied functions using
// the Currying approach

// Creation of object
object curr
{

    // the main method
    def main(args: Array[String])
    {

        // Creating a Partially applied
        // function
        def div(x: Double, y: Double)
        : Double =
        {
            x / y
        }

        // applying the currying approach
        val m = (div _).curried

        // Display the division
        println(m(52)(7))
    }
}
```

   The currying strategy divides the provided function into two functions, each with one argument, thus we must send one value to each of the functions to produce the required output.

SCALA METHOD OVERLOADING

Polymorphism is commonly implemented by method overloading. It is the capacity to redefine a function in a variety of ways. Function overloading may implement by declaring two or more functions in the same class with the same name. Scala may differentiate between methods by using various method signatures. Within the same class, methods can have the same name but distinct parameter lists (i.e., the number of parameters, the order of the parameters, and the data types of the parameters).

- The amount and kind of parameters given as arguments distinguish overloaded methods.
- We cannot define more than one method with the same name, Order, and argument type. It would be a compiler mistake.
- When differentiating the overloaded method, the compiler does not consider the return type. However, two methods with the same signature but distinct return types cannot declare. It will generate a compilation error.

Why Do We Require Method Overloading?

If we need to perform the same operation in multiple ways, for example, for distinct inputs. In the following example, we perform the addition operation on several inputs. It is difficult to come up with several relevant labels for a single action.

Different Approaches to Overloading Methods

Method overloading is accomplished by changing:

- The total number of parameters in two methods.
- The data types of method parameters.
- The sequence of method parameters.

By varying the number of parameters

Example:

```
// program to demonstrate the function overloading
by changing the number
```

```
// of the parameters
class PFP
{

    // function 1 with the two parameters
    def fun(p:Int, q:Int)
    {
        var Sum = p + q;
        println("The Sum in function 1 is:" + Sum);
    }

    // function 2 with the three parameters
    def fun(p:Int, q:Int, r:Int)
    {
        var Sum = p + q + r;
        println("The Sum in function 2 is:" + Sum);
    }
}
object Main
{
    // the main function
    def main(args: Array[String])
    {

    // Creation of object of PFP class
    var obj = new PFP();
    obj.fun(7, 9);
    obj.fun(4, 9, 72);
    }
}
```

By modifying the parameter data types:

Example:

```
// program to demonstrate the function overloading
by changing data types
// of parameters
class PFP
{

    // Adding the three integer elements
    def fun(p:Int, q:Int, r:Int)
```

```
    {
        var Sum = p + q + r;
        println("The Sum in function 1 is:"+Sum);
    }

    // Adding the three double elements
    def fun(p:Double, q:Double, r:Double)
    {
        var Sum = p + q + r;
        println("The Sum in function 2 is:"+Sum);
    }
}
object Main
{
    // the main method
    def main(args: Array[String])
    {

    // Creation of object of PFP class
    var obj = new PFP();
    obj.fun(7, 6, 11);
    obj.fun(6.8, 12.01, 44.9);
    }
}
```

By changing the parameters in a different order:

Example:

```
// program to demonstrate the function overloading
by changing
// order of parameters
class PFP
{

    // Function1
    def fun(name:String, No:Int)
    {
        println("The Name of the watch company:"
+ name);
        println("The Total number of watch :" + No);
    }
```

```
    // Function2
    def fun(No:Int, name:String )
    {
        println("The Name of the watch company:" +
name);
        println("The Total number of watch :" +
No);
    }
}
object Main
{
    // the main Function
    def main(args: Array[String])
    {

    // Creation of object of PFP class
    var obj = new PFP();
    obj.fun("Rolex", 20);
    obj.fun("Omega", 20);
    }
}
```

What Happens When the Method Signature and Return Type Are the Same?

The compiler will generate error because the return value alone is insufficient for the compiler to determine which function to call. Method overloading is only feasible if both methods have different argument types (and so have separate signature).

Example:

```
// Program to show an error when the method
signature is the same
// and return the type is different.
object Main {

    // the main method
    def main(args: Array[String]) {
println("The Sum in function 1 is:" + fun(7, 6) );
println("The Sum in function 2 is:" + fun(7, 6) );
}
```

```
// function1
def fun(p:Int, q:Int) : Int = {
    var Sum: Int = p + q;
    return Sum;

}

// function2
def fun(p:Int, q:Int) : Double = {
    var Sum: Double = p + q + 4.5;
    return Sum;
}
}
```

SCALA METHOD OVERRIDING

Method overriding in Scala is similar to method overriding in Java; however, the overriding capabilities in Scala are more developed, since both methods as well as var or val may override. Method Overriding occurs when a subclass has a method name that is identical to the method name defined in the parent class. This occurs when subclasses inherited by the declared superClass override the method defined in the superClass using the override keyword.

When Should We Use Method Overriding?

When a subclass intends to provide a specific implementation for a method declared in the parent class, the subclass overrides the parent class's defined method. Method overriding can use to recreate the method specified in the superClass.

Example:

```
// program of the method overriding

// Creation of a super class
class School
{

    // Method definition
    def NumberOfStudents()=
    {
        0 // Utilized for the returning an Integer
    }
}
```

```
// Creation of a subclass
class class1 extends School
{

    // Using the Override keyword
    override def NumberOfStudents()=
    {
        30
    }
}

// Creation of a subclass
class class2 extends School
{

    // Using the override keyword
    override def NumberOfStudents()=
    {
        34
    }
}

// Creation of a subclass
class class3 extends School
{

    // Using override keyword
    override def NumberOfStudents()=
    {
        27
    }
}

// Creation of object
object PfP
{

    // the main method
    def main(args:Array[String])
    {

        // Creation of instances of all
        // the sub-classes
```

```
        var x=new class1()
        var y=new class2()
        var z=new class3()

        // Display the number of students in
class1
        println("The Number of students in class 1
: " +

x.NumberOfStudents())

        // Display the number of students in
class2
        println("The Number of students in class 2
: " +

y.NumberOfStudents())

        // Display the number of students in
class3
        println("The Number of students in class 3
: " +

z.NumberOfStudents())

    }
}
```

Method Overriding Guidelines

There are a few limitations to method overriding, which are as follows:

- One important guideline for method overriding is that the overriding class must use the modifier override or override annotation.
- Auxiliary constructors cannot instantly invoke superClass constructors. They can rarely call the primary constructors, which will call the superClass constructor reverse.
- We will not be allowed to override a var with a def or val in Method Overriding; otherwise, an error will throw.
- We cannot override a val in the superClass with a var or def in the subclass, and if the var in the superClass is abstract, we can override it in the subclass.

- If a field is declared var, it can override the def defined in the superClass. The var can override only a getter or setter combination in the superClass.

Note: Auxiliary Constructors are defined similarly to methods, with the def and this keyword, where this is the constructor's name.

Primary Constructors start at the beginning of the class definition and extend throughout the whole body of the class.

Example:

```
// program of method Overriding

// Creation of a class
class Animal
{

    // Defining the method
    def number()
    {
        println("We have 2 animals")

    }
}

// Extending class Animal
class Dog extends Animal
{

    // using the override keyword
    override def number()
    {

        // Display the output
        println("We have 2 dogs")

    }
}

// Creation of object
object PfP
{
```

```
    // the main method
    def main(args:Array[String])
    {

        // Creation of object of subclass
        // Dog
        var x = new Dog()

        // Calling the overridden method
        x.number()

    }
}
```

We overrode the method here by using the term override. In the above example, the superclass Animal has a method called the number that the subclass Dog overrides. As a result, constructing a subclass object may invoke the overridden method.

Example:

```
// program of method overriding

// Creation of super-class
class Students(var rank:Int, var name:String)
{

    // overriding the method 'toString()'
    override def toString():String =
    {
        " Rank of "+name+" is : "+rank
    }
}

// Creating the subclass of Students
class newStudents(rank:Int, name:String)
        extends Students(rank, name){
}

// Inheriting the main method of
// trait 'App'
object PfP extends App
```

```
{

    // Creating object of a super-class
    val students = new Students(2, "Riya Sood")

    // Display the output
    println(students)

    // Creating object of a subclass
    val newstudents = new newStudents(4, "Paras
Sharma")

    // Display the output
    println(newstudents)

}
```

The superClass constructor, Students, is called from the primary constructor of the subclass, newStudents, therefore the superClass constructor is called using the term extends.

OVERRIDING VS OVERLOADING

In Scala, method overloading is a property that allows us to define methods with the same name. Still, different parameters or data types, whereas method overriding allows us to redefine the method body of the superClass in the subclass with the same name and same parameters or data types in order to change the method's performance.

Method overriding in Scala uses the override modifier to override a method defined in the superClass, whereas method overloading does not require any keyword or modifier; we simply need to change the order of the parameters used, the number of parameters in the method, or the data types of the parameters for method overloading.

WHY IS METHOD OVERRIDING REQUIRED?

Method overriding allows us to redefine a single method in several ways and do distinct actions with the same method name. In the picture above, for example, a superClass School contains a method entitled NumberOfStudents() that is overridden by the sub-classes to do various tasks.

METHOD INVOCATION IN SCALA

Method Invocation is a technique that displays various syntax by dynamically calling methods of a class using an object.

Scala uses the same name standards as Java, which are as follows:

- There should be no space between the invocation object/target and the dot(.), nor should there be any space between the dot and the method name.

```
objt.display("name") // correct
objt.display ("name") // incorrect but legal
objt. display("name") // incorrect but legal
```

- Additionally, there should be no space between the method name and the parenthesis.

```
println("students") // correct
println ("students") // incorrect but legal
```

- A single space and a comma should separate the parameters.

```
objt.display("name", 33) // correct
objt.display ("name", 33) // incorrect but legal
objt. display("name", 33) // incorrect but legal
```

Let's look at few methods with various parameters and styles.

- **Arity-0:** When there are no arguments to pass to the method, the arity is zero. As a result, adding parentheses to methods is not required. It improves code readability, and removing parenthesis reduces the number of characters to some extent.

