Scala - Quick Guide

Scala - Overview

Scala, short for Scalable Language, is a hybrid functional programming language. It was created by Martin Odersky. Scala smoothly integrates the features of object-oriented and functional languages. Scala is compiled to run on the Java Virtual Machine. Many existing companies, who depend on Java for business critical applications, are turning to Scala to boost their development productivity, applications scalability and overall reliability.

Here we have presented a few points that makes Scala the first choice of application developers.

Scala is object-oriented

Scala is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes and traits which will be explained in subsequent chapters.

Classes are extended by **subclassing** and a flexible **mixin-based composition** mechanism as a clean replacement for multiple inheritance.

Scala is functional

Scala is also a functional language in the sense that every function is a value and every value is an object so ultimately every function is an object.

Scala provides a lightweight syntax for defining **anonymous functions**, it supports **higher-order functions**, it allows functions to be **nested**, and supports **currying**. These concepts will be explained in subsequent chapters.

Scala is statically typed

Scala, unlike some of the other statically typed languages (C, Pascal, Rust, etc.), does not expect you to provide redundant type information. You don't have to specify a type in most cases, and you certainly don't have to repeat it.

Scala runs on the JVM

Scala is compiled into Java Byte Code which is executed by the Java Virtual Machine (JVM). This means that Scala and Java have a common runtime platform. You can easily move from Java to Scala.

The Scala compiler compiles your Scala code into Java Byte Code, which can then be executed by the '**scala**' command. The '**scala**' command is similar to the **java** command, in that it executes your compiled Scala code.

Scala can Execute Java Code

Scala enables you to use all the classes of the Java SDK and also your own custom Java classes, or your favorite Java open source projects.

Scala can do Concurrent & Synchronize processing

Scala allows you to express general programming patterns in an effective way. It reduces the number of lines and helps the programmer to code in a type-safe way. It allows you to write codes in an immutable manner, which makes it easy to apply concurrency and parallelism (Synchronize).

Scala vs Java

Scala has a set of features that completely differ from Java. Some of these are −

- All types are objects
- Type inference
- Nested Functions
- Functions are objects
- Domain specific language (DSL) support
- Traits
- Closures
- Concurrency support inspired by Erlang

Scala Web Frameworks

Scala is being used everywhere and importantly in enterprise web applications. You can check a few of the most popular Scala web frameworks −

- The Lift Framework
- The Play framework
- The Bowler framework

Scala - Environment Setup

Scala can be installed on any UNIX flavored or Windows based system. Before you start installing Scala on your machine, you must have Java 1.8 or greater installed on your computer.

Follow the steps given below to install Scala.

Step 1: Verify Your Java Installation

First of all, you need to have Java Software Development Kit (SDK) installed on your system. To verify this, execute any of the following two commands depending on the platform you are working on.

If the Java installation has been done properly, then it will display the current version and specification of your Java installation. A sample output is given in the following table.

Platform	Command	Sample Output
Windows	Open Command Console and type − **\>java –version**	Java version "1.8.0_31" Java (TM) SE Run Time Environment (build 1.8.0_31-b31) Java Hotspot (TM) 64-bit Server VM (build 25.31-b07, mixed mode)
Linux	Open Command terminal and type − **$java –version**	Java version "1.8.0_31" Open JDK Runtime Environment (rhel-2.8.10.4.el6_4-x86_64) Open JDK 64-Bit Server VM (build 25.31-b07, mixed mode)

We assume that the readers of this tutorial have Java SDK version 1.8.0_31 installed on their system.

In case you do not have Java SDK, download its current version from http://www.oracle.com/technetwork/java/javase/downloads/index.html and install it.

Step 2: Set Your Java Environment

Set the environment variable JAVA_HOME to point to the base directory location where Java is installed on your machine. For example,

Sr.No	Platform & Description
1	**Windows** Set JAVA_HOME to C:\ProgramFiles\java\jdk1.7.0_60
2	**Linux** Export JAVA_HOME=/usr/local/java-current

Append the full path of Java compiler location to the System Path.

Sr.No	Platform & Description
1	**Windows** Append the String "C:\Program Files\Java\jdk1.7.0_60\bin" to the end of the system variable PATH.
2	**Linux** Export PATH=$PATH:$JAVA_HOME/bin/

Execute the command **java -version** from the command prompt as explained above.

Step 3: Install Scala

You can download Scala from http://www.scala-lang.org/downloads . At the time of writing this tutorial, I downloaded ‘scala-2.11.5-installer.jar’. Make sure you have admin privilege to proceed. Now, execute the following command at the command prompt −

Platform	Command & Output	Description
Windows	\>java –jar scala-2.11.5-installer.jar\>	This command will display an installation wizard, which will guide you to install Scala on your windows machine. During installation, it will ask for license agreement, simply accept it and further it will ask a path where Scala will be installed. I selected default given path *"C:\Program Files\Scala",* you can select a suitable path as per your convenience.
Linux	**Command** − $java –jar scala-2.9.0.1-installer.jar **Output** − Welcome to the installation of Scala 2.9.0.1! The homepage is at − http://Scala-lang.org/ press 1 to continue, 2 to quit, 3 to redisplay 1..... [Starting to unpack] [Processing package: Software Package Installation (1/1)] [Unpacking finished] [Console installation done]	During installation, it will ask for license agreement, to accept it type 1 and it will ask a path where Scala will be installed. I entered */usr/local/share,* you can select a suitable path as per your convenience.

Finally, open a new command prompt and type **Scala -version** and press Enter. You should see the following −

Platform	Command	Output
Windows	\>scala -version	Scala code runner version 2.11.5 -- Copyright 2002-2013, LAMP/EPFL
Linux	$scala -version	Scala code runner version 2.9.0.1 – Copyright 2002-2013, LAMP/EPFL

Scala - Basic Syntax

If you have a good understanding on Java, then it will be very easy for you to learn Scala. The biggest syntactic difference between Scala and Java is that the ';' line end character is optional.

When we consider a Scala program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods and instance variables mean.

- **Object** − Objects have states and behaviors. An object is an instance of a class. Example − A dog has states - color, name, breed as well as behaviors - wagging, barking, and eating.
- **Class** − A class can be defined as a template/blueprint that describes the behaviors/states that are related to the class.
- **Methods** − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Fields** − Each object has its unique set of instance variables, which are called fields. An object's state is created by the values assigned to these fields.
- **Closure** − A **closure** is a function, whose return value depends on the value of one or more variables declared outside this function.
- **Traits** − A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Traits are used to define object types by specifying the signature of the supported methods.

First Scala Program

We can execute a Scala program in two modes: one is **interactive mode** and another is **script mode**.

Interactive Mode

Open the command prompt and use the following command to open Scala.

```
\>scala
```

If Scala is installed in your system, the following output will be displayed −

```
Welcome to Scala version 2.9.0.1
Type in expressions to have them evaluated.
Type :help for more information.
```

Type the following text to the right of the Scala prompt and press the Enter key −

```
scala> println("Hello, Scala!");
```

It will produce the following result −

```
Hello, Scala!
```

Script Mode

Use the following instructions to write a Scala program in script mode. Open notepad and add the following code into it.

Example

```
object HelloWorld {
   /* This is my first java program.
   * This will print 'Hello World' as the output
   */
   def main(args: Array[String]) {
      println("Hello, world!") // prints Hello World
   }
}
```

Save the file as − **HelloWorld.scala**.

Open the command prompt window and go to the directory where the program file is saved. The '**scalac**' command is used to compile the Scala program and it will generate a few class files in the current directory. One of them will be called **HelloWorld.class**. This is a bytecode which will run on Java Virtual Machine (JVM) using '**scala**' command.

Use the following command to compile and execute your Scala program.

```
\> scalac HelloWorld.scala
\> scala HelloWorld
```

Output

```
Hello, World!
```

Basic Syntax

The following are the basic syntaxes and coding conventions in Scala programming.

- **Case Sensitivity** − Scala is case-sensitive, which means identifier **Hello** and **hello** would have different meaning in Scala.
- **Class Names** − For all class names, the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

  Example − class MyFirstScalaClass.

- **Method Names** − All method names should start with a Lower Case letter. If multiple words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

  Example − def myMethodName()

- **Program File Name** − Name of the program file should exactly match the object name. When saving the file you should save it using the object name (Remember Scala is case-sensitive) and append '**.scala**' to the end of the name. (If the file name and the object name do not match your program will not compile).

  Example − Assume 'HelloWorld' is the object name. Then the file should be saved as 'HelloWorld.scala'.

- **def main(args: Array[String])** − Scala program processing starts from the main() method which is a mandatory part of every Scala Program.

Scala Identifiers

All Scala components require names. Names used for objects, classes, variables and methods are called identifiers. A keyword cannot be used as an identifier and identifiers are case-sensitive. Scala supports four types of identifiers.

Alphanumeric Identifiers

An alphanumeric identifier starts with a letter or an underscore, which can be followed by further letters, digits, or underscores. The '$' character is a reserved keyword in Scala and should not be used in identifiers.