```
objt.display() //correct
objt.display   //correct
```

- **Arity-1:** When there is just one parameter to pass to the arity-1 method. We can avoid using parentheses around the provided parameter if we utilize this rule for fully functional programming or methods that take functions as parameters. Infix notation is another name for this type of syntax.

  Example:

```
// program for arity 1
// Creation of object
class x
```

```
{
    // Defining the method
    def display(str: String)=
    {
        println(str)
    }
}

// Creation of object
object PfP
{
    // the main method
    def main(args: Array[String])
    {
        val objt = new x
        objt.display("student") // correct
        objt display ("student") // correct
        objt display "student" // correct

    }
}
```

- **Higher order function:** A function is referred to as a Higher Order Function if it contains other functions as parameters or returns another function as an output, that is, functions that operate on other functions are referred to as Higher Order Functions. It is important to note that higher order functions apply to functions and methods that take functions as parameters or return a function as a result. This is possible because the Scala compiler allows us to coerce methods into functions.

  Example:

```
// program of higher order function

// Creation of object
object PfP
{
    // the main method
    def main(args: Array[String])
    {

        // Creating the Set of strings
```

```
        val names = Set("shreya","anusha",
"ritik")

        // Defining the method
        def captainDesignation = (y: String) =>
"captain " + y

        // Creating the higher order function
        // that is assigned to String elements
        val result = names.map(y =>
captainDesignation(y))

        // Display the output
        println("Multiplied List: " + result)
    }
}
```

Important notes:

- When there are no side effects, this syntax should use.
- These conventions are used to increase readability and make the code more understandable.
- It can save space by omitting certain additional characters.

FORMAT AND FORMATTED METHOD IN SCALA

Competitive programming often needs to print the results in a specific format. The printf function in C is well-known to most users. Let's look at how we may format the output in Scala. When a String has both values and text following it, formatting is necessary for updating values and adding text enclosing it.

Example:

```
We have written 23 articles
```

Note:

- String formatting in Scala may be accomplished using two methods: format() and formatted().
- These techniques have been accessed using the StringLike Trait.

Format Method

We may use the Format method to format strings and pass arguments to it, where %d is used for integers and %f is used for floating-points or doubles.

Example:

```
// program of format method

// Creating the object
object PFP
{

    // the main method
    def main(args: Array[String])
    {

        // Creating the format string
        val x = "There are %d books and" +
                    "cost of the each book is %f"

        // Assigning the values
        val y = 17
        val z = 235.87

        // Applying the format method
        val r = x.format(y, z)

        // Display the format string
        println(r)
    }
}
```

Example:

```
// program of format for strings and characters.

// Creating the object
object GPFP
{

    // the main method
    def main(args: Array[String])
```

```
    {

        // Creating the format string
        val x = "Riit%c is a %s."

        // Assigning the values
        val a = 'a'
        val b = "coder"

        // Applying the format method
        val r = x.format(a, b)

        // Display the format string
        println(r)
    }
}
```

Formatted Method

This method may use for any object, including integer, double, and strings.

Example:

```
// program for formatted method

// Creating the object
object PFP
{

    // the main method
    def main(args: Array[String])
    {

        // Assigning the values
        val x = 42

        // Applying the formatted method
        val r = x.formatted("We have written %d
articles.")

        // Display the format string
        println(r)
    }
}
```

SCALA CONTROLLING METHOD SCOPE

Access Modifiers in Scala, as the name implies, assist to limit the scope of a class, variable, function, or data member. Method Scope Control Scala allows us to limit the scope of a method or data member. In Scala, there are five types of controlling method scope:

- Public Scope
- Private Scope
- Protected Scope
- Object-private Scope
- Package Specific

Public Scope

When no access modifier is given for a class, method, or data member, the default access modifier is used.

Data members, classes, or methods not defined with any access modifiers, that is, with the default access modifier, are available from anywhere via package & imports or by creating new instances.

Example:

```
// program of the Public Scope
// package testA
class classA
{
    def method1(): Unit=
    {
        println("method1")
    }
}

// Creating the object
object PfP
{
    // the main method
    def main(args: Array[String])
    {
        // classA in a same package
```

```
        // as main method
        var x = new classA
        x.method1()
    }
}
```

Private Scope

In Java, the private modifier is the same as the private keyword. When a method or variable is marked as private, it is only accessible to the current class and its members and any instances of the same class.

Other objects/classes in the same package will be unable to access the secret members.

This is accomplished by employing the private access modifier.

Example:

```
// program of the Private Scope
// package testA
class classA
{
    var x = 1
    private def method1: Unit =
    {
        println("method1")
    }
}

// Creating the object
object PfP
{
    // the main method
    def main(arg: Array[String])
    {
        var obj1 = new classA
        printf("x = "+obj1.x)
        // println(obj1.method1) error: method
        // method1 in the class classA cannot
        // access in classA
    }
}
```

Protected Scope

Scala protected differs from Java protected. To protect a member, use the term protected before a class or variable.

Only subclasses in the same package have access to protected members.

Example:

```
// program of Protected Scope package test
class classab
{
    protected var ab: Int=4
    var ad: Int =1
}

// Creating the object
object PfP extends classab
{
    // sub-class
    // the main method
    def main(args: Array[String])
    {
        println(ab) //can access
        println(ad) //can access
    }
}
```

Even with imports, protected members cannot be accessed by other members in other packages.

Example:

```
// Scala program of Protected Scope
// package testA
package testA
{
    class classA
    {
        protected var ab: Int=4
        var ad: Int =1
    }
}
```

```
// the another package testB
package testB
{
    // importing all members
    // from the testA package
    import testA._

    // Creating the object
    object PfP
    {
        // the main method
        def main(args: Array[String])
        {
            var ta= new classA
            ta.ad
            ta.ab //error
        }
    }
}
```

Object Private/Protected Scope

- Private is the same as object private. The sole distinction is that a member designated object private will be available exclusively to the object in which it is specified, that is, no other object will be able to access it, thus the term object private.
- The sole distinction between object protected and protected is that the member is only available to the class in which it is defined or to the subclasses and not to the objects.
- Use the terms private[this] to make a member object private.
- Use the phrase protected[this] to protect a member object, where this refers to or references to the current object.

Example:

```
// program of the Object Private/Protected Scope
// package test1.test11
class class11
{
    private[this] var x = 1
    private var t = 2
```

```
    var z = 3
    def method11(other: class11): Unit =
    {
        println(x)
        println(t)
        println(z)

        // println(other.x)
        println(other.t)
        println(other.z)
    }
}
// here on line14 x can only
// access from the inside in which
// it is defined

// Creating the object
object PfP
{
    // the main method
    def main(arg: Array[String])
    {
        var obj11 = new class11() //current
instance creation
        var y = 2
        println(obj11.method11(obj11))
        println(obj11.z)
        //println(obj11.t) //error: t cannot
access
        //println(obj11.x) //error: x is not the
member of the class11
        //according to obj11, x is not the member
    }
}
```

Package Specific

- When we need a member to be available throughout the entire package, it is now time to define that member as private [package_name].
- All members inside the package have access to that member.
- Any other package whose name is qualified to can access Member.

Example:

```
// program of the Package Specific
// program of the Package Specific
package aa
class peek
{
    class g1
    {
        // the inner class
        // private to the class g1
        private var a = 0

        // available to the package aa
        private[aa] var b = 0
        def method()
        {
            a = a + 1
            b = b + 1
            println("welcome to the inner class g1")
            println("a= "+a)
        }
    }
}

// Creating the object
object main
{
    // Driver code
    def main(args: Array[String])
    {
        val obj = new geek()
        val ob = new obj.g1
        ob.method();
        println("b= "+ob.b);
    }
}
```

SCALA REPEATED METHOD PARAMETERS

Repeated Method parameters are supported in Scala, which is useful when we don't know how many arguments a method requires. This Scala feature is used to pass an infinite number of arguments to a specified method.

Important details:

- Each parameter in the procedures with Repeated Parameters should be the same type.
- A Repeated parameter is always the method's final parameter.
- As Repeated Parameters, the method we constructed can only have one parameter.

Example:

```
// program of repeated parameters

// Creating the object
object repeated
{

    // the main method
    def main(args: Array[String])
    {

        // Creating the method with
        // the repeated parameters
        def add(x: Int*)
        : Int =
        {

            // Applying the 'fold' method to
            // perform the binary operation
            x.fold(0)(_+_)

        }

        // Display the Addition
        println(add(4, 5, 7, 9, 2, 11, 14, 15))
    }
}
```

To add any number of additional parameters, include * mark after the kind of parameter being used.

Here are a few more instances of Repeated Parameters:

- In the Repeated Parameter method, an Array can pass.

Example:

```
// program of repeated parameters

// Creating the object
object arr
{

    // the main method
    def main(args: Array[String])
    {

        // Creating the method with
        // repeated parameters
        def mul(x: Int*)
        : Int =
        {

            // Applying 'product' method to
            // perform the multiplication
            x.product

        }

        // Display the product
        println(mul(Array(8, 4, 5, 11): _*))
    }
}
```

To pass an array in the described procedure, we must use a colon, that is, : and _* mark after supplying the array values.

- An illustration of how Repeated Parameters are always the method's final argument.

Example:

```
// program of repeated parameters

// Creating the object
object str
{

    // the main method
    def main(args: Array[String])
```

```
    {

        // Creating the method with
        // the repeated parameters
        def show(x: String, y: Any*) =
        {

            // using 'mkString' method to
            // convert the collection to a
            // string with the separator
            "%s is a %s".format(x,
y.mkString("_"))
        }

        // Display the string
        println(show("PeeksforPeeks",
"Computer",
                              "Sciecne",
"Session"))
    }
}
```

SCALA PARTIAL FUNCTIONS

When a function cannot create a return for every single variable input data provided to it, it is referred to as a partial function. It can only determine the output for a subset of possible inputs. It can only apply in part to the specified inputs.

Here are some key points:

- Partial functions can help us grasp various inconsistencies in Scala routines.
- Case statements can use to interpret it.
- It is a Trait that must be implemented using two methods: isDefinedAt and apply.

Example:

```
// program of Partial function

// Creating the object
object Case
```

```
{

    // the main method
    def main(args: Array[String])
    {

        // Creating the Partial function
        // using a two methods
        val r = new PartialFunction[Int, Int]
        {

            // Applying the isDefinedAt method
            def isDefinedAt(q: Int) = q != 0

            // Applying the apply method
            def apply(q: Int) = 13 * q

        }

        // Display the output if
        // the condition is satisfied
        println(r(12))
    }
}
```

In this part, two methods for applying the Partial function are defined, where isDefinedAt indicates the condition and apply conducts the action if the specified condition is satisfied.

Partial Function Definition Methods

Case statements, collect method, andThen, and orElse are some approaches for defining Partial function.

- **Case statement partial function:** Using the case statement, we will write a Partial function below.

  Example:

```
// program using case statements

// Creating the object
object Case
```

```
{

    // the main method
    def main(args: Array[String])
    {

        // Creating the Partial function
        val d: PartialFunction[Int, Int] =
        {

            // using the case statement
            case k if (k % 4) == 0 => k * 4
        }

        // Display the output if
        // the condition is
        // satisfied
        println(d(4))
    }
}
```

Because the Partial function is built using a case statement, apply, and isDefinedAt are unnecessary.

- **orElse is a partial function:** This approach is useful for connecting Partial functions.

Example:

```
// program using orElse

// Creating the object
object orElse
{

    // the main method
    def main(args: Array[String])
    {

        // Creating the Partial function1
        val M: PartialFunction[Int, Int] =
```

```
        {

            // using the case statement
            case x if (x % 6) == 0 => x * 6
        }

        // Creating the Partial function2
        val m: PartialFunction[Int, Int] =
        {

            // using the case statement
            case y if (y % 3) == 0 => y * 3
        }

        // chaining two partial
        // functions using the orElse
        val r = M orElse m

        // Display the output for
        // which given condition
        // is satisfied
        println(r(6))
        println(r(3))
    }
}
```

- **Partial function using Collect method:** The Collect method asks for a Partial function for each member of the collection and aids in constructing a new collection.

  Example:

```
// program using the collect method

// Creating the object
object Collect
{

    // the main method
    def main(args: Array[String])
    {

        // Creating the Partial function
        val M: PartialFunction[Int, Int] =
```

```
        {

            // using the case statement
            case x if (x % 6) != 0 => x * 6
        }