Following are **legal alphanumeric identifiers** −

```
age, salary, _value,  __1_value
```

Following are **illegal identifiers** −

```
$salary, 123abc, -salary
```

Operator Identifiers

An operator identifier consists of one or more operator characters. Operator characters are printable ASCII characters such as +, :, ?, ~ or #.

Following are legal operator identifiers −

```
+ ++ ::: <?> :>
```

The Scala compiler will internally "mangle" operator identifiers to turn them into legal Java identifiers with embedded $ characters. For instance, the identifier :-> would be represented internally as $colon$minus$greater.

Mixed Identifiers

A mixed identifier consists of an alphanumeric identifier, which is followed by an underscore and an operator identifier.

Following are legal mixed identifiers −

```
unary_+,  myvar_=
```

Here, unary_+ used as a method name defines a unary + operator and myvar_= used as method name defines an assignment operator (operator overloading).

Literal Identifiers

A literal identifier is an arbitrary string enclosed in back ticks (` . . . `).

Following are legal literal identifiers −

```
`x` `<clinit>` `yield`
```

Scala Keywords

The following list shows the reserved words in Scala. These reserved words may not be used as constant or variable or any other identifier names.

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

Comments in Scala

Scala supports single-line and multi-line comments very similar to Java. Multi-line comments may be nested, but are required to be properly nested. All characters available inside any comment are ignored by Scala compiler.

```
object HelloWorld {
   /* This is my first java program.
    * This will print 'Hello World' as the output
    * This is an example of multi-line comments.
    */
   def main(args: Array[String]) {
      // Prints Hello World
      // This is also an example of single line comment.
      println("Hello, world!")
   }
}
```

Blank Lines and Whitespace

A line containing only whitespace, possibly with a comment, is known as a blank line, and Scala totally ignores it. Tokens may be separated by whitespace characters and/or comments.

Newline Characters

Scala is a line-oriented language where statements may be terminated by semicolons (;) or newlines. A semicolon at the end of a statement is usually optional. You can type one if you want but you don't have to if the statement appears by itself on a single line. On the other hand, a semicolon is required if you write multiple statements on a single line. Below syntax is the usage of multiple statements.

```
val s = "hello"; println(s)
```

Scala Packages

A package is a named module of code. For example, the Lift utility package is net.liftweb.util. The package declaration is the first non-comment line in the source file as follows −

```
package com.liftcode.stuff
```

Scala packages can be imported so that they can be referenced in the current compilation scope. The following statement imports the contents of the scala.xml package −

```
import scala.xml._
```

You can import a single class and object, for example, HashMap from the scala.collection.mutable package −

```
import scala.collection.mutable.HashMap
```

You can import more than one class or object from a single package, for example, TreeMap and TreeSet from the scala.collection.immutable package −

```
import scala.collection.immutable.{TreeMap, TreeSet}
```

Apply Dynamic

A marker trait that enables dynamic invocations. Instances x of this trait allow method invocations x.meth(args) for arbitrary method names meth and argument lists args as well as field accesses x.field for arbitrary field namesfield. This feature is introduced in Scala-2.10.

If a call is not natively supported by x (i.e. if type checking fails), it is rewritten according to the following rules −

```
foo.method("blah") ~~> foo.applyDynamic("method")("blah")
foo.method(x = "blah") ~~> foo.applyDynamicNamed("method")(("x", "blah"))
foo.method(x = 1, 2) ~~> foo.applyDynamicNamed("method")(("x", 1), ("", 2))
foo.field ~~> foo.selectDynamic("field")
foo.varia = 10 ~~> foo.updateDynamic("varia")(10)
foo.arr(10) = 13 ~~> foo.selectDynamic("arr").update(10, 13)
foo.arr(10) ~~> foo.applyDynamic("arr")(10)
```

Scala - Data Types

Scala has all the same data types as Java, with the same memory footprint and precision. Following is the table giving details about all the data types available in Scala −

Sr.No	Data Type & Description
1	**Byte** 8 bit signed value. Range from -128 to 127
2	**Short** 16 bit signed value. Range -32768 to 32767
3	**Int** 32 bit signed value. Range -2147483648 to 2147483647
4	**Long** 64 bit signed value. -9223372036854775808 to 9223372036854775807
5	**Float** 32 bit IEEE 754 single-precision float
6	**Double** 64 bit IEEE 754 double-precision float
7	**Char** 16 bit unsigned Unicode character. Range from U+0000 to U+FFFF
8	**String** A sequence of Chars
9	**Boolean** Either the literal true or the literal false
10	**Unit** Corresponds to no value
11	**Null** null or empty reference

12	**Nothing** The subtype of every other type; includes no values
13	**Any** The supertype of any type; any object is of type *Any*
14	**AnyRef** The supertype of any reference type

All the data types listed above are objects. There are no primitive types like in Java. This means that you can call methods on an Int, Long, etc.

Scala Basic Literals

The rules Scala uses for literals are simple and intuitive. This section explains all basic Scala Literals.

Integral Literals

Integer literals are usually of type Int, or of type Long when followed by a L or l suffix. Here are some integer literals −

```
0
035
21
0xFFFFFFFF
0777L
```

Floating Point Literal

Floating point literals are of type Float when followed by a floating point type suffix F or f, and are of type Double otherwise. Here are some floating point literals −

```
0.0
1e30f
3.14159f
1.0e100
.1
```

Boolean Literals

The Boolean literals **true** and **false** are members of type Boolean.

Symbol Literals

A symbol literal 'x is a shorthand for the expression **scala.Symbol("x")**. Symbol is a case class, which is defined as follows.

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

Character Literals

A character literal is a single character enclosed in quotes. The character is either a printable Unicode character or is described by an escape sequence. Here are some character literals −

```
'a'
'\u0041'
'\n'
'\t'
```

String Literals

A string literal is a sequence of characters in double quotes. The characters are either printable Unicode character or are described by escape sequences. Here are some string literals −

```
"Hello,\nWorld!"
"This string contains a \" character."
```

Multi-Line Strings

A multi-line string literal is a sequence of characters enclosed in triple quotes """ ... """. The sequence of characters is arbitrary, except that it may contain three or more consecutive quote characters only at the very end.

Characters must not necessarily be printable; newlines or other control characters are also permitted. Here is a multi-line string literal −

```
"""the present string
spans three
lines."""
```

Null Values

The null value is of type **scala.Null** and is thus compatible with every reference type. It denotes a reference value which refers to a special "null" object.

Escape Sequences

The following escape sequences are recognized in character and string literals.

Escape Sequences	Unicode	Description
\b	\u0008	backspace BS
\t	\u0009	horizontal tab HT
\n	\u000c	formfeed FF
\f	\u000c	formfeed FF
\r	\u000d	carriage return CR
\"	\u0022	double quote "
\'	\u0027	single quote .
\\	\u005c	backslash \

A character with Unicode between 0 and 255 may also be represented by an octal escape, i.e., a backslash '\' followed by a sequence of up to three octal characters. Following is the example to show few escape sequence characters −

Example

```
object Test {
   def main(args: Array[String]) {
      println("Hello\tWorld\n\n" );
   }
}
```

When the above code is compiled and executed, it produces the following result −

Output

```
Hello   World
```

Scala - Variables

Variables are nothing but reserved memory locations to store values. This means that when you create a variable, you reserve some space in memory.

Based on the data type of a variable, the compiler allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

Variable Declaration

Scala has a different syntax for declaring variables. They can be defined as value, i.e., constant or a variable. Here, myVar is declared using the keyword var. It is a variable that can change value and this is called **mutable variable**. Following is the syntax to define a variable using **var** keyword −

Syntax

```
var myVar : String = "Foo"
```

Here, myVal is declared using the keyword val. This means that it is a variable that cannot be changed and this is called **immutable variable**. Following is the syntax to define a variable using val keyword −

Syntax

```
val myVal : String = "Foo"
```

Variable Data Types

The type of a variable is specified after the variable name and before equals sign. You can define any type of Scala variable by mentioning its data type as follows −

Syntax

```
val or val VariableName : DataType = [Initial Value]
```

If you do not assign any initial value to a variable, then it is valid as follows −

Syntax

```
var myVar :Int;
val myVal :String;
```

Variable Type Inference

When you assign an initial value to a variable, the Scala compiler can figure out the type of the variable based on the value assigned to it. This is called variable type inference. Therefore, you could write these variable declarations like this −

Syntax

```
var myVar = 10;
val myVal = "Hello, Scala!";
```

Here, by default, myVar will be Int type and myVal will become String type variable.

Multiple assignments

Scala supports multiple assignments. If a code block or method returns a Tuple (**Tuple** − Holds collection of Objects of different types), the Tuple can be assigned to a val variable. [**Note** − We will study Tuples in subsequent chapters.]

Syntax

```
val (myVar1: Int, myVar2: String) = Pair(40, "Foo")
```

And the type inference gets it right −

Syntax

```
val (myVar1, myVar2) = Pair(40, "Foo")
```

Example Program

The following is an example program that explains the process of variable declaration in Scala. This program declares four variables — two variables are defined with type declaration and remaining two are without type declaration.

Example

```
object Demo {
   def main(args: Array[String]) {
      var myVar :Int = 10;
      val myVal :String = "Hello Scala with datatype declaration.";
      var myVar1 = 20;
      val myVal1 = "Hello Scala new without datatype declaration.";

      println(myVar); println(myVal); println(myVar1);
      println(myVal1);
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
10
Hello Scala with datatype declaration.
20
Hello Scala without datatype declaration.
```

Variable Scope

Variables in Scala can have three different scopes depending on the place where they are being used. They can exist as fields, as method parameters and as local variables. Below are the details about each type of scope.

Fields

Fields are variables that belong to an object. The fields are accessible from inside every method in the object. Fields can also be accessible outside the object depending on what access modifiers the field is declared with. Object fields can be both mutable and immutable types and can be defined using either **var** or **val**.

Method Parameters

Method parameters are variables, which are used to pass the value inside a method, when the method is called. Method parameters are only accessible from inside the method but the objects passed in may be accessible from the outside, if you have a reference to the object from outside the method. Method parameters are always immutable which are defined by **val** keyword.

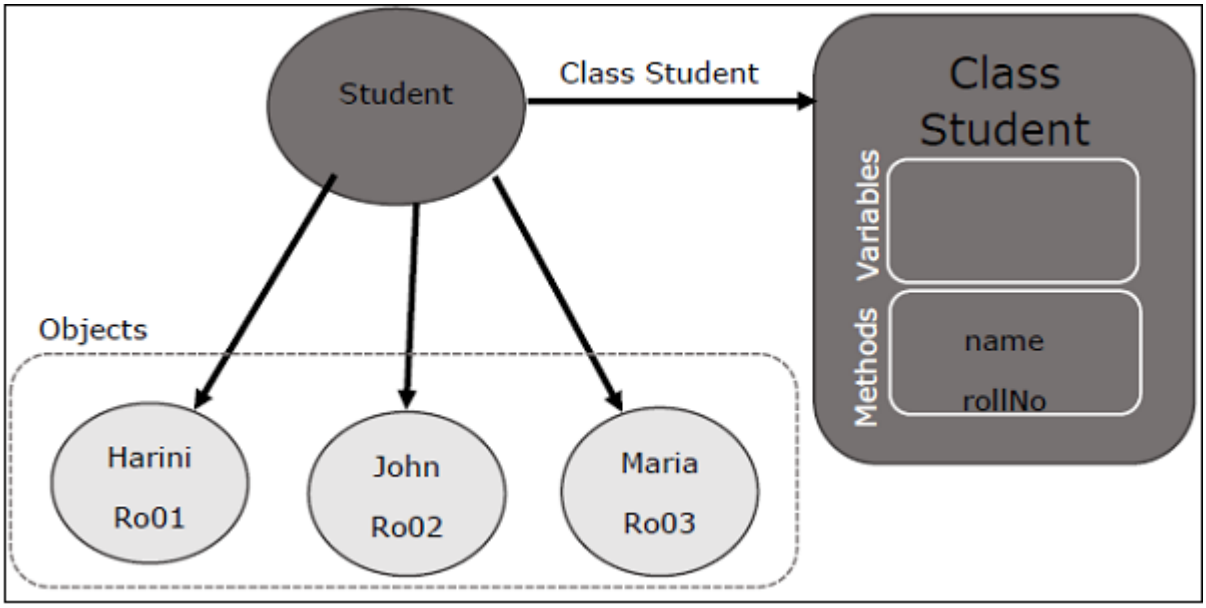
Local Variables

Local variables are variables declared inside a method. Local variables are only accessible from inside the method, but the objects you create may escape the method if you return them from the method. Local variables can be both mutable and immutable types and can be defined using either **var** or **val**.

Scala - Classes & Objects

This chapter takes you through how to use classes and objects in Scala programming. A class is a blueprint for objects. Once you define a class, you can create objects from the class blueprint with the keyword **new**. Through the object you can use all functionalities of the defined class.

The following diagram demonstrates the class and object by taking an example of class student, which contains the member variables (name and roll no) and member methods (setName() and setRollNo()). Finally all are members of the class. Class is a blue print and objects are real here. In the following diagram, Student is a class and Harini, John, and Maria are the objects of Student class, those are having name and roll-number.



Basic Class

Following is a simple syntax to define a basic class in Scala. This class defines two variables **x** and **y** and a method: **move**, which does not return a value. Class variables are called, fields of the class and methods are called class methods.

The class name works as a class constructor which can take a number of parameters. The above code defines two constructor arguments, **xc** and **yc**; they are both visible in the whole body of the class.

Syntax

```
class Point(xc: Int, yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("Point x location : " + x);
      println ("Point y location : " + y);
   }
}
```

As mentioned earlier in this chapter, you can create objects using a keyword **new** and then you can access class fields and methods as shown below in the example −

Example

```
import java.io._

class Point(val xc: Int, val yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("Point x location : " + x);
      println ("Point y location : " + y);
   }
}

object Demo {
   def main(args: Array[String]) {
      val pt = new Point(10, 20);

      // Move to a new location
      pt.move(10, 10);
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Point x location : 20
Point y location : 30
```

Extending a Class

You can extend a base Scala class and you can design an inherited class in the same way you do it in Java (use **extends** key word), but there are two restrictions: method overriding requires the **override** keyword, and only the **primary** constructor can pass parameters to the base constructor. Let us extend our above class and add one more class method.

Example

Let us take an example of two classes Point class (as same example as above) and Location class is inherited class using extends keyword. Such an '**extends**' clause has two effects: it makes Location class inherit all non-private members from Point class, and it makes the type *Location* a subtype of the type *Point* class. So here the Point class is called **superclass** and the class *Location* is called **subclass**. Extending a class and inheriting all the features of a parent class is called **inheritance** but Scala allows the inheritance from just one class only.

Note − Methods move() method in Point class and **move() method in Location class** do not override the corresponding definitions of move since they are different definitions (for example, the former take two arguments while the latter take three arguments).

Try the following example program to implement inheritance.

```
import java.io._

class Point(val xc: Int, val yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("Point x location : " + x);
      println ("Point y location : " + y);
   }
}

class Location(override val xc: Int, override val yc: Int,
   val zc :Int) extends Point(xc, yc){
   var z: Int = zc

   def move(dx: Int, dy: Int, dz: Int) {
      x = x + dx
      y = y + dy
      z = z + dz
      println ("Point x location : " + x);
      println ("Point y location : " + y);
      println ("Point z location : " + z);
   }
}

object Demo {
   def main(args: Array[String]) {
      val loc = new Location(10, 20, 15);

      // Move to a new location
      loc.move(10, 10, 5);
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Point x location : 20
Point y location : 30
Point z location : 20
```

Implicit Classes

Implicit classes allow implicit conversations with class's primary constructor when the class is in scope. Implicit class is a class marked with 'implicit' keyword. This feature is introduced in Scala 2.10.

Syntax − The following is the syntax for implicit classes. Here implicit class is always in the object scope where all method definitions are allowed because implicit class cannot be a top level class.

Syntax

```
object <object name> {
   implicit class <class name>(<Variable>: Data type) {
      def <method>(): Unit =
   }
}
```

Example

Let us take an example of an implicit class named **IntTimes** with the method times(). It means the times () contain a loop transaction that will execute the given statement in number of times that we give. Let us assume the given statement is "4 times println ("Hello")" means the println (""Hello") statement will execute 4 times.

The following is the program for the given example. In this example two object classes are used (Run and Demo) so that we have to save those two classes in different files with their respective names as follows.

Run.scala − Save the following program in Run.scala.

```
object Run {
   implicit class IntTimes(x: Int) {
      def times [A](f: =>A): Unit = {
         def loop(current: Int): Unit =
```

```
         if(current > 0){
            f
            loop(current - 1)
         }
         loop(x)
      }
   }
}
```

Demo.scala − Save the following program in Demo.scala.

```
import Run._

object Demo {
   def main(args: Array[String]) {
      4 times println("hello")
   }
}
```

The following commands are used to compile and execute these two programs.

Command

```
\>scalac Run.scala
\>scalac Demo.scala
\>scala Demo
```

Output

```
Hello
Hello
Hello
Hello
```

Note −

- Implicit classes must be defined inside another class/object/trait (not in top level).
- Implicit classes may only take one non –implicit argument in their constructor.
- Implicit classes may not be any method, member or object in scope with the same name as the implicit class.

Singleton Objects

Scala is more object-oriented than Java because in Scala, we cannot have static members. Instead, Scala has **singleton objects**. A singleton is a class that can have only one instance, i.e., Object. You create singleton using the keyword **object** instead of class keyword. Since you can't instantiate a

singleton object, you can't pass parameters to the primary constructor. You already have seen all the examples using singleton objects where you called Scala's main method.

Following is the same example program to implement singleton.

Example

```
import java.io._

class Point(val xc: Int, val yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
   }
}

object Demo {
   def main(args: Array[String]) {
      val point = new Point(10, 20)
      printPoint

      def printPoint{
         println ("Point x location : " + point.x);
         println ("Point y location : " + point.y);
      }
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Point x location : 10
Point y location : 20
```

Scala - Access Modifiers

This chapter takes you through the Scala access modifiers. Members of packages, classes or objects can be labeled with the access modifiers private and protected, and if we are not using either of these two keywords, then access will be assumed as public. These modifiers restrict accesses to the members to certain regions of code. To use an access modifier, you include its keyword in the definition of members of package, class or object as we will see in the following section.