        // Applying the collect method
        val y = List(6, 14, 8) collect { M }

        // Display the output for which
        // given condition is satisfied
        println(y)
    }
}
```

In this case, Collect will apply the Partial function on all of the List's elements and return a new List based on the conditions specified.

- **andThen is a partial function:** This method adds at the end of chains, which is utilized to go on to further Partial function chains.

  Example:

```
// program using the andThen method

// Creating the object
object andThen
{

    // the main method
    def main(args: Array[String])
    {

        // Creating the Partial function
        val M: PartialFunction[Int, Int] =
        {

            // using the case statement
            case k if (k % 5) != 0 => k * 5
        }

        // Creating the another function
        val append = (k: Int) => k * 10
```

```
            // Applying the andThen method
            val y = M andThen append

            // Display the output after
            // appending another function given
            println(y(8))
        }
    }
```

SCALA LAMBDA EXPRESSION

A Lambda Expression is an expression that substitutes an anonymous function for a variable or value. Lambda expressions are handier when we just need to employ a simple function. These expressions are more expressive and quicker than defining a whole function. We may reuse our lambda expressions for any type of change. It can loop through a set of objects and apply some sort of alteration to them.

Syntax:

```
val lambdaexp = (variable:Type) =>
TransformationExpression
```

Example:

```
// lambda expression to find the double of y
val ex = (y:Int) => y + y
```

Making Use of Lambda Expressions

Lambda expression ca be used as follows:

- A lambda can pass values exactly like any other function call.

Example:

```
// program to show working of lambda expression

// Creating the object
object PfP
{

// the main method
def main(args:Array[String])
```

```
{
    // the lambda expression
    val ex1 = (x:Int) => x + 4

    // with the multiple parameters
    val ex2 = (x:Int, y:Int) => x * y

    println(ex1(8))
    println(ex2(3, 4))
}
}
```

- We often use the map() method to perform transformation to any collection. It is a higher-order function that accepts our lambda expression as an argument and transforms every collection element according to its description.

  Example:

```
// program to apply a transformation on collection

// Creating the object
object GPfP
{

// the main method
def main(args:Array[String])
{
    // list of the numbers
    val l = List(2, 2, 3, 4, 6, 9)

    // squaring each element of a list
    val res = l.map( (y:Int) => y * y )

/* OR
val res = l.map( x=> y * y )
*/
    println(res)
}
}
```

As we can see, the anonymous function built to execute the square operation is not reusable.

- We're using it as an example. We can, however, make it reusable and use it with multiple collections.

  Example:

```
// program to apply transformation on the
collection

// Creating the object
object PfP
{
    // the main method
    def main(args:Array[String])
    {
        // list of the numbers
        val l1 = List(2, 2, 4, 5, 7, 9)
        val l2 = List(14, 22, 38)

        // the reusable lambda
        val func = (x:Int) => x * x

        // squaring each element of a lists
        val res1 = l1.map( func )
        val res2 = l2.map( func )

        println(res1)
        println(res2)
    }
}
```

- A lambda can also be used as a function argument.

  Example:

```
// program to pass lambda as parameter to
function

// Creating the object
object PfP
{

    // transform function with the integer x and
    // function f as a parameter
```

```
    // f accepts Int and returns Double
    def transform( x:Int, f:Int => Double)
    =
    f(x)

    // the main method
    def main(args:Array[String])
    {

        // lambda is passed to f:Int => Double
        val res = transform(4, r => 3.14 * r *
r)

        println(res)
}
}
```

In the above example, the transform function receives an integer x and a function f, and then performs the transformation to x described by f. The argument Lambda in a function call returns a Double type. As a result, argument f must adhere to the lambda specification.

- We can do the same task with any collection. In the case of collections, the only change to the transform function is to use the map function to apply the transformation described by f to each member of the list l.

Example:

```
// program to pass lambda as a parameter to a
function

// Creating the object
object PfP
{

    // transform function with the integer list
l and
    // function f as a parameter
    // f accepts Int and returns Double
    def transform( l:List[Int], f:Int => Double)
    =
    l.map(f)
```

```
    // the main method
    def main(args:Array[String])
    {
        // lambda is passed to f:Int => Double
        val res = transform(List(2, 4, 6), r =>
3.14 * r * r)
        println(res)
    }
}
```

SCALA VARARGS

Scala, like other programming languages, allows us to provide variable length arguments to functions. It enables us to declare that the function's final parameter is of variable length. It may be repeated several times. It allows us to indicate that the function's last parameter is of variable length and may be repeated several times. We are free to make as many arguments as we wish. This enables programmers to call the function with variable-length parameter lists. The type of args stated inside the method is really preserved as an Array[Datatype], for example, String* is actually Array[String].

Note: To make the last parameter variable in length, we use *.

Syntax:

```
def Nameoffunction(args: Int *) : Int = { s
foreach println. }
```

The following are some limitations of varargs:

- The repeating argument must be the last parameter in the list.

  ```
  def sum(a :Int, b :Int, args: Int *)
  ```

- There are no default values for any arguments in the method that contains the varargs.
- All values must be of the same data type, else an error will occur.

  ```
  > sum(4, 6, 2000, 3000, 4000, "one")
  > error: type mismatch;
  found : String("one")
  required: Int
  ```

- Because args is an array within the body, all values are packed into an array.

Example:

```
// program of varargs
object PFP
{

    // the Driver code
    def main(args: Array[String])
    {

        // Calling function
        println("The Sum is: " + sum(5, 3, 1000,
2000, 3000));
    }


    // declaration and definition of the
function
    def sum(x :Int, y :Int, args: Int *) : Int =
    {
        var result = x + y

        for(arg <- args)
        {
            result += arg
        }

        return result
    }
}
```

Example:

```
// program of varargs
object GPFP
{
    // the Driver code
    def main(args: Array[String])
    {

        // calling the function
        printPeek("Peeks", "for", "peeks")

    }
```

```
    // declaration and definition of the function
    def printPeek(strings: String*)
    {
        strings.map(println)
    }

}
```

SCALA FUNCTION COMPOSITION

The Function composition is a process of combining two or more functions. To execute its job, one function keeps a reference to another function during the composition. Function composition can take place in a variety of ways, as shown below:

- **compose:** The compose technique uses val functions.

  Syntax:

  ```
  (function_1 compose function_2)(parameter)
  ```

- In the aforementioned syntax, function_2 first works with the parameter passed, then passes and returns a value to function_1.

  First example:

  ```
  // program to illustrate compose method with the
  val function

  // Creating the object
  object PFP
  {
      // the main method
      def main(args: Array[String])
      {
          println((add compose mul)(2))

          // adding the more methods
          println((add compose mul compose sub)(2))
      }

      val add=(x: Int)=> {
          x + 1
      }
  ```

```
    val mul=(x: Int)=> {
        x * 2
    }

    val sub=(x: Int) =>{
        x - 1
    }
}
```

- In the above instance, the mul function was used first, yielding 4(2 * 2), followed by the add function, yielding 5(4 + 1). (add compose mul compose sub)(2) will print 3 (step 1: 2 − 1 = 1, step 2: 1 * 2 = 2, step 3: 2 + 1 = 3).
- **andThen:** The andThen technique works with val functions as well.

  Syntax:

  ```
  (function_1 andThen function_2)(parameter)
  ```

- In the above syntax, function_1 operates with the parameter passed first, then passes and returns a value to function_2. or the following:

  ```
  Function_2(function_1(parameter))
  ```

  Second example:

  ```
  // program to illustrate andThen method with the
  val function

  // Creating the object
  object PFP
  {
      // the main method
      def main(args: Array[String])
      {
          println((add andThen mul)(2))

          // Adding the more methods
          println((add andThen mul andThen sub)(2))
      }
  ```

```
    val add=(x: Int)=> {
        x + 1
    }

    val mul=(x: Int)=> {
        x * 2
    }

    val sub=(x: Int) =>{
        x - 1
    }
}
```

- In the preceding example, we used the add function to get 3(2 + 1), then the mul function to get 6(3 * 2). Similarly, adding (andThen mul andThen sub)(2) produces 5 (step 1: 2 + 1 = 3, step 2: 3 * 2 = 6, step 3: 6 – 1 = 5).
- Method to method passing: Other methods are passed methods.

  Syntax:

  ```
  Function_1(function_2(parameter))
  ```

- It functions similarly to the compose function but uses the def and val methods.

  Third example:

  ```
  // program to illustrate passing methods to
  methods

  // Creating the object
  object PFP
  {
      // the main method
      def main(args: Array[String])
      {
          println(add(mul(2)))

          // Adding the more methods
          println(add(mul(sub(2))))
      }
  ```

```
    val add=(x: Int)=> {
        x + 1
    }

    val mul=(x: Int)=> {
        x * 2
    }

    val sub=(x: Int) =>{
        x - 1
    }
}
```

- In the above example, the mul function was used first, yielding 4(2 * 2), followed by the add function, yielding 5(4 + 1). Similarly, add(mul(sub(2)) produces 3 (step 1: 2 – 1 = 1, step 2: 1 * 2 = 2, step 3: 2 + 1 = 3).

IN SCALA, CALL A METHOD ON A SUPERsCLASS

When we wish to call a method from another class, we use this idea. So, anytime a base and subclass have the same named methods, we utilize the super keyword to call the base class function to resolve ambiguity. The concept of Inheritance introduced the keyword "super."

The following is an example of calling a method on a superclass.

First example:

```
// program to call the method on a superclass in
the Scala

/* The Base class ComputerScience */
class ComputerScience
{
    def read
    {
        println("We're reading")
    }
    def write
    {
        println("We're writing")
    }
}
```

```
/* Subclass */
class Scala extends ComputerScience
{
    // Note that readThanWrite() is only in the
Scala class
    def readThanWrite()
    {
        // Will invoke or call the parent class
read() method
        super.read

        // Will invoke or the call parent class
write() method
        super.write
    }
}

// Creating the object
object Peeks
{
    // the main method
    def main(args: Array[String])
    {
        var ob = new Scala();

        // Calling the readThanWrite() of Scala
        ob.readThanWrite();
    }
}
```

Second example:

```
// program to call the method on a superclass in
the Scala

/* Super-class Person */
class Person
{
    def message()
    {
        println("This is the person class");
    }

}
```

```
/* Sub-class Student */
class Student extends Person
{

override def message()
    {
        println("This is the student class")
    }

    // Note that display() is only in the Student
class
    def display()
    {
        // will invoke or call the current class
message() method
        message ()

        // will invoke or call the parent class
message() method
        super.message
    }
}

/* Creating the object */
object Peeks
{
    // the main method
    def main(args: Array[String])
    {
        var st = new Student();

        // Calling the display() of Student
        st.display();
    }
}
```

SCALA IMPLICIT CONVERSIONS

In Scala, implicit conversions are a collection of methods that are invoked when an object of the incorrect type is utilized. It enables the compiler to transform from one type to another automatically.

Implicit conversions are used in two scenarios:

- First, if a type X and S expression do not match the intended type Y expression.
- Second, if the selector m does not represent a member of X in a selection e.m of expression e of type X.

In the first condition, a conversion suited to the phrase and whose result type matches Y is sought. In the second condition, a conversion relevant to the expression is sought, resulting in a member named m.

Let us illustrate with an example.

It is permissible to perform the following operation on the two lists ca and da of type List[Int]:

```
ca = da
```

Assume the following implicit methods listorder and intorder are declared in the scope:

```
implicit def listorder[X](a: List[X])
(implicit elemorder: X => Ordered[X]):
Ordered[List[X]] =
new Ordered[List[X]] { /* .. */ }
implicit def intorder(a: Int): Ordered[Int] =
new Ordered[Int] { /* .. */ }
```

Example:

```
// Scala program of implicit conversions
import X.fromString
import scala.language.implicitConversions

case class X(s: String)
object X
{
    // Using implicitConversions
    implicit def fromString(s: String): X = X(s)
}

class Z
```

```
{
def m1(x: X) = println(x)
def m(s: String) = m1(s)
}

// Creating object
object Z
{
    // Main method
    def main(args: Array[String])
    {
        var c : X = ("PeeksforPeeks")
        println(c)

    }
}
```

The compiler warns while compiling the implicit conversion definition if it is used at random.

To prevent the warnings, we must either:

- In the scope of the implicit conversion definition, include scala.language.implicitConversions.
- With -language:implicitConversions, run the compiler.

Example:

```
// program of the implicit conversions
import ComplexImplicits._

object ComplexImplicits
{
    // the implicit conversion
    implicit def DoubleComplex(value : Double) =
                        new Complex(value,0.0)

    implicit def TupleComplex(value :
Tuple2[Double,Double]) =
                        new
Complex(value._1,value._2);

}
```

```
// Creating the class containing different method
class Complex(val r : Double, val i : Double)
{

    def +(that: Complex) : Complex =
    (this.r + that.r, this.i + that.i)

    def -(that: Complex) : Complex =
    (this.r - that.r, this.i + that.i)

    def unary_~ = Math.sqrt(r * r + i * i)

    override def toString = r + " + " + i + "i"

}

// Creating the Object
object Complex
{

val i = new Complex(0,1);

// the main method
def main(args : Array[String]) : Unit =
{
    var a : Complex = (5.0,5.0)
    var b : Complex = (2.0,3.0)
    println(a)
    println(a + b)
    println(a - b)
    println(~b)

    var c = 6 + b
    println(c)
    var d = (4.0,4.0) + c
    println(d)
}
}
```

This chapter covered Named Arguments, Functions Call-by-Name, Closures, Nested Functions, Parameterless Method, Recursion, Partially Applied functions, and Method Overloading and Overriding. Moreover, Method Invocation, Format and Formatted Method, Controlling Method Scope, Repeated Method Parameters, Partial Functions, Lambda Expression, and Function Composition.

CHAPTER 7

Scala Exceptions

IN THIS CHAPTER

- Exception Handling
- Throw Keyword
- Try-Catch Exceptions
- Finally Exceptions
- Either

In the previous chapter, we covered Scala methods and packages, and in this chapter, we will discuss exceptions in Scala.

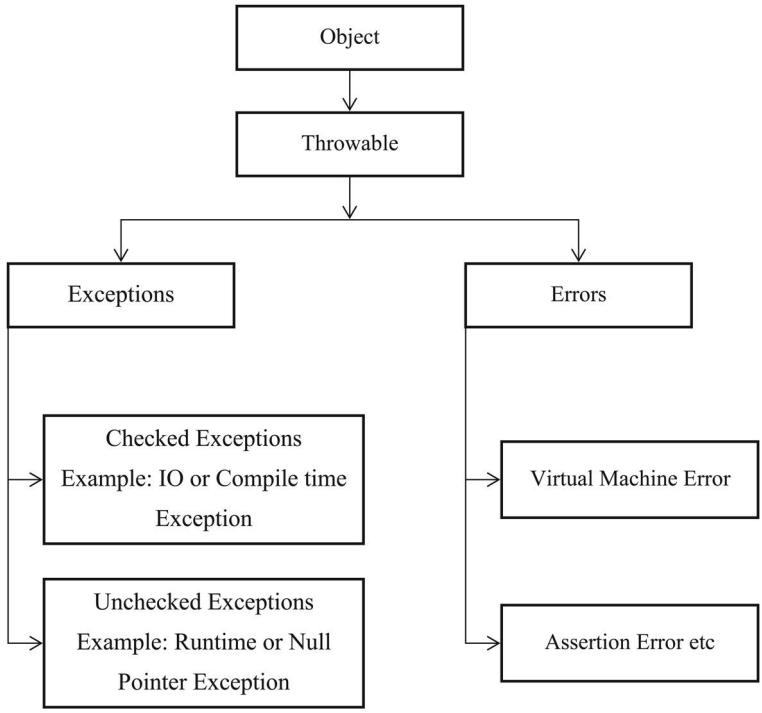
EXCEPTION HANDLING IN SCALA

An exception is an unexpected or unwelcome occurrence that occurs during the execution of a program, that is, during run time. These events affect the program's flow control during execution. These aren't hazardous circumstances, and the program can manage them.

HIERARCHY OF EXCEPTION

All exception and error types are subclasses of the root class of the hierarchy, Throwable. Exception is in charge of one branch. This class represents exceptional conditions that user programs should be aware of. An example of such an exception is NullPointerException. Another branch, Error, is used by the Java runtime system to identify faults in the runtime environment. An instance of such an error is StackOverflowError.

DOI: 10.1201/9781003358527-7



Hierarchy of exception.

SCALA EXCEPTIONS

Scala's exception handling is built differently, although it acts precisely like Java and works smoothly with current Java libraries. In Scala, all exceptions are unchecked. There is no idea of a checked exception. Scala provides a lot of flexibility in terms of being able to select whether or not to catch an exception.

Note: In Java, "checked" exceptions are checked at build time. If a method has the potential to throw an IOException, it must be declared.

What Is the Scala Exception?

Exceptions in Scala work in the same manner that they do in C++ or Java. The current operation is canceled when an exception occurs, such as the ArithmeticException seen in the preceding example. The runtime system searches for an exception handler that can accept an ArithmeticException. The innermost handler regains control. If no such handler is found, the application exits.

THROWING EXCEPTIONS

Making an exception. It appears to be the same as in Java. We build an exception object and then throw it using the throw keyword.

Syntax:

```
throw new ArithmeticException
```

TRY/CATCH CONSTRUCT

Scala's try/catch construct differs from Java's; in Scala, try/catch is an expression. Instead of creating a distinct catch clause for each exception, the exception in Scala that results in a value can be pattern matched in the catch block because try/catch is an expression in Scala. In Scala, here's an instance of exception handling using the traditional try-catch block.

```
// program of the try-catch Exception
import java.io.IOException

// Creating the object
object PFP
{
    // the main method
    def main(args:Array[String])
    {
        try
        {
            var N = 6/0

        }
        catch
        {
            // Catch block contains the cases.
            case i: IOException =>
            {
                println("IOException occurred.")
            }
            case a : ArithmeticException =>
```

```
            {
                println("The Arithmetic Exception
occurred.")
            }

        }
    }
}
```

A single catch block in Scala may handle all types of exceptions, giving flexibility.

THE FINALLY CLAUSE

A finally block can use if we want some element of our code to run regardless of how the expression ends. Here's an illustration of what we mean:

```
// program of the finally Exception

// Creating the object
object PFP
{
    // The main method
    def main(args: Array[String])
    {
        try
        {
            var N = 6/0
        }
        catch
        {
            // Catch block contains the case.
            case ex: ArithmeticException =>
            {
                println("The Arithmetic Exception
occurred.")
            }
        }
        finally
        {
            // The Finally block will execute
```

```
            println("This is the final block.")
        }
    }
}
```

SCALA THROW KEYWORD

In Scala, the throw keyword explicitly throws an exception from a function or a block of code. The throw keyword is used in Scala to throw and catch exceptions explicitly. It may also use to throw user-defined exceptions. The handling of exceptions in Java and Scala is relatively similar. Except that Scala only sees all errors as runtime exceptions; thus, it does not compel us to handle them and instead utilizes pattern matching in the catch block. In Scala, the throw is an expression with the outcome type nothing. If the result does not evaluate anything, we can substitute it for any other expression.

Essential things to keep in mind:

- The throw expression returns nothing.
- The throw is the keyword for throwing an exception, whereas throws are the keyword for declaring an exception.
- To handle exceptions, the catch block employs pattern matching.

Syntax:

```
throw exception object
Example:
 throw new ArithmeticException("divide by 0")
```

Example:

```
val z = if (d % 10 == 0) 5 else throw new
RuntimeException("d must be a multiple of 10")
```

Explanation: If d is a multiple of 10, then 5 is assigned to z; otherwise, an exception is thrown before z can be initialized to anything. As a result, we may claim that throw has any value. Throwing an exception in Scala is the same as in Java. We build an exception object and then use the throw keyword to throw it:

Example:

```
// program of the throw keyword

// Creating the object
object Main
{
    // Define the function
    def validate(article:Int)=
    {
        // Using the throw keyword
        if(article < 25)
            throw new ArithmeticException("We are
not eligible for internship")
        else println("We are eligible for
internship")
    }

    // the main method
    def main(args: Array[String])
    {
            validate(27)
    }
}
```

If the number of articles is fewer than 20, we receive no output. A scala method can throw an exception instead of returning when an error occurs. The following example shows a single exception raised by a function.

```
// program of throw keyword

// Creating the object
object PFP
{
    // the main method
    def main(args: Array[String])
    {

    try
    {
        func();

    }
```

```
    catch
    {
        case ex: Exception => println("Exception
caught in the main: " + ex);

    }
    }

    // Defining the function
    def func()
    {
        // Using the throw exception
        throw new Exception("Exception thrown from the
func");

    }
}
```

SCALA TRY-CATCH EXCEPTIONS

The Try-Catch construct is different in Scala than in Java; in Scala, Try-Catch is an expression. In the catch clause, Scala employs pattern matching. If we have to implement a series of code that can throw an exception and we want to control that exception, we should use the Try-Catch segment because it allows us to try-catch all types of exceptions in a single block. We must write a series of case statements in catch because Scala uses matching to analyze and handle exceptions.

Example:

```
// program of try-catch exception

// Creating the object
object Arithmetic
{

// the main method
def main(args: Array[String])
{

    // Try clause
    try
```

```
    {
        // Dividing the number
        val result = 12/0
    }

    // Catch clause
    catch
    {
            // Case-statement
            case x: ArithmeticException =>
            {

            // Display this if the exception is
found
            println("Exception: Number is not
divisible by zero.")
        }
    }
}
}
```

Example:

```
// program of Try-Catch Exception
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

// Creating the object
object PfP
{

// the main method
def main(args: Array[String])
{

    // Try clause
    try
    {
        // Creating object for FileReader
        val t = new FileReader("input.txt")
    }
```

```
    // Catch clause
    catch
    {

        // Case statement1
        case x: FileNotFoundException =>
        {
            // Displays this if the file is
missing
            println("Exception: File missing")

        }

        // Case statement2
        case x: IOException =>
        {

            // Displays this if the input/output
exception is found
            println("Input/output Exception")

        }
    }
}
}
```

The try block is performed first, and if an exception is raised, each catch clause case is tested, and the one that matches the thrown exception is returned as output.

SCALA FINALLY EXCEPTIONS

The final block in Scala performs critical functionality such as terminating a connection, stream, or releasing resources (it can be a file, network connection, database connection, etc). It will always be performed, whether or not an exception is thrown. After the try-and-catch blocks, but before control returns to its origin, the finally block will be performed.

Syntax:

```
try {
    // scala code here
 }
```

```
finally {
        println("this block of the code is always
executed")
        // scala code here, such as to close the
database connection
     }
```

CONTROL FLOW IN TRY-FINALLY

In this scenario, whether or not an exception occurs in the try-block, finally is always executed. However, control flow is determined by whether or not an exception occurs in the try block.

1. **Exception raised:** Control flow will eventually be blocked, and the default exception handling method will use. If exception occurs in the try block.

   Example:

```
// program to demonstrate the control flow of
the try-finally clause
// when exceptions occur in the try block

// Creating the object
object PFP
{
    // the main method
    def main(args: Array[String])
    {
        var arr = new Array[Int](5)

        try
        {
            var i = arr(5)

            // this statement will never execute
as the exception is raised
            // by the above statement
            println("Inside try block")
        }

        finally
```

```
        {
            println("finally block execute")
        }

        // the rest program will not execute
        println("Outside try-finally clause")

    }
}
```

2. **Exception not raised:** If no exception is raised in the try block, control will pass to the finally block, which the rest of the program will follow.

   Example:

```
// program to demonstrate control flow of try-
finally clause
// when an exception does not occur in the try
block

// Creating the object
object PFP
{
    // the main method
    def main(args: Array[String])
    {

    try
        {
            val str1 = "123".toInt

            // this statement will execute as no
            // any exception is raised by the
above statement
            println("Inside try block")
        }

        finally
        {
            println("The finally block
executed")
        }
```

```
            // rest program will execute
            println("The Outside try-finally
  clause")
        }


    }
```

TRY-CATCH-FINALLY CLAUSE

Finally, when combined with a try/catch block, ensure that a part of the code is performed even if an exception is raised.

Example:

```
// program of the try-catch-finally clause

// Creating the object
object PFP
{
    // the main method
    def main(args:Array[String])
    {
        try
        {
            // creation of an array
            var array = Array(2, 7, 4)
            var b = array(5)
        }
        catch
        {
            // the Catch block contain cases.
            case e: ArithmeticException =>
println(e)
            case ex: Exception => println(ex)
            case th: Throwable=> println("The
unknown exception"+th)
        }
        finally
        {
            // The Finally block will execute
            println("this block always executes")
        }
```

```
        // the rest program will execute
        println(" The rest of code executing")
    }
}
```

In the above example, we build an array in the try block and assign the value to variable b from that array. Still, it throws an exception because the index of the array we are using to give the value to variable b is out of the range of array indexes. Finally, no matter what, the catch block will catch the exception and display the message.

SCALA EITHER

Either acts precisely like an Option in Scala. The main difference is that with Either, it is possible to produce a text explaining the instructions about the problem that occurred. Either has two children, Right and Left, where Right is identical to the Some class and Left is similar to the None class. Left is used for failure, and we may return the error that happened inside the child Left of the Either, whereas Right is used for success.

Example:

```
Either[String, Int]
```

In this case, the String is used for the Left child of Either because it is the left argument of an Either, and the Int is used for the Right child because it is the right argument of an Either. Let's now go through it in more depth using some instances.

Example:

```
// program of Either

// Creating the object and inheriting
// main method of trait App
object PfP extends App
{

    // Defining the method and applying Either
    def Name(name: String): Either[String, String] =
    {

        if (name.isEmpty)
```

```
            // Left child for the failure
            Left("There is no name.")

        else
            // Right child for the success
            Right(name)
    }

    // Display this if the name is not empty
    println(Name("PeeksforPeeks"))

    // Display the String present in the Left
child
    println(Name(""))
}
```

The isEmpty method examines if the name field is empty or filled; if it is empty, the Left child returns the String within itself; if it is not empty, the Right child returns the name specified.

Example:

```
// program of Either with the Pattern matching

// Creating the object and inheriting
// main method of trait App
an object either extends App
{

    // Defining the method and applying Either
    def Division(q: Int, r: Int): Either[String,
Int] =
    {
        if (q == 0)
            // the Left child for the failure
            Left("Division not possible.")
        else
            // Right child for the success
            Right(q / r)
    }

    // Assigning the values
    val x = Division(4, 2)
```

```
    // Applying pattern matching
    x match
    {
        case Left(l) =>

        // Display this if division is not
possible
        println("Left: " + l)