Private Members

A private member is visible only inside the class or object that contains the member definition.

Following is the example code snippet to explain Private member −

Example

```
class Outer {
   class Inner {
      private def f() { println("f") }

      class InnerMost {
         f() // OK
      }
   }
   (new Inner).f() // Error: f is not accessible
}
```

In Scala, the access (new Inner). f() is illegal because f is declared private in Inner and the access is not from within class Inner. By contrast, the first access to f in class Innermost is OK, because that access is contained in the body of class Inner. Java would permit both accesses because it lets an outer class access private members of its inner classes.

Protected Members

A protected member is only accessible from subclasses of the class in which the member is defined.

Following is the example code snippet to explain protected member −

Example

```
package p {
   class Super {
      protected def f() { println("f") }
   }

   class Sub extends Super {
      f()
   }

   class Other {
      (new Super).f() // Error: f is not accessible
```

```
   }
}
```

The access to f in class Sub is OK because f is declared protected in 'Super' class and 'Sub' class is a subclass of Super. By contrast the access to f in 'Other' class is not permitted, because class 'Other' does not inherit from class 'Super'. In Java, the latter access would be still permitted because 'Other' class is in the same package as 'Sub' class.

Public Members

Unlike private and protected members, it is not required to specify Public keyword for Public members. There is no explicit modifier for public members. Such members can be accessed from anywhere.

Following is the example code snippet to explain public member −

Example

```
class Outer {
   class Inner {
      def f() { println("f") }

      class InnerMost {
         f() // OK
      }
   }
   (new Inner).f() // OK because now f() is public
}
```

Scope of Protection

Access modifiers in Scala can be augmented with qualifiers. A modifier of the form private[X] or protected[X] means that access is private or protected "up to" X, where X designates some enclosing package, class or singleton object.

Consider the following example −

Example

```
package society {
   package professional {
      class Executive {
         private[professional] var workDetails = null
         private[society] var friends = null
         private[this] var secrets = null

         def help(another : Executive) {
            println(another.workDetails)
            println(another.secrets) //ERROR
         }
```

```
        }
    }
}
```

Note − the following points from the above example −

- Variable workDetails will be accessible to any class within the enclosing package professional.
- Variable friends will be accessible to any class within the enclosing package society.
- Variable secrets will be accessible only on the implicit object within instance methods (this).

Scala - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. Scala is rich in built-in operators and provides the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

The following arithmetic operators are supported by Scala language. For example, let us assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus operator finds the remainder after division of one number by another	B % A will give 0

Relational Operators

The following relational operators are supported by Scala language. For example let us assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

The following logical operators are supported by Scala language. For example, assume variable A holds 1 and variable B holds 0, then −

Show Examples

Operator	Description	Example
&&	It is called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	It is called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.
!	It is called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit by bit operation. The truth tables for &, |, and ^ are as follows −

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows −

```
A = 0011 1100
B = 0000 1101
-----------------------
A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011
```

The Bitwise operators supported by Scala language is listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

Show Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61, which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The bit positions of the left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The Bit positions of the left operand value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

Assignment Operators

There are following assignment operators supported by Scala language −

Show Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Operators Precedence in Scala

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Take a look at the following table. Operators with the highest precedence appear at the top of the table and those with the lowest precedence appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

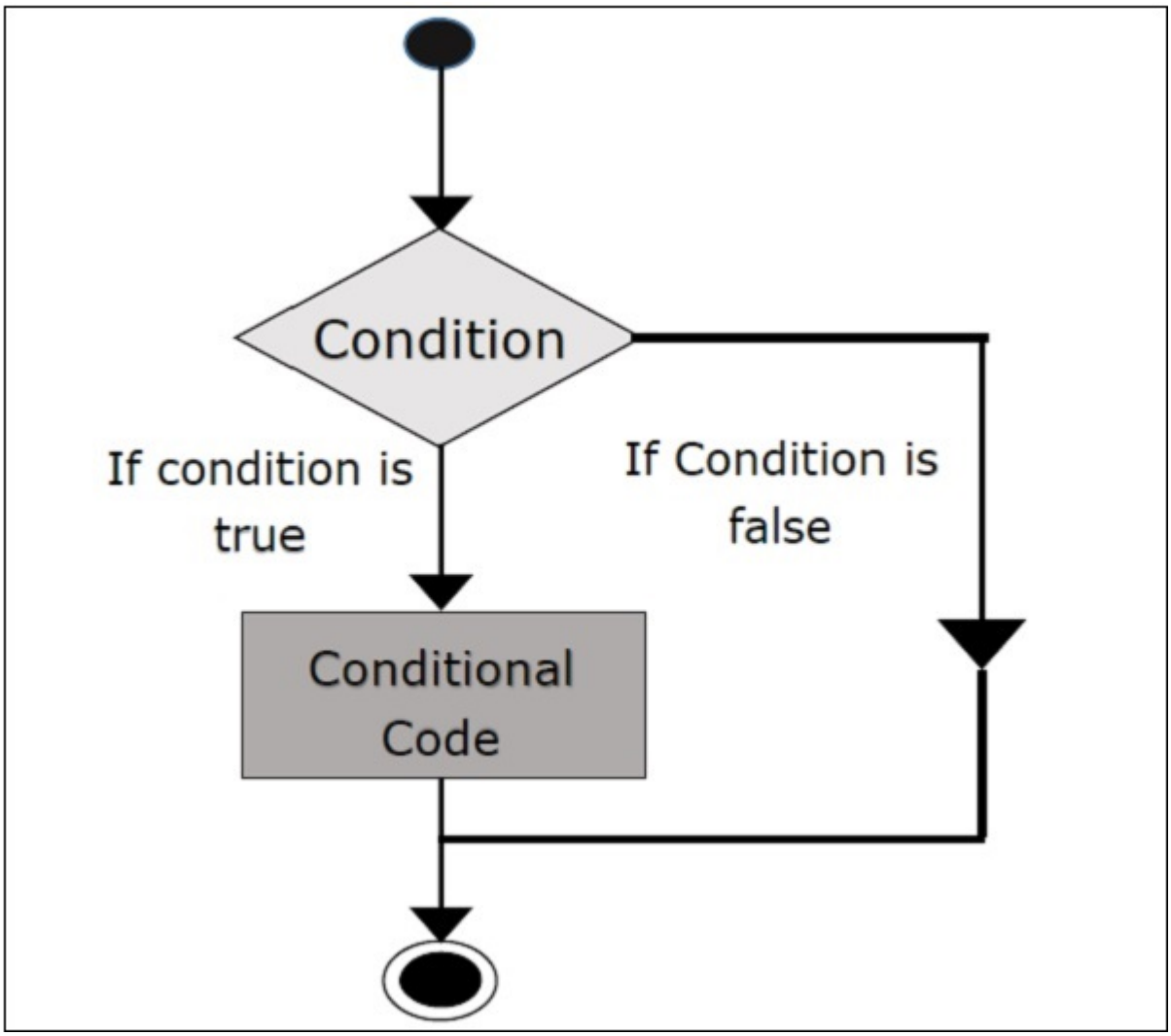
Category	Operator	Associativity
Postfix	() []	Left to right
Unary	! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Scala - IF ELSE Statements

This chapter takes you through the conditional construction statements in Scala programming. Following is the general form of a typical decision making IF...ELSE structure found in most of the programming languages.

Flow Chart

The following is a flow chart diagram for conditional statement.



if Statement

'if' statement consists of a Boolean expression followed by one or more statements.

Syntax

The syntax of an 'if' statement is as follows.

```
if(Boolean_expression) {
   // Statements will execute if the Boolean expression is true
}
```

If the Boolean expression evaluates to true then the block of code inside the 'if' expression will be executed. If not, the first set of code after the end of the 'if' expression (after the closing curly brace) will be executed.

Try the following example program to understand conditional expressions (if expression) in Scala Programming Language.

Example

```
object Demo {
   def main(args: Array[String]) {
      var x = 10;

      if( x < 20 ){
         println("This is if statement");
      }
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
This is if statement
```

If-else Statement

An 'if' statement can be followed by an optional *else* statement, which executes when the Boolean expression is false.

Syntax

The syntax of a if...else is −

```
if(Boolean_expression){
   //Executes when the Boolean expression is true
} else{
   //Executes when the Boolean expression is false
}
```

Try the following example program to understand conditional statements (if- else statement) in Scala Programming Language.

Example

```
object Demo {
   def main(args: Array[String]) {
      var x = 30;
```

```
      if( x < 20 ){
         println("This is if statement");
      } else {
         println("This is else statement");
      }
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
This is else statement
```

If-else-if-else Statement

An 'if' statement can be followed by an optional '*else if...else*' statement, which is very useful to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

- An 'if' can have zero or one else's and it must come after any else if's.
- An 'if' can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.

Syntax

The following is the syntax of an ‘if...else if...else’ is as follows −

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
} else if(Boolean_expression 2){
   //Executes when the Boolean expression 2 is true
} else if(Boolean_expression 3){
   //Executes when the Boolean expression 3 is true
} else {
   //Executes when the none of the above condition is true.
}
```

Try the following example program to understand conditional statements (if- else- if- else statement) in Scala Programming Language.

Example

```
object Demo {
   def main(args: Array[String]) {
      var x = 30;

      if( x == 10 ){
         println("Value of X is 10");
      } else if( x == 20 ){
         println("Value of X is 20");
      } else if( x == 30 ){
         println("Value of X is 30");
      } else{
         println("This is else statement");
      }
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Value of X is 30
```

Nested if-else Statement

It is always legal to nest **if-else** statements, which means you can use one **if** or **else-if** statement inside another **if** or **else-if** statement.

Syntax

The syntax for a nested if-else is as follows −

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true

   if(Boolean_expression 2){
```

```
      //Executes when the Boolean expression 2 is true
   }
}
```

Try the following example program to understand conditional statements (nested- if statement) in Scala Programming Language.

Example

```
object Demo {
   def main(args: Array[String]) {
      var x = 30;
      var y = 10;

      if( x == 30 ){
         if( y == 10 ){
            println("X = 30 and Y = 10");
         }
      }
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
X = 30 and Y = 10
```

Scala - Loop Statements

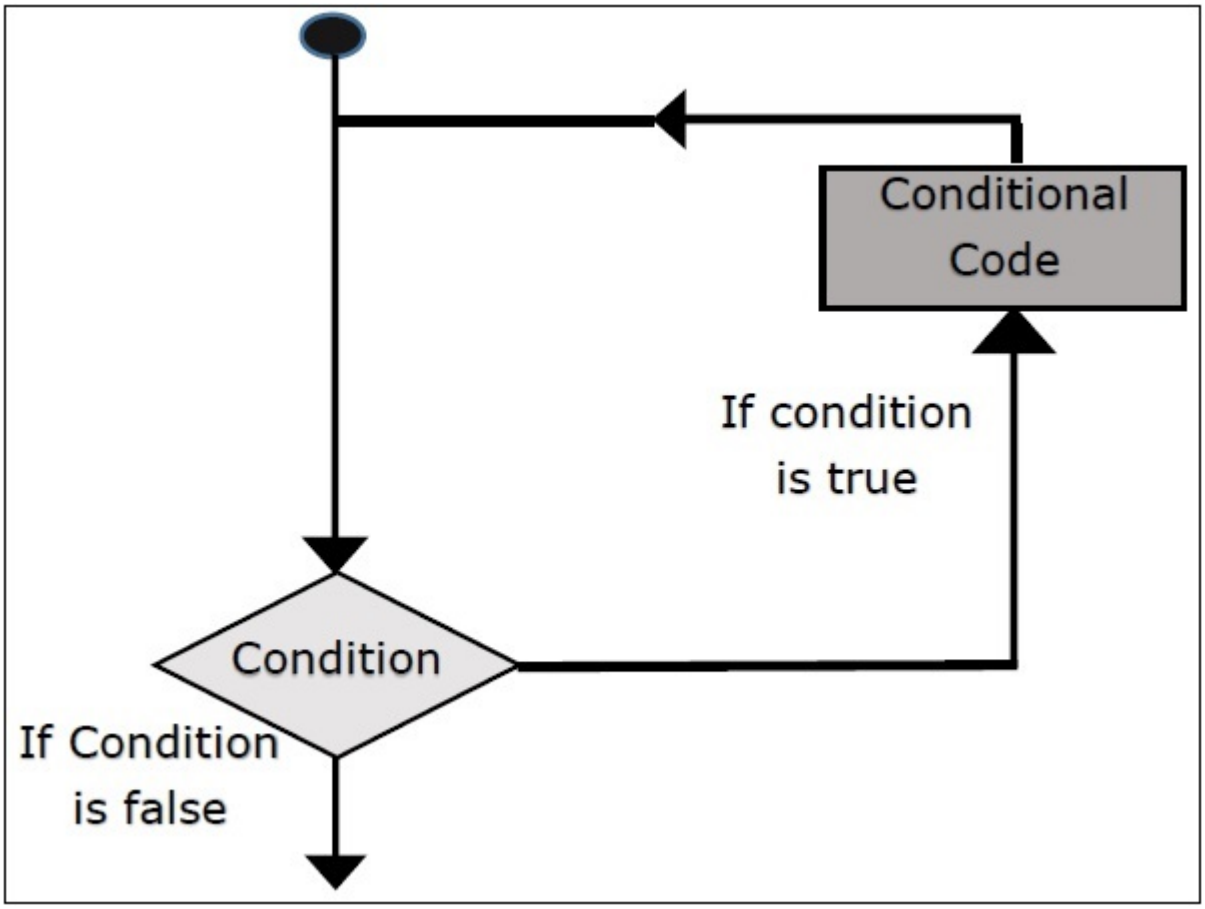
This chapter takes you through the loop control structures in Scala programming languages.

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −

Flow Chart



Scala programming language provides the following types of loops to handle looping requirements. Click the following links in the table to check their detail.

Sr.No	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	do-while loop Like a while statement, except that it tests the condition at the end of the loop body.
3	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. As such Scala does not support **break** or **continue** statement like Java does but starting from Scala version 2.8, there is a way to break the loops. Click the following links to check the detail.

Sr.No	Control Statement & Description
1	break statement Terminates the **loop** statement and transfers execution to the statement immediately following the loop.

The infinite Loop

A loop becomes an infinite loop if a condition never becomes false. If you are using Scala, the **while** loop is the best way to implement infinite loop.

The following program implements infinite loop.

Example

```
object Demo {
   def main(args: Array[String]) {
      var a = 10;

      // An infinite loop.
      while( true ){
         println( "Value of a: " + a );
      }
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

If you will execute above code, it will go in infinite loop which you can terminate by pressing Ctrl + C keys.

```
Value of a: 10
Value of a: 10
Value of a: 10
Value of a: 10
…….
```

Scala - Functions

A function is a group of statements that perform a task. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically, the division usually is so that each function performs a specific task.

Scala has both functions and methods and we use the terms method and function interchangeably with a minor difference. A Scala method is a part of a class which has a name, a signature, optionally some annotations, and some bytecode where as a function in Scala is a complete object which can be assigned to a variable. In other words, a function, which is defined as a member of some object, is called a method.

A function definition can appear anywhere in a source file and Scala permits nested function definitions, that is, function definitions inside other function definitions. Most important point to note is that Scala function's name can have characters like +, ++, ~, &,-, --, \, /, :, etc.

Function Declarations

A Scala function declaration has the following form −

```
def functionName ([list of parameters]) : [return type]
```

Methods are implicitly declared *abstract* if you don’t use the equals sign and the method body.

Function Definitions

A Scala function definition has the following form −

Syntax

```
def functionName ([list of parameters]) : [return type] = {
   function body
   return [expr]
}
```

Here, **return type** could be any valid Scala data type and **list of parameters** will be a list of variables separated by comma and list of parameters and return type are optional. Very similar to Java, a **return** statement can be used along with an expression in case function returns a value. Following is the function which will add two integers and return their sum −

Syntax

```
object add {
   def addInt( a:Int, b:Int ) : Int = {
      var sum:Int = 0
      sum = a + b
      return sum
   }
}
```

A function, that does not return anything can return a **Unit** that is equivalent to **void** in Java and indicates that function does not return anything. The functions which do not return anything in Scala, they are called procedures.

Syntax

Here is the syntax −

```
object Hello{
   def printMe( ) : Unit = {
      println("Hello, Scala!")
   }
}
```

Calling Functions

Scala provides a number of syntactic variations for invoking methods. Following is the standard way to call a method −

```
functionName( list of parameters )
```

If a function is being called using an instance of the object, then we would use dot notation similar to Java as follows −

```
[instance.]functionName( list of parameters )
```

Try the following example program to define and then call the same function.

Example

```
object Demo {
   def main(args: Array[String]) {
      println( "Returned Value : " + addInt(5,7) );
   }

   def addInt( a:Int, b:Int ) : Int = {
```

```
      var sum:Int = 0
      sum = a + b

      return sum
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Returned Value : 12
```

Scala functions are the heart of Scala programming and that's why Scala is assumed as a functional programming language. Following are few important concepts related to Scala functions which should be understood by a Scala programmer.

Functions Call-by-Name	Functions with Named Arguments
Function with Variable Arguments	Recursion Functions
Default Parameter Values	Higher-Order Functions
Nested Functions	Anonymous Functions
Partially Applied Functions	Currying Functions

Scala - Closures

A **closure** is a function, whose return value depends on the value of one or more variables declared outside this function.

The following piece of code with anonymous function.

```
val multiplier = (i:Int) => i * 10
```

Here the only variable used in the function body, i * 10 , is i, which is defined as a parameter to the function. Try the following code −

```
val multiplier = (i:Int) => i * factor
```

There are two free variables in multiplier: **i** and **factor**. One of them, i, is a formal parameter to the function. Hence, it is bound to a new value each time multiplier is called. However, **factor** is not a formal parameter, then what is this? Let us add one more line of code.