        case Right(r) =>

        // Displays this if the division is
possible
        println("Right: " + r)
    }
}
```

Because division is allowed in this case, Right returns 2. In this example of Either, we've used Pattern Matching.

This chapter covered Exception Handling, Throw Keyword, Try-Catch Exceptions, Finally Exceptions, and Either.

Appraisal

A functional and object-oriented software program with great abstraction is called Scala. Static types in Scala assist to avoid problems in complicated applications, while the JVM and JavaScript runtimes enable the creation of high-performance systems with simple access to huge library ecosystems.

A modern computer program called Scala was created by Martin Odersky. The language's creation began in 2001 and was introduced to the public in early 2004. Martin Odersky was heavily involved in developing javac (the major Java compiler) and also invented Generic Java, a generic programming capability added to the Java programming language in 2004. Because of this, it is not surprising that Scala is similar to Java in many aspects and was even created to run in the JVM (Java Virtual Machine). Many other programming languages and concepts from programming research both had an impact on the creation of Scala. In fact, Martin Odersky has remarked that there aren't many "new" features in Scala and that the language's "innovation derives mostly from how its constructs are put together." Essentially, the goal was to create a "better language."

SCALA'S HISTORY

Scala's initial design was established by Martin Odersky at the École Polytechnique Fédérale de Lausanne in 2001 (EPFL). Scala was being developed at the same time as Funnel, a programming language that combined ideas from Petri nets and functional programming, was being created. Before developing Scala, Odersky worked on Generic Java, javac, and Sun's Java compiler. The first time the language was used internally was in 2003.

In addition, starting in January 2004, Scala was made accessible to the entire public. In June 2004, the language was formally made available on the Java and.NET platforms. The language's 2.0 version, which has better capabilities, was published in March 2006. The language's development led to the formal termination of.Net support in 2012.

DOI: 10.1201/9781003358527-8

The Scala team received a five-year, €2.3 million research grant from the European Research Council on January 17, 2011, as a result of the language's consistent development and strong focus on the functional programming paradigm. The grant had a five-year expiration date.

On the May 12, 2011, Odersky and other relevant colleagues continued the establishment of Typesafe Inc. Typesafe Inc., a company whose mission is to provide training, commercial support, and other Scala-related services. In 2011, Greylock Partners fueled the operations of Typesafe by investing around $3 million in the firm.

IS SCALA VALUABLE IN 2022?

A language to know in 2022 is Scala. The pay for Scala developers is competitive, and they are in high demand. There are currently over 24,000 Scala job advertisements on LinkedIn. The average Scala developer salary in the US is $139,292, according to ZipRecruiter.

SCALA IS WORTH LEARNING FOR SOME REASONS

Still not sure that studying Scala is a worthwhile investment of your time and effort? Aside from the increasing need for Scala developers and the high pay potential in this area, there are several more reasons why studying Scala is worthwhile.

- **Java compatibility is required:** Scala is executed via the JVM. As a result, it can interact with Java code cleanly, allowing us to use Java libraries straight from Scala code. To put it differently, we can use Java to call Scala code and develop parts of our program in Scala while the rest is done in Java. With this capability, it is not surprising that Scala will become a popular language.
- **Language with multiple paradigms:** Scala is distinguished from Java by its support for both object-oriented and functional programming paradigms. Learn one language from each paradigm, such as imperative, logical, functional, and object-oriented programming (OOP), to improve your competitive programming abilities. Scala enables us to study both functional and OOP at the same time.
- **Typing that is static:** Without running a program, static typing allows us to identify programming problems fast, reliably, and automatically. This contrasts with dynamic typing when problems are discovered after running the program. Scala is a statically typed

language that frequently feels fluid and dynamic. We can work more effectively with precise code, saving time on debugging and testing.

- **Concise syntax:** As previously stated, Scala is compiled and statically typed. It is much shorter than Java. Scala is compared to a scripting language. Scala developers can work considerably quicker and more effectively if they write succinct and clean code.

- **Productivity and efficiency:** Scala's multi-paradigm and static typing capabilities, concise syntax, and Java compatibility enable us to be a more efficient and productive developer. We can create fewer lines of code, complete projects faster, decrease problems earlier, and improve our program's end-user experience.

- **Marketable:** Who doesn't want to be marketable as a programmer? A better job and professional progress are good reasons to master a new technology or framework. Learning Scala will undoubtedly increase your marketability. Scala is being used or migrated by many firms, including Twitter, LinkedIn, Foursquare, and Quora.

  Given Scala's marketing as a Scalable language, the large investment banks and financial companies will soon look at Scala for low latency solutions. Twitter has already revealed recommended practices for developing Scala applications as Effective Scala, similar to the guidance given in Effective Java.

- **Built-in language for Best Practices and Patterns:** One thing we might not know about Scala is that it was created at the Swiss university EPFL in an attempt to apply for recent advances in programming language research to a language that could garner public acceptance, similar to Java.

  Several best practices and patterns are built into the language; for example, val declares top-level immutability far superior to the overloaded final in Java or const/read-only in C# with its strange constraints. Scala also supports closures, a feature borrowed from the functional programming paradigm of dynamic languages such as Python and Ruby.

- **Language expressiveness:** Scala wins when we compare Scala to Java, as we did in my earlier piece on the differences between Scala and Java. Scala is a naturally expressive language. Scala also has a

plethora of excellent and helpful code. Scala is attracting an increasing number of Java engineers who appreciate attractive code.

To give us a sense, below is a word count application developed in both Java and Scala; you can see the difference in language expressiveness for ourselves. Scala has accomplished in one line what Java has taken more than ten lines to do.

- **Frameworks in development, such as Akka, Play, and Spark:** We may be aware that Scala is expanding, and it is expanding rapidly. There are a lot of great libraries and frameworks on the way. Companies that have begun to use Scala are also contributing to Scala's recent emergence as a mainstream language.

  Various good Scala web frameworks are available, such as Lift and Play. Akka, another Scala-based concurrent framework, has already established itself as a toolkit and runtime for developing highly concurrent, distributed, and fault-tolerant event-driven JVM applications.

  Scala is also employed in the Big Data sector alongside Apache Spark, which has driven its popularity by many Java professionals interested in the Big Data space.

- **Relative, it is simple to learn:** Learning a traditional functional programming language like Haskell or OCaml is more challenging for a Java developer than Scala. Scala is very straightforward to learn due to its OOP functionality.

While learning Functional Programming, Java professionals may still be productive in Scala by utilizing their previous OOP skills. Scala, like Java, features simple syntax, decent libraries, extensive online documentation, and significant industry support.

SCALA IS USED BY LARGE CORPORATIONS SUCH AS TWITTER, NETFLIX, AND AIRBNB

- **Scala is involved in commercial ventures:** Scala is often assumed to be utilized in a wide range of enterprise-related settings, with many significant firms utilizing the capabilities of such a programming language. However, there is a lot of doubt about Scala's practicality, particularly in real-world business applications. For that purpose, it may be beneficial to understand the essence of Scala and its theory and to see actual examples of Scala's application in the business context.

- **Scala's nature:** Scala is a computer language that blends two separate programming paradigms, featuring aspects of both object-oriented and function-oriented languages. Such a technological spectrum offers at least two significant benefits. First and foremost, because it runs on JVM, it enables the use of any general programming language's reliability, omnipresence, features, and repute (JVM). Second, Scala continues to enable access to Java libraries, both third-party and your own, finally permitting any borrowing of the matter. Scala quickly extends via libraries to solve numerous challenges, hence "scaling" in size as its name indicates.
- **Cases in the enterprise:** It is worth mentioning that the Scala feature has long ensured the backing of several large corporations worldwide. The reasons behind this will become apparent after a thorough examination of real-world business cases:
    - **LinkedIn:** LinkedIn is the most popular professional networking site and is well-known for incorporating Scala into its operational principles. The clearest example would be Norbert, a Scala-based framework that enables the rapid development of client-server message-based applications used in LinkedIn's real-time Social Graph and Search Engine. According to Chris Conrad, Head of Engineering Team, the "killer feature [of Scala] is the seamless integration of Java and Scala, making it low risk to add and minimizes the burden to experiment." It significantly reduces the amount of frustration associated with software development. In turn, Scala is just a lot better than Java; that simplifies the whole writing process: it's clearer, concise, re-usable using mixins and characteristics, and compensates for the usage of Actors concurrently. Scala's selection over the other programming languages was thus totally warranted.
    - **Twitter:** When discussing Twitter, a well-known social network, it's critical to note how much it relies on Scala in various infrastructure groups. Scala for Twitter's social adjacency store, name search, "whom to follow," streaming API, storage systems, and geo service are just a few instances of its breadth. In reality, Twitter went from Ruby to Scala due to the latter's language flexibility, dependability, and high performance. Compared to Ruby, Scala provides a more robust and reliable foundation for Twitter's long-lived servers, which can be attributed solely to the language's JVM foundation.

- **Airbnb:** Airbnb is a website where users can rent out a house for a certain length of time. It's a massive platform constructed with the aid of many computer languages, most notably Scala. Scala is mostly utilized in AirBnB's financial reporting pipeline, a system that evaluates the economic effect of different goods at different times in their life cycle. Airbnb largely used Scala due to its immutability, ease of use, and lazy evaluation, allowing them to examine each product independently and draw appropriate accounting conclusions.
- **Thatcham:** Thatcham is a research-focused organization that provides different statistics to assist car manufacturers in lowering insurance claims costs. The firm uses Scala to meet its requirements for delivering data on the safe and cost-effective repair of automobiles. Thatcham chose Scala because it allows for more work with less code, provides concurrency-ready notions, and allows him to expand on existing Java models. Thatcham's website and research concepts are focused on Scala for that purpose.
- **Tumblr:** Tumblr, a microblogging social network, has had significant scaling challenges as it has grown in popularity throughout the world. The platform found the implementation of microservices appealing, though it failed to organize a well-structured infrastructure. To that end, a business created Colosseum, a Scala-based framework that tries to overcome historical issues by providing a clear and straightforward architecture for scaling high-quality, stable, and durable microservices. Tumblr was drawn to Scala's ability to create expressive code and construct maximally simple DSL with minimal boilerplates.
- **Netflix:** Scala is used in the architecture and design of Netflix, the world's largest movie and TV show streaming service. According to business executives, Scala works well with the Netflix Platform and the JVM Ecosystem while allowing for the reuse of existing Groovy and Java code. Similarly, many sophisticated Java libraries already in Netflix's toolbox can easily access Scala, which the firm finds attractive. Scala is an excellent choice for developing a restful and reliable API with various tools that improve the searching algorithms, service's ML-based viewing suggestions, and enable interactive testing.

ADVANTAGES OF SCALA

If a programming language wishes to threaten Java's supremacy, it needs to give programmers some compelling features. To that aim, Scala brings numerous pros to the table. Here's a taste of its benefits:

- Scala offers accurate grammar, minimizing boilerplate programming. Scala-based apps require less code than Java-based counterparts.
- It is a functional and object-oriented language of programming. Because of this combination, Scala is the best choice for web development.
- We can use Scala to run Java code.
- Scala utilizes an expressive type system that guarantees statistical abstraction is safe and consistent.
- It's straightforward to learn, particularly for programmers with an object-oriented experience in Java or equivalent language.
- Scala is very scalable and ideal for constructing fault-tolerant, highly parallel systems.
- It's perfect for data analytics when assisted by technologies like Apache Spark.

HOW ARE SCALA AND JAVA DISSIMILAR?

Herein is the true comparison. Although some of these things have previously been discussed, we will revisit them for a more direct comparison.

- Scala combines functional programming, statistically typed, and object-oriented languages, whereas Java is an object-oriented, general-purpose programming language.
- In Java, functions are objects, but in Scala, they are variables.
- To do ordinary operations, Java takes several lines of code, but Scala programming is quick and concise. Scala code is composed of half as many lines as Java code.
- Scala does not offer backward compatibility, unlike Java.

- Scala is more difficult to learn than Java, with a steeper learning curve and more sophisticated syntax.
- Using the "lazy" keyword, Scala's "lazy evaluation" feature enables programmers to delay time-consuming operations until they are required. Java lacks this option.
- Java does not allow operator overloading, although Scala does.

DRAWBACKS OF SCALA VS. JAVA

Scala, like Java, has its drawbacks, which include:

- It has a small community, especially when compared to Java.
- Scala has limited backward compatibility.
- Despite the name being simple to learn, Scala has concepts and functionalities that many programmers are unfamiliar with, resulting in a higher learning curve.
- Scala's development tools are still in their infancy; they are not as complex or sophisticated as Java's, particularly the IDE plug-ins.

HOW DO SCALA AND KOTLIN COMPARE?

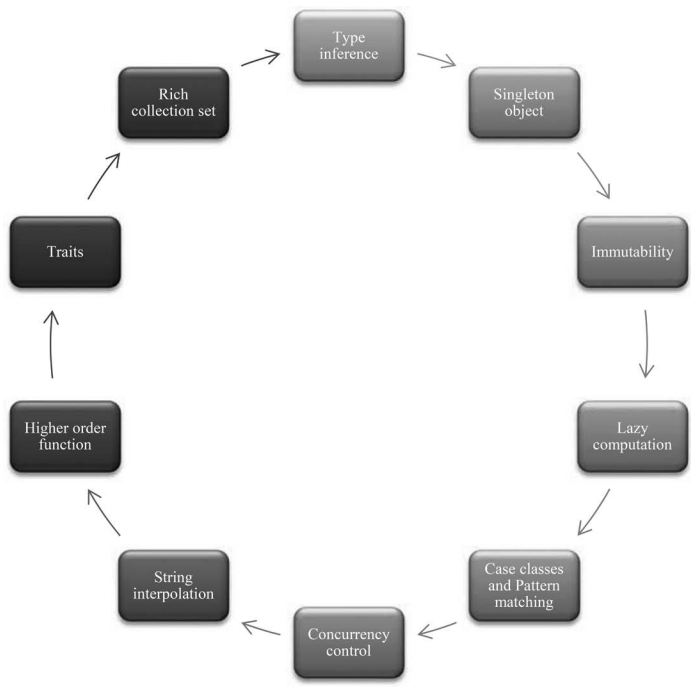
Kotlin also provides several current features that Scala lacks. Included are null safety, extension functions, lambdas, top-level functions, ranges for iterating over collections, and object declarations such as data classes and object classes (similar to C structs). Scala and Kotlin both provide Null Safety, although at various degrees. When comparing Kotlin to Scala for pattern matching, it is clear that Kotlin is lacking. Users find this Scala capability useful and commend it highly.

The Kotlin programming language supports the overloading operator. The operator may have multiple implementations based on the parameters. The Scala programming language also offers operator overloading. Comparing Scala vs. Kotlin in terms of simplicity, it is easy to conclude that Kotlin is the more accessible programming language. Because it was created with Java compatibility in mind, this is the case. Because it utilizes the same JVM bytecode as Scala, you can use Kotlin with existing Java libraries without worrying about compatibility concerns. These are some reasons why you should consider adopting Kotlin instead of other programming languages, such as Java or Scala.

SCALA CHARACTERISTICS

Scala has the following characteristics:

- Type inference
- Singleton object
- Immutability
- Lazy computation
- Case classes and Pattern matching
- Concurrency control
- String interpolation
- Higher order function
- Traits
- Rich collection set



Types of Scala characteristics.

Type Inference

Scala does not need us to provide the data type and function return type explicitly. Scala is intelligent enough to infer the type of data. The type of the final expression in the function determines the function's return type.

Singleton Object

There are no static variables or functions in Scala. Scala employs singleton objects, classes with just one object in the source file. The object keyword is used instead of the class keyword to declare a singleton object.

Immutability

Scala employs the idea of immutability. By default, each declared variable is immutable. Immutable indicates that we cannot change its value. We may also construct variables that can alter. Immutable data aids in the management of concurrency control, which necessitates the management of data.

Computational Laziness

Scala's computation is by default lazy. Scala only evaluates expressions when they are needed. The lazy keyword can use to declare a lazy variable. It's utilized to boost performance.

Pattern Matching and Case Classes

Scala case classes are simply normal classes that are immutable by default and decomposable by pattern matching. By default, all of the arguments in the case class are public and immutable. Pattern matching is supported through case classes. As a result, you can create more rational code.

Concurrency Control

Scala comes with a standard library that incorporates the actor model. Using actor, you may write concurrent code. Scala offers another platform and tool for dealing with concurrency called Akka. Akka is an independent open-source framework that supports actor-based concurrency. Akka actors can be distributed or used in conjunction with software transactional memory.

String Interpolation

Scala 2.10.0 introduces a new way of creating strings from data. String interpolation is the term for this technique. String interpolation enables

users to incorporate variable references in processed string literals directly. Scala has three methods for string interpolation: s, f, and raw.

Higher Order Functions

A higher-order function accepts or returns a function as an argument. In other words, a function that interacts with another function is a higher-order function. Higher order functions enable the creation of function composition, lambda functions, anonymous functions, and so on.

Traits

A trait is analogous to an interface with just a partial implementation. The trait is a set of abstract and non-abstract methods in Scala. We can define a trait with just abstract methods or some abstract methods and non-abstract methods. Traits are compiled into Java interfaces with implementation classes containing any methods implemented in the traits.

Rich Set of Collection

Scala has an extensive collection library. It includes classes and attributes for data collection. These collections may or may not be changeable. We can put it to use how we see fit. The Scala.collection.mutable package includes all mutable collections. While using this package, we may add, remove, and update data.

The Scala.collection.immutable package includes all immutable collections. It does not permit data modification.

Bibliography

1. Introduction to Scala: https://www.geeksforgeeks.org/introduction-to-scala/, accessed on July 7, 2022.
2. History of Scala: https://www.javatpoint.com/history-of-scala#:~:text=Scala%20is%20a%20general%20purpose,released%20on%20January%2020%2C%202004, accessed on July 12, 2022.
3. Scala Programming Language: History, Features, Application, and Why Should Learn?: https://www.answersjet.com/2021/06/scala-programming-language-history-features-application-and-why-should-learn.html, accessed on July 15, 2022.
4. Scala Programming Language: History, Features, Applications, Installation, QA: https://www.devopsschool.com/blog/scala-programming-language-history-features-applications-installation-qa/, accessed on July 7, 2022.
5. Setting up the environment in Scala: https://www.geeksforgeeks.org/setting-up-the-environment-in-scala/, accessed on July 7, 2022.
6. Scala - Environment Setup: https://www.tutorialspoint.com/scala/scala_environment_setup.htm, accessed on July 7, 2022.
7. Scala - Environment Setup: https://www.bbminfo.com/scala/scala-environment-setup.php, accessed on July 7, 2022.
8. Scala Environment Setup: https://www.stechies.com/scala-environment-setup/, accessed on July 8, 2022.
9. Scala Installation: http://www.w3big.com/scala/scala-install.html, accessed on July 8, 2022.
10. Uses of Scala: https://www.educba.com/uses-of-scala/, accessed on July 8, 2022.
11. Hello, World! : https://docs.scala-lang.org/overviews/scala-book/hello-world-1.html, accessed on July 8, 2022.
12. Scala Basic: https://www.w3resource.com/scala-exercises/basic/scala-basic-exercise-1.php, accessed on July 8, 2022.
13. How to Write a Hello World Program in Scala?: https://www.educative.io/answers/how-to-write-a-hello-world-program-in-scala, accessed on July 8, 2022.
14. Hello world by defining a “main” method. Retrieved from https://riptutorial.com/scala/example/806/hello-world-by-defining-a–main–method, accessed on July 8, 2022.
15. What is Scala?: https://www.edureka.co/blog/what-is-scala/, accessed on July 8, 2022.

16. 6 Reasons Scala is Better Than Java.https://levelup.gitconnected.com/6-reasons-scala-is-better-than-java-c328cfb410d1, accessed on July 8, 2022.
17. UniformAccessPrinciple: https://martinfowler.com/bliki/UniformAccessPrinciple.html, accessed on July 9, 2022.
18. Scala | Uniform Access Principle: https://www.geeksforgeeks.org/scala-uniform-access-principle/, accessed on July 9, 2022.
19. Uniform Access Principle: https://www.scala-exercises.org/std_lib/uniform_access_principle, accessed on July 9, 2022.
20. Java vs. Scala: Which Is More Better?: https://javaassignmenthelp.wixsite.com/mysite/post/java-vs-scala-which-is-more-better, accessed on July 9, 2022.
21. Apache Spark and Scala Certification Training: https://www.simplilearn.com/big-data-and-analytics/apache-spark-scala-certification-training, accessed on July 9, 2022.
22. Python vs. Scala Comparison: Which Language to Choose for Apache Spark? : https://streamsets.com/blog/python-vs-scala-comparison/#:~:text=When%20it%20comes%20to%20performance%2C%20Scala%20is%20the%20clear%20winner,variable%20or%20expression%20at%20runtime, accessed on July 9, 2022.
23. Python vs. Scala – Which One to Choose for Big Data Processing? : https://www.netguru.com/blog/python-versus-scala, accessed on July 9, 2022.
24. Why Should I Learn Scala?: https://www.toptal.com/scala/why-should-i-learn-scala, accessed on July 9, 2022.
25. Scala Archive: https://www.scala-archive.org/nancy-bledsoe, accessed on July 9, 2022.
26. Kotlin vs Scala. Baeldung:. https://www.baeldung.com/kotlin/kotlin-vs-scala, accessed on July 9, 2022.
27. Scala vs Kotlin: Which One is Better for Your Project? DAC.digital:. https://dac.digital/scala-vs-kotlin-which-one-is-better-for-your-project/#:~:text=While%20both%20languages%20are%20statically,about%20types%20all%20the%20time, accessed on July 9, 2022.
28. Kotlin vs Scala: Which One to Choose for Your Next Project?: https://appinventiv.com/blog/kotlin-vs-scala/, accessed on July 9, 2022.
29. Difference Between Kotlin and Scala:. https://www.geeksforgeeks.org/difference-between-kotlin-and-scala/, accessed on July 9, 2022.
30. Kotlin vs Scala vs Java: https://dzone.com/articles/kotlin-vs-scala-vs-java, accessed on July 9, 2022.
31. Kotlin vs Scala - Which is the Best JVM Language for Developing Apps?:. https://appwrk.com/kotlin-vs-scala-which-is-the-best-jvm-language-for-developing-apps, accessed on July 9, 2022.
32. Kotlin vs Scala: https://www.spec-india.com/blog/kotlin-vs-scala, accessed on July 9, 2022.
33. Scala vs Kotlin: https://towardsdatascience.com/scala-vs-kotlin-practical-considerations-for-the-pragmatic-programmer-d50bcc96765f, accessed on July 10, 2022.
34. Scala Identifiers. https://www.geeksforgeeks.org/scala-identifiers/#:~:text=Scala%20identifiers%20are%20case%2Dsensitive,below%20four%20types%20of%20identifiers, accessed on July 10, 2022.

35. Scala Identifiers Example Tutorial: https://www.journaldev.com/8170/scala-identifiers-example-tutorial, accessed on July 10, 2022.
36. Identifiers in Scala: https://www.includehelp.com/scala/identifiers.aspx, accessed on July 10, 2022.
37. Learning Scala: https://www.oreilly.com/library/view/learning-scala/9781449368814/apa.html accessed on July 10, 2022.
38. Different Types of Identifiers in Scala: https://www.bartleby.com/essay/Different-Types-Of-Identifiers-In-Scala-PK3SCGU5L4P, accessed on July 10, 2022.
39. Scala - Data Types: https://www.tutorialspoint.com/scala/scala_data_types.htm, accessed on July 10, 2022.
40. Data Types in Scala: https://www.geeksforgeeks.org/data-types-in-scala/, accessed on July 10, 2022.
41. Basic Types in Scala: https://intellipaat.com/blog/tutorial/scala-tutorial/basic-types-in-scala/, accessed on July 10, 2022.
42. Spark SQL - Data Types: https://spark.apache.org/docs/latest/sql-ref-datatypes.html, accessed on July 10, 2022.
43. Scala–Variables:https://www.tutorialspoint.com/scala/scala_variables.htm, accessed on July 10, 2022.
44. Variables in Scala: https://www.geeksforgeeks.org/variables-in-scala/, accessed on July 10, 2022.
45. Variables in Scala: https://www.datacamp.com/tutorial/variables-in-scala, accessed on July 10, 2022.
46. Scala Variable and Data Types: https://www.javatpoint.com/scala-variable-and-data-types, accessed on July 10, 2022.
47. Scala Variables: https://data-flair.training/blogs/scala-variables/, accessed on July 10, 2022.
48. Scala | console println(), printf() and readLine():. https://www.geeksforgeeks.org/scala-console-println-printf-and-readline/#:~:text=Console%20implements%20functions%20for%20displaying,with%20the%20function%20from%20scala, accessed on July 11, 2022.
49. println: Scala: https://www.dotnetperls.com/println-scala, accessed on July 11, 2022.
50. Scala println: How to Print Lines in Scala: https://thedeveloperblog.com/scala/println-scala, accessed on July 11, 2022.
51. scala.Console. Scala Standard Library API. https://www.scala-lang.org/api/current/scala/Console$.html, accessed on July 11, 2022.