```
var factor = 3
val multiplier = (i:Int) => i * factor
```

Now **factor** has a reference to a variable outside the function but in the enclosing scope. The function references **factor** and reads its current value each time. If a function has no external references, then it is trivially closed over itself. No external context is required.

Try the following example program.

Example

```
object Demo {
   def main(args: Array[String]) {
      println( "multiplier(1) value = " +  multiplier(1) )
      println( "multiplier(2) value = " +  multiplier(2) )
   }
   var factor = 3
   val multiplier = (i:Int) => i * factor
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
multiplier(1) value = 3
multiplier(2) value = 6
```

Scala - Strings

This chapter takes you through the Scala Strings. In Scala, as in Java, a string is an immutable object, that is, an object that cannot be modified. On the other hand, objects that can be modified, like arrays, are

called mutable objects. Strings are very useful objects, in the rest of this section, we present important methods of **java.lang.String** class.

Creating a String

The following code can be used to create a String −

```
var greeting = "Hello world!";

or

var greeting:String = "Hello world!";
```

Whenever compiler encounters a string literal in the code, it creates a String object with its value, in this case, “Hello world!”. String keyword can also be given in alternate declaration as shown above.

Try the following example program.

Example

```
object Demo {
   val greeting: String = "Hello, world!"

   def main(args: Array[String]) {
      println( greeting )
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Hello, world!
```

As mentioned earlier, String class is immutable. String object once created cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters then use String Builder Class available in Scala!.

String Length

Methods used to obtain information about an object are known as accessor methods. One accessor method that can be used with strings is the length() method, which returns the number of characters contained in the string object.

Use the following code segment to find the length of a string −

Example

```
object Demo {
   def main(args: Array[String]) {
      var palindrome = "Dot saw I was Tod";
      var len = palindrome.length();

      println( "String Length is : " + len );
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
String Length is : 17
```

Concatenating Strings

The String class includes a method for concatenating two strings −

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in −

```
"My name is ".concat("Zara");
```

Strings are more commonly concatenated with the + operator, as in −

```
"Hello," + " world" + "!"
```

Which results in −

```
"Hello, world!"
```

The following lines of code to find string length.

Example

```
object Demo {
   def main(args: Array[String]) {
      var str1 = "Dot saw I was ";
      var str2 =  "Tod";

      println("Dot " + str1 + str2);
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Dot Dot saw I was Tod
```

Creating Format Strings

You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

Try the following example program, which makes use of printf() method −

Example

```
object Demo {
   def main(args: Array[String]) {
      var floatVar = 12.456
      var intVar = 2000
      var stringVar = "Hello, Scala!"

      var fs = printf("The value of the float variable is " + "%f, while the value of t

      println(fs)
```

```
    }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
The value of the float variable is 12.456000,
while the value of the integer variable is 2000,
and the string is Hello, Scala!()
```

String Interpolation

String Interpolation is the new way to create Strings in Scala programming language. This feature supports the versions of Scala-2.10 and later. String Interpolation: The mechanism to embed variable references directly in process string literal.

There are three types (interpolators) of implementations in String Interpolation.

The ‘s’ String Interpolator

The literal ‘s’ allows the usage of variable directly in processing a string, when you prepend ‘s’ to it. Any String variable with in a scope that can be used with in a String. The following are the different usages of ‘s’ String interpolator.

The following example code snippet for the implementation of ‘s’ interpolator in appending String variable ($name) to a normal String (Hello) in println statement.

```
val name = “James”
println(s “Hello, $name”) //output: Hello, James
```

String interpolater can also process arbitrary expressions. The following code snippet for Processing a String (1 + 1) with arbitrary expression (${1 + 1}) using ‘s’ String interpolator. Any arbitrary expression can be embedded in ‘${}’.

```
println(s “1 + 1 = ${1 + 1}”) //output: 1 + 1 = 2
```

Try the following example program of implementing ‘s’ interpolator.

Example

```
object Demo {
   def main(args: Array[String]) {
      val name = "James"

      println(s"Hello, $name")
      println(s"1 + 1 = ${1 + 1}")
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Hello, James
1 + 1 = 2
```

The ‘ f ’ Interpolator

The literal ‘f’ interpolator allows to create a formatted String, similar to printf in C language. While using ‘f’ interpolator, all variable references should be followed by the **printf** style format specifiers such as %d, %i, %f, etc.

Let us take an example of append floating point value (height = 1.9d) and String variable (name = “James”) with normal string. The following code snippet of implementing ‘f’ Interpolator. Here $name%s to print (String variable) James and $height%2.2f to print (floating point value) 1.90.

```
val height = 1.9d
val name = "James"
println(f"$name%s is $height%2.2f meters tall") //James is 1.90 meters tall
```

It is type safe (i.e.) the variable reference and following format specifier should match otherwise it is showing error. The ‘ f ’ interpolator makes use of the String format utilities (format specifiers) available in Java. By default means, there is no % character after variable reference. It will assume as %s (String).

‘raw’ Interpolator

The ‘raw’ interpolator is similar to ‘s’ interpolator except that it performs no escaping of literals within a string. The following code snippets in a table will differ the usage of ‘s’ and ‘raw’ interpolators. In outputs

of ‘s’ usage ‘\n’ effects as new line and in output of ‘raw’ usage the ‘\n’ will not effect. It will print the complete string with escape letters.

‘s’ interpolator usage	‘raw’ interpolator usage

Program −
`object Demo {`
`  def main(args: Array[String]) {`
`  println(s"Result = \n a \n b")`
`  }`
`}`
	Program −
	`object Demo {`
	`  def main(args: Array[String]) {`
	`  println(raw"Result = \n a \n b")`
	`  }`
	`}`

Output −
`Result =`
`a`
`b`
	Output −
	`Result = \n a \n b`

String Methods

Following is the list of methods defined by **java.lang.String** class and can be used directly in your Scala programs −

Sr.No	Methods with Description
1	**char charAt(int index)** Returns the character at the specified index.
2	**int compareTo(Object o)** Compares this String to another Object.
3	**int compareTo(String anotherString)** Compares two strings lexicographically.
4	**int compareToIgnoreCase(String str)** Compares two strings lexicographically, ignoring case differences.
5	**String concat(String str)** Concatenates the specified string to the end of this string.
6	**boolean contentEquals(StringBuffer sb)** Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
7	**static String copyValueOf(char[] data)** Returns a String that represents the character sequence in the array specified.
8	**static String copyValueOf(char[] data, int offset, int count)** Returns a String that represents the character sequence in the array specified.
9	**boolean endsWith(String suffix)** Tests if this string ends with the specified suffix.
10	**boolean equals(Object anObject)** Compares this string to the specified object.
11	**boolean equalsIgnoreCase(String anotherString)** Compares this String to another String, ignoring case considerations.

12	**byte getBytes()** Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
13	**byte[] getBytes(String charsetName)** Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
14	**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** Copies characters from this string into the destination character array.
15	**int hashCode()** Returns a hash code for this string.
16	**int indexOf(int ch)** Returns the index within this string of the first occurrence of the specified character.
17	**int indexOf(int ch, int fromIndex)** Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
18	**int indexOf(String str)** Returns the index within this string of the first occurrence of the specified substring.
19	**int indexOf(String str, int fromIndex)** Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
20	**String intern()** Returns a canonical representation for the string object.
21	**int lastIndexOf(int ch)** Returns the index within this string of the last occurrence of the specified character.
22	

	int lastIndexOf(int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
23	**int lastIndexOf(String str)** Returns the index within this string of the rightmost occurrence of the specified substring.
24	**int lastIndexOf(String str, int fromIndex)** Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
25	**int length()** Returns the length of this string.
26	**boolean matches(String regex)** Tells whether or not this string matches the given regular expression.
27	**boolean regionMatches(boolean ignoreCase, int toffset, String other, int offset, int len)** Tests if two string regions are equal.
28	**boolean regionMatches(int toffset, String other, int offset, int len)** Tests if two string regions are equal.
29	**String replace(char *oldChar*, char newChar)** Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
30	**String replaceAll(String regex, String replacement** Replaces each substring of this string that matches the given regular expression with the given replacement.
31	**String replaceFirst(String regex, String replacement)** Replaces the first substring of this string that matches the given regular expression with the given replacement.
32	**String[] split(String regex)**

	Splits this string around matches of the given regular expression.
33	**String[] split(String regex, int limit)** Splits this string around matches of the given regular expression.
34	**boolean startsWith(String prefix)** Tests if this string starts with the specified prefix.
35	**boolean startsWith(String prefix, int toffset)** Tests if this string starts with the specified prefix beginning a specified index.
36	**CharSequence subSequence(int beginIndex, int endIndex)** Returns a new character sequence that is a subsequence of this sequence.
37	**String substring(int beginIndex)** Returns a new string that is a substring of this string.
38	**String substring(int beginIndex, int endIndex)** Returns a new string that is a substring of this string.
39	**char[] toCharArray()** Converts this string to a new character array.
40	**String toLowerCase()** Converts all of the characters in this String to lower case using the rules of the default locale.
41	**String toLowerCase(Locale locale)** Converts all of the characters in this String to lower case using the rules of the given Locale.
42	**String toString()** This object (which is already a string!) is itself returned.
43	**String toUpperCase()** Converts all of the characters in this String to upper case using the rules of the default locale.

44	**String toUpperCase(Locale locale)** Converts all of the characters in this String to upper case using the rules of the given Locale.
45	**String trim()** Returns a copy of the string, with leading and trailing whitespace omitted.
46	**static String valueOf(primitive data type x)** Returns the string representation of the passed data type argument.

Scala - Arrays

Scala provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables. The index of the first element of an array is the number zero and the index of the last element is the total number of elements minus one.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array and you must specify the type of array the variable can reference.

The following is the syntax for declaring an array variable.

Syntax

```
var z:Array[String] = new Array[String](3)

or

var z = new Array[String](3)
```

Here, z is declared as an array of Strings that may hold up to three elements. Values can be assigned to individual elements or get access to individual elements, it can be done by using commands like the following −

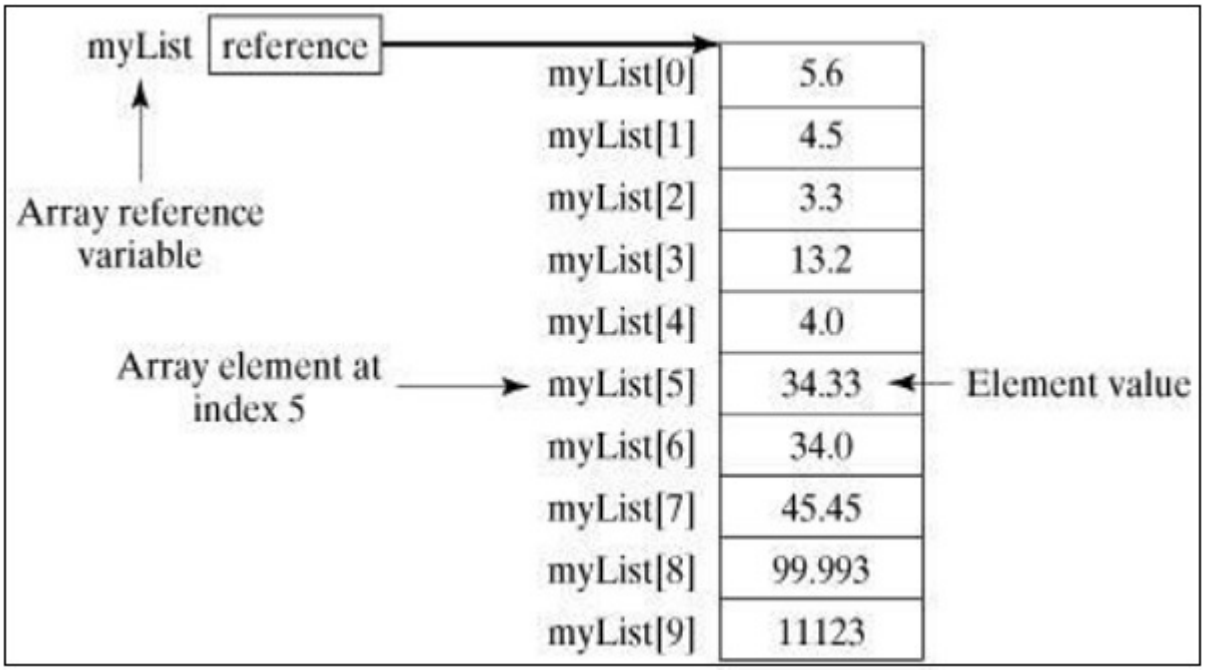
Command

```
z(0) = "Zara"; z(1) = "Nuha"; z(4/2) = "Ayan"
```

Here, the last example shows that in general the index can be any expression that yields a whole number. There is one more way of defining an array −

```
var z = Array("Zara", "Nuha", "Ayan")
```

Following picture represents an array **myList**. Here, **myList** holds ten double values and the indices are from 0 to 9.



Processing Arrays

When processing array elements, we often use loop contol structures because all of the elements in an array are of the same type and the size of the array is known.

Below is an example program of showing how to create, initialize and process arrays −

Example

```
object Demo {
   def main(args: Array[String]) {
      var myList = Array(1.9, 2.9, 3.4, 3.5)

      // Print all the array elements
      for ( x <- myList ) {
         println( x )
      }
```

```
      // Summing all elements
      var total = 0.0;

      for ( i <- 0 to (myList.length - 1)) {
         total += myList(i);
      }
      println("Total is " + total);

      // Finding the largest element
      var max = myList(0);

      for ( i <- 1 to (myList.length - 1) ) {
         if (myList(i) > max) max = myList(i);
      }

      println("Max is " + max);
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

Scala does not directly support various array operations and provides various methods to process arrays in any dimension. If you want to use the different methods then it is required to import **Array._** package.

Multi-Dimensional Arrays

There are many situations where you would need to define and use multi-dimensional arrays (i.e., arrays whose elements are arrays). For example, matrices and tables are examples of structures that can be realized as two-dimensional arrays.

The following is the example of defining a two-dimensional array −

```
var myMatrix = ofDim[Int](3,3)
```

This is an array that has three elements each being an array of integers that has three elements.

Try the following example program to process a multi-dimensional array −

Example

```
import Array._

object Demo {
   def main(args: Array[String]) {
      var myMatrix = ofDim[Int](3,3)

      // build a matrix
      for (i <- 0 to 2) {
         for ( j <- 0 to 2) {
            myMatrix(i)(j) = j;
         }
      }

      // Print two dimensional array
      for (i <- 0 to 2) {
         for ( j <- 0 to 2) {
            print(" " + myMatrix(i)(j));
         }
         println();
      }
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
0 1 2
0 1 2
0 1 2
```

Concatenate Arrays

Try the following example which makes use of concat() method to concatenate two arrays. You can pass more than one array as arguments to concat() method.

Example

```
import Array._

object Demo {
   def main(args: Array[String]) {
      var myList1 = Array(1.9, 2.9, 3.4, 3.5)
      var myList2 = Array(8.9, 7.9, 0.4, 1.5)

      var myList3 =  concat( myList1, myList2)

      // Print all the array elements
      for ( x <- myList3 ) {
         println( x )
      }
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
1.9
2.9
3.4
3.5
8.9
7.9
0.4
1.5
```

Create Array with Range

Use of range() method to generate an array containing a sequence of increasing integers in a given range. You can use final argument as step to create the sequence; if you do not use final argument, then step would be assumed as 1.

Let us take an example of creating an array of range (10, 20, 2): It means creating an array with elements between 10 and 20 and range difference 2. Elements in the array are 10, 12, 14, 16, and 18.

Another example: range (10, 20). Here range difference is not given so by default it assumes 1 element. It create an array with the elements in between 10 and 20 with range difference 1. Elements in the array are 10, 11, 12, 13, …, and 19.

The following example program shows how to create an array with ranges.

Example

```
import Array._

object Demo {
   def main(args: Array[String]) {
      var myList1 = range(10, 20, 2)
      var myList2 = range(10,20)

      // Print all the array elements
      for ( x <- myList1 ) {
         print( " " + x )
      }

      println()
      for ( x <- myList2 ) {
         print( " " + x )
      }
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
10 12 14 16 18
10 11 12 13 14 15 16 17 18 19
```

Scala Array Methods

Following are the important methods, which you can use while playing with array. As shown above, you would have to import **Array.**_ package before using any of the mentioned methods. For a complete list of

methods available, please check official documentation of Scala.

Sr.No	Methods with Description
1	**def apply(x: T, xs: T*): Array[T]** Creates an array of T objects, where T can be Unit, Double, Float, Long, Int, Char, Short, Byte, Boolean.
2	**def concat[T](xss: Array[T]*): Array[T]** Concatenates all arrays into a single array.
3	**def copy(src: AnyRef, srcPos: Int, dest: AnyRef, destPos: Int, length: Int): Unit** Copy one array to another. Equivalent to Java's System.arraycopy(src, srcPos, dest, destPos, length).
4	**def empty[T]: Array[T]** Returns an array of length 0
5	**def iterate[T](start: T, len: Int)(f: (T) => T): Array[T]** Returns an array containing repeated applications of a function to a start value.
6	**def fill[T](n: Int)(elem: => T): Array[T]** Returns an array that contains the results of some element computation a number of times.
7	**def fill[T](n1: Int, n2: Int)(elem: => T): Array[Array[T]]** Returns a two-dimensional array that contains the results of some element computation a number of times.
8	**def iterate[T](start: T, len: Int)(f: (T) => T): Array[T]** Returns an array containing repeated applications of a function to a start value.
9	**def ofDim[T](n1: Int): Array[T]** Creates array with given dimensions.
10	**def ofDim[T](n1: Int, n2: Int): Array[Array[T]]** Creates a 2-dimensional array
11	

	def ofDim[T](n1: Int, n2: Int, n3: Int): Array[Array[Array[T]]] Creates a 3-dimensional array
12	**def range(start: Int, end: Int, step: Int): Array[Int]** Returns an array containing equally spaced values in some integer interval.
13	**def range(start: Int, end: Int): Array[Int]** Returns an array containing a sequence of increasing integers in a range.
14	**def tabulate[T](n: Int)(f: (Int)=> T): Array[T]** Returns an array containing values of a given function over a range of integer values starting from 0.
15	**def tabulate[T](n1: Int, n2: Int)(f: (Int, Int) => T): Array[Array[T]]** Returns a two-dimensional array containing values of a given function over ranges of integer values starting from 0.