52. Console Scala: println (), printf () and readLine (). Acervo Lima. https://acervolima.com/console-scala-println-printf-e-readline/, accessed on July 11, 2022.
53. Printing debugging output in Scala with println and debugging symbols: https://www.oreilly.com/library/view/scala-cookbook/9781449340292/ch14s13.html, accessed on July 11, 2022.
54. Scala Identifiers: https://www.geeksforgeeks.org/scala-identifiers/, accessed on July 11, 2022.

55. Scala Identifiers: https://www.journaldev.com/8170/scala-identifiers-example-tutorial, accessed on July 11, 2022.
56. Identifiers in Scala: https://www.includehelp.com/scala/identifiers.aspx, accessed on July 11, 2022.
57. Identifiers, Names and Scopes: https://scala-lang.org/files/archive/spec/2.13/02-identifiers-names-and-scopes.html, accessed on July 11, 2022.
58. Scala Pattern Matching: https://www.geeksforgeeks.org/scala-pattern-matching/#:~:text=Pattern%20matching%20is%20a%20way,statement%20of%20Java%20and%20C, accessed on July 11, 2022.
59. Scala – Pattern Matchin: https://www.tutorialspoint.com/scala/scala_pattern_matching.htm, accessed on July 11, 2022.
60. Scala Pattern Matching: https://www.baeldung.com/scala/pattern-matching, accessed on July 11, 2022.
61. Advanced Pattern Matching in Scala: https://blog.knoldus.com/advanced-pattern-match/, accessed on July 11, 2022.
62. Scala Pattern Matching: Types of Pattern Matching with Example: https://data-flair.training/blogs/scala-pattern-matching/, accessed on July 11, 2022.
63. Comments in Scala: https://www.geeksforgeeks.org/comments-in-scala/, accessed on July 11, 2022.
64. Scala comments: https://www.javatpoint.com/scala-comments, accessed on July 11, 2022.
65. Scala Comments: Single & Multi-line Comments with Examples: https://data-flair.training/blogs/scala-comments/, accessed on July 12, 2022.
66. Scala – Comments: http://www.java2s.com/Tutorials/Java/Scala/0090__Scala_Comments.htm, accessed on July 12, 2022.
67. Scala – Comments: https://www.alphacodingskills.com/scala/scala-comments.php, accessed on July 12, 2022.
68. Comments in Scala: https://www.includehelp.com/scala/comments-in-scala.aspx, accessed on July 12, 2022.
69. IncludeHelp. (n.d.). Command Line Arguments in Scala: https://www.includehelp.com/scala/command-line-arguments-in-scala.aspx, accessed on July 12, 2022.
70. How to Read Environment Variables in Scala: https://www.oreilly.com/library/view/scala-cookbook/9781449340292/ch14s12.html, accessed on July 12, 2022.
71. Scope of Variables in Scala: https://www.geeksforgeeks.org/scope-of-variables-in-scala/, accessed on July 12, 2022.
72. Scala Variables - Variable Scopes, Field Variables & Method Parameters Example:https://www.journaldev.com/7581/scala-variables-variable-scopes-field-variables-method-parameters-example, accessed on July 12, 2022.
73. Scala Variables: https://data-flair.training/blogs/scala-variables/, accessed on July 12, 2022.
74. Scope of Scala Variables: https://www.includehelp.com/scala/scope-of-scala-variables.aspx, accessed on July 12, 2022.
75. Enumeration in Scala: https://www.geeksforgeeks.org/enumeration-in-scala/, accessed on July 12, 2022.

76. Enumerations in Scala: https://sodocumentation.net/scala/topic/1499/enumerations, accessed on July 12, 2022.
77. Enumerations in Scala: https://www.baeldung.com/scala/enumerations, accessed on July 12, 2022.
78. Scala Variables: https://data-flair.training/blogs/scala-variables/, accessed on July 12, 2022.
79. Enums: https://dotty.epfl.ch/docs/reference/enums/enums.html, accessed on July 13, 2022.
80. The Scala Programming Language: https://www.scala-lang.org/, accessed on July 13, 2022.
81. Scala Ranges: https://www.geeksforgeeks.org/scala-ranges/, accessed on July 13, 2022.
82. Scala Ranges: https://www.geeksforgeeks.org/scala-ranges/, accessed on July 13, 2022.
83. Scala Tutorial: https://www.tutorialspoint.com/scala/index.htm, accessed on July 13, 2022.
84. Scala Decision Making (if, if-else, nested if-else, if-else-if): https://www.geeksforgeeks.org/scala-decision-making-if-if-else-nested-if-else-if-else-if/, accessed on July 13, 2022.
85. DataFlair Team. (2019). Scala Control Structures – Comprehensive Guide. Retrieved from https://data-flair.training/blogs/scala-control-structures-comprehensive-guide/, accessed on July 15, 2023.
86. Scala – Loop Control Statements – while, do-while, for Loops: https://www.tutorialspoint.com/scala/scala_loop_types.htm, accessed on July 13, 2022.
87. Scala Loop Control Statements – while, do-while, for Loops: https://www.journaldev.com/7905/scala-loop-control-statements-while-do-while-for-loops, accessed on July 13, 2022.
88. Scala Object Oriented Programming: https://data-flair.training/blogs/scala-object-oriented-programming/, accessed on July 13, 2022.
89. Object-Oriented Programming in Scala: https://blog.knoldus.com/object-oriented-programming-in-scala/, accessed on July 13, 2022.
90. Class and Object in Scala: https://www.geeksforgeeks.org/class-and-object-in-scala/, accessed on July 13, 2022.
91. Scala Classes and Objects: https://www.baeldung.com/scala/classes-objects, accessed on July 13, 2022.
92. Inheritance from inner classes across path-dependent types: https://users.scala-lang.org/t/inheritance-from-inner-classes-across-path-dependent-types/7551, accessed on July 13, 2022.
93. Inner Class in Scala: How to Create Inner Class?: https://www.includehelp.com/scala/inner-class-in-scala-how-to-create-inner-class.aspx, accessed on July 13, 2022.
94. Inheritance in Scala: https://www.geeksforgeeks.org/inheritance-in-scala/, accessed on July 15, 2023.
95. Scala – Classes and Objects: https://www.tutorialspoint.com/scala/scala_classes_objects.htm, accessed on July 13, 2022.

96. Scala for Java Developers: Traits, Abstract Classes and Operators https://blog.birost.com/a?ID=01050-cb4ec8e8-8e99-4418-944f-2862e639143a, accessed on July 13, 2022.
97. Operators in Scala: https://www.geeksforgeeks.org/operators-in-scala/, accessed on July 13, 2022.
98. Scala – Operators: https://www.tutorialspoint.com/scala/scala_operators.htm, accessed on July 13, 2022.
99. Operators in Scala: https://www.datacamp.com/tutorial/operators-in-scala, accessed on July 13, 2022.
100. Operators Precedence in Scala: https://www.geeksforgeeks.org/operators-precedence-in-scala/, accessed on July 13, 2022.
101. Operators: https://docs.scala-lang.org/tour/operators.html, accessed on July 13, 2022.
102. Scala - Operator Precedence: https://www.alphacodingskills.com/scala/notes/scala-operators-precedence.php, accessed on July 14, 2022.
103. Scala – Operators: https://www.tutorialspoint.com/scala/scala_operators.htm, accessed on July 14, 2022.
104. Operator Precedence: https://riptutorial.com/scala/example/22543/operator-precedence, accessed on July 14, 2022.
105. Learn Scala from Scratch: https://www.educative.io/courses/learn-scala-from-scratch/g201WmJ1jyG, accessed on July 14, 2022.
106. Scala – Singleton and Companion Objects: https://www.geeksforgeeks.org/scala-singleton-and-companion-objects/, accessed on July 14, 2022.
107. Scala Singleton and Companion Object: https://www.javatpoint.com/scala-singleton-and-companion-object, accessed on July 14, 2022.
108. Scala Singleton Object Tutorial: Creating, Syntax and Uses: https://data-flair.training/blogs/scala-singleton-object/, accessed on July 14, 2022.
109. Traits vs. Abstract Classes in Scala: https://www.baeldung.com/scala/traits-vs-abstract-classes, accessed on July 14, 2022.
110. Scala Abstract Class: https://www.javatpoint.com/scala-abstract-class, accessed on July 15, 2022.
111. Scala Abstract Class: https://linuxhint.com/scala-abstract-class/, accessed on July 15, 2022.
112. Generic Classes: https://docs.scala-lang.org/tour/generic-classes.html, accessed on July 15, 2022.
113. Generic Classes in Scala: https://www.geeksforgeeks.org/generic-classes-in-scala/, accessed on July 15, 2022.
114. Scala Generics – Basics: https://www.baeldung.com/scala/generics-basics, accessed on July 15, 2022.
115. Scala Generic Classes and Variance: https://blog.knoldus.com/scala-generic-classes-and-variance/, accessed on July 15, 2022.
116. Access Modifiers in Scala: https://www.geeksforgeeks.org/access-modifiers-in-scala/, accessed on July 15, 2022.
117. Scala - Access Modifiers: https://www.tutorialspoint.com/scala/scala_access_modifiers.htm, accessed on July 15, 2022.

118. Scala Access Modifiers - Private, Protected, and Public: https://www.journaldev.com/7584/scala-access-modifiers-private-protected-and-public, accessed on July 15, 2023.
119. Traits vs. Abstract Classes in Scala: https://www.baeldung.com/scala/traits-vs-abstract-classes, accessed on July 15, 2022.
120. Scala Abstract Class: https://www.javatpoint.com/scala-abstract-class, accessed on July 15, 2022.
121. Scala Abstract Class: https://www.javatpoint.com/scala-abstract-class, accessed on July 15, 2022.
122. Scala Abstract Class: https://linuxhint.com/scala-abstract-class/, accessed on July 15, 2022.
123. Generic Classes: https://docs.scala-lang.org/tour/generic-classes.html, accessed on July 15, 2022.
124. Generic Classes in Scala: https://www.geeksforgeeks.org/generic-classes-in-scala/, accessed on July 16, 2022.
125. Scala Generics – Basics: https://www.baeldung.com/scala/generics-basics, accessed on July 16, 2022.
126. Scala Generic Classes and Variance: https://www.signifytechnology.com/blog/2019/04/scala-generic-classes-and-variance-by-ayush-hooda, accessed on July 16, 2022.
127. Scala Generic: https://www.educba.com/scala-generic/, accessed on July 16, 2022.
128. What are Generic Classes in Scala?: https://www.educative.io/answers/what-are-generic-classes-in-scala, accessed on July 16, 2022.
129. Scala Generic Classes and Variance: https://blog.knoldus.com/scala-generic-classes-and-variance/, accessed on July 16, 2022.
130. Access Modifiers in Scala: https://www.geeksforgeeks.org/access-modifiers-in-scala/, accessed on July 16, 2022.
131. Scala - Access Modifiers: https://www.tutorialspoint.com/scala/scala_access_modifiers.htm, accessed on July 16, 2022.
132. Scala Access Modifiers - Private, Protected, and Public: https://www.journaldev.com/7584/scala-access-modifiers-private-protected-and-public, accessed on July 17, 2022.
133. Scala Access Modifier: https://www.javatpoint.com/scala-access-modifier, accessed on July 17, 2022.
134. Scala Access Modifiers - Private, Protected & Public: https://data-flair.training/blogs/scala-access-modifiers/, accessed on July 17, 2022.
135. Scala Constructors: https://www.geeksforgeeks.org/scala-constructors/, accessed on July 17, 2022.
136. Scala Constructor: https://www.javatpoint.com/scala-constructor, accessed on July 17, 2022.
137. Classes. Scala Language Documentation: https://docs.scala-lang.org/overviews/scala-book/classes.html, accessed on July 17, 2022.
138. Scala Constructor – Syntax, Types of Constructors with Examples: https://data-flair.training/blogs/scala-constructor/, accessed on July 17, 2022.

139. Scala – Controlling Visibility of Constructor Fields: https://www.geeksforgeeks.org/scala-controlling-visibility-of-constructor-fields/, accessed on July 17, 2022.
140. How to control the visibility of Scala constructor fields: https://www.oreilly.com/library/view/scala-cookbook/9781449340292/ch04s03.html, accessed on July 17, 2022.
141. Scala – Strings: https://www.tutorialspoint.com/scala/scala_strings.htm, accessed on July 17, 2022.
142. Scala – String: https://www.geeksforgeeks.org/scala-string/, accessed on July 17, 2022.
143. Package Objects: https://docs.scala-lang.org/tour/package-objects.html, accessed on July 17, 2022.
144. Scala Import and Package Objects: https://www.baeldung.com/scala/package-import, accessed on July 18, 2022.
145. How to use package objects in Scala: https://www.oreilly.com/library/view/scala-cookbook/9781449340292/ch01s11.html, accessed on July 18, 2022.
146. Scala Package Object: https://linuxhint.com/scala-package-object/, accessed on July 18, 2022.
147. Scala – Functions: https://www.tutorialspoint.com/scala/scala_functions.htm#:~:text=A%20Scala%20method%20is%20a,object%2C%20is%20called%20a%20method, accessed on July 18, 2022.
148. Scala Functions and Methods: https://www.baeldung.com/scala/functions-methods, accessed on July 18, 2022.
149. Scala Primary Constructor: https://www.geeksforgeeks.org/scala-primary-constructor/, accessed on July 18, 2022.
150. Scala Auxiliary Constructor: https://www.geeksforgeeks.org/scala-auxiliary-constructor/, accessed on July 18, 2022.
151. Scala functions: https://www.javatpoint.com/scala-functions, accessed on July 18, 2022.
152. Scala exception handling: https://www.geeksforgeeks.org/scala-exception-handling/, accessed on July 19, 2022.
153. Scala exception handling: https://www.baeldung.com/scala/exception-handling, accessed on July 19, 2022.
154. Scala exception handling: https://www.tutorialspoint.com/scala/scala_exception_handling.htm, accessed on July 19, 2022.
155. Functional error handling: https://docs.scala-lang.org/overviews/scala-book/functional-error-handling.html, accessed on July 19, 2022.
156. Exception handling in Scala: https://blog.knoldus.com/exception-handling-in-scala/, accessed on July 20, 2022.
157. Scala Exception Handling: https://intellipaat.com/blog/tutorial/scala-tutorial/scala-exception-handling/, accessed on July 20, 2022.
158. State exploration of Scala actor programs: https://1library.net/document/q27l4xey-state-exploration-of-scala-actor-programs.html, accessed on July 20, 2022.
159. Is Scala worth learning?: https://careerkarma.com/blog/is-scala-worth-learning/, accessed on July 20, 2022.

160. What big companies use Scala?: https://datarootlabs.com/blog/big-companies-use-scala, accessed on July 20, 2022.
161. Scala vs. Java: https://www.simplilearn.com/scala-vs-java-article, accessed on July 20, 2022.
162. Learn Scala from scratch: https://www.educative.io/courses/learn-scala-from-scratch/qAoLO5npEj2, accessed on July 20, 2022.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P