Scala - Collections

Scala has a rich set of collection library. Collections are containers of things. Those containers can be sequenced, linear sets of items like List, Tuple, Option, Map, etc. The collections may have an arbitrary number of elements or be bounded to zero or one element (e.g., Option).

Collections may be **strict** or **lazy**. Lazy collections have elements that may not consume memory until they are accessed, like **Ranges**. Additionally, collections may be **mutable** (the contents of the reference can change) or **immutable** (the thing that a reference refers to is never changed). Note that immutable collections may contain mutable items.

For some problems, mutable collections work better, and for others, immutable collections work better. When in doubt, it is better to start with an immutable collection and change it later if you need mutable ones.

This chapter throws light on the most commonly used collection types and most frequently used operations over those collections.

Sr.No	Collections with Description
1	Scala Lists Scala's List[T] is a linked list of type T.
2	Scala Sets A set is a collection of pairwise different elements of the same type.
3	Scala Maps A Map is a collection of key/value pairs. Any value can be retrieved based on its key.
4	Scala Tuples Unlike an array or list, a tuple can hold objects with different types.
5	Scala Options Option[T] provides a container for zero or one element of a given type.
6	Scala Iterators An iterator is not a collection, but rather a way to access the elements of a collection one by one.

Scala - Traits

A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of traits.

Traits are used to define object types by specifying the signature of the supported methods. Scala also allows traits to be partially implemented but traits may not have constructor parameters.

A trait definition looks just like a class definition except that it uses the keyword **trait**. The following is the basic example syntax of trait.

Syntax

```
trait Equal {
   def isEqual(x: Any): Boolean
```

```
   def isNotEqual(x: Any): Boolean = !isEqual(x)
}
```

This trait consists of two methods **isEqual** and **isNotEqual**. Here, we have not given any implementation for isEqual where as another method has its implementation. Child classes extending a trait can give implementation for the un-implemented methods. So a trait is very similar to what we have **abstract classes** in Java.

Let us assume an example of trait **Equal** contain two methods **isEqual()** and **isNotEqual()**. The trait **Equal** contain one implemented method that is **isEqual()** so when user defined class **Point** extends the trait **Equal**, implementation to **isEqual()** method in **Point** class should be provided.

Here it is required to know two important method of Scala, which are used in the following example.

- **obj.isInstanceOf [Point]** To check Type of obj and Point are same are not.
- **obj.asInstanceOf [Point]** means exact casting by taking the object obj type and returns the same obj as Point type.

Try the following example program to implement traits.

Example

```
trait Equal {
   def isEqual(x: Any): Boolean
   def isNotEqual(x: Any): Boolean = !isEqual(x)
}

class Point(xc: Int, yc: Int) extends Equal {
   var x: Int = xc
   var y: Int = yc

   def isEqual(obj: Any) = obj.isInstanceOf[Point] && obj.asInstanceOf[Point].x == y
}

object Demo {
   def main(args: Array[String]) {
      val p1 = new Point(2, 3)
      val p2 = new Point(2, 4)
      val p3 = new Point(3, 3)

      println(p1.isNotEqual(p2))
      println(p1.isNotEqual(p3))
      println(p1.isNotEqual(2))
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
true
false
true
```

Value classes and Universal Traits

Value classes are new mechanism in Scala to avoid allocating runtime objects. It contains a primary constructor with exactly one **val** parameter. It contains only methods (def) not allowed var, val, nested classes, traits, or objects. Value class cannot be extended by another class. This can be possible by extending your value class with AnyVal. The typesafety of custom datatypes without the runtime overhead.

Let us take an examples of value classes Weight, Height, Email, Age, etc. For all these examples it is not required to allocate memory in the application.

A value class not allowed to extend traits. To permit value classes to extend traits, **universal traits** are introduced which extends for **Any**.

Example

```
trait Printable extends Any {
   def print(): Unit = println(this)
}
class Wrapper(val underlying: Int) extends AnyVal with Printable

object Demo {
   def main(args: Array[String]) {
      val w = new Wrapper(3)
      w.print() // actually requires instantiating a Wrapper instance
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

It will give you the hash code of Wrapper class.

```
Wrapper@13
```

When to Use Traits?

There is no firm rule, but here are few guidelines to consider −

- If the behavior will not be reused, then make it a concrete class. It is not reusable behavior after all.
- If it might be reused in multiple, unrelated classes, make it a trait. Only traits can be mixed into different parts of the class hierarchy.
- If you want to **inherit** from it in Java code, use an abstract class.
- If you plan to distribute it in compiled form, and you expect outside groups to write classes inheriting from it, you might lean towards using an abstract class.
- If efficiency is very important, lean towards using a class.

Scala - Pattern Matching

Pattern matching is the second most widely used feature of Scala, after function values and closures. Scala provides great support for pattern matching, in processing the messages.

A pattern match includes a sequence of alternatives, each starting with the keyword **case**. Each alternative includes a **pattern** and one or more **expressions**, which will be evaluated if the pattern matches. An arrow symbol => separates the pattern from the expressions.

Try the following example program, which shows how to match against an integer value.

Example

```
object Demo {
   def main(args: Array[String]) {
      println(matchTest(3))
   }

   def matchTest(x: Int): String = x match {
      case 1 => "one"
      case 2 => "two"
      case _ => "many"
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
many
```

The block with the case statements defines a function, which maps integers to strings. The match keyword provides a convenient way of applying a function (like the pattern matching function above) to an object.

Try the following example program, which matches a value against patterns of different types.

Example

```
object Demo {
   def main(args: Array[String]) {
      println(matchTest("two"))
      println(matchTest("test"))
      println(matchTest(1))
   }

   def matchTest(x: Any): Any = x match {
      case 1 => "one"
      case "two" => 2
      case y: Int => "scala.Int"
      case _ => "many"
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
2
many
one
```

Matching using Case Classes

The **case classes** are special classes that are used in pattern matching with case expressions. Syntactically, these are standard classes with a special modifier: **case**.

Try the following, it is a simple pattern matching example using case class.

Example

```
object Demo {
   def main(args: Array[String]) {
      val alice = new Person("Alice", 25)
      val bob = new Person("Bob", 32)
      val charlie = new Person("Charlie", 32)

      for (person <- List(alice, bob, charlie)) {
         person match {
            case Person("Alice", 25) => println("Hi Alice!")
            case Person("Bob", 32) => println("Hi Bob!")
            case Person(name, age) => println(
               "Age: " + age + " year, name: " + name + "?")
         }
      }
   }
   case class Person(name: String, age: Int)
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Hi Alice!
Hi Bob!
Age: 32 year, name: Charlie?
```

Adding the case keyword causes the compiler to add a number of useful features automatically. The keyword suggests an association with case expressions in pattern matching.

First, the compiler automatically converts the constructor arguments into immutable fields (vals). The val keyword is optional. If you want mutable fields, use the var keyword. So, our constructor argument lists are now shorter.

Second, the compiler automatically implements **equals, hashCode,** and **toString** methods to the class, which use the fields specified as constructor arguments. So, we no longer need our own toString() methods.

Finally, also, the body of **Person** class becomes empty because there are no methods that we need to define!

Scala - Regular Expressions

This chapter explains how Scala supports regular expressions through **Regex** class available in the scala.util.matching package.

Try the following example program where we will try to find out word **Scala** from a statement.

Example

```
import scala.util.matching.Regex

object Demo {
   def main(args: Array[String]) {
      val pattern = "Scala".r
      val str = "Scala is Scalable and cool"

      println(pattern findFirstIn str)
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Some(Scala)
```

We create a String and call the **r( )** method on it. Scala implicitly converts the String to a RichString and invokes that method to get an instance of Regex. To find a first match of the regular expression, simply call the **findFirstIn()** method. If instead of finding only the first occurrence we would like to find all occurrences of the matching word, we can use the **findAllIn( )** method and in case there are multiple Scala words available in the target string, this will return a collection of all matching words.

You can make use of the mkString( ) method to concatenate the resulting list and you can use a pipe (|) to search small and capital case of Scala and you can use **Regex** constructor instead or **r()** method to create a pattern.

Try the following example program.

Example

```
import scala.util.matching.Regex

object Demo {
   def main(args: Array[String]) {
      val pattern = new Regex("(S|s)cala")
      val str = "Scala is scalable and cool"

      println((pattern findAllIn str).mkString(","))
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Scala,scala
```

If you would like to replace matching text, we can use **replaceFirstIn( )** to replace the first match or **replaceAllIn( )** to replace all occurrences.

Example

```
object Demo {
   def main(args: Array[String]) {
      val pattern = "(S|s)cala".r
      val str = "Scala is scalable and cool"

      println(pattern replaceFirstIn(str, "Java"))
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Java is scalable and cool
```

Forming Regular Expressions

Scala inherits its regular expression syntax from Java, which in turn inherits most of the features of Perl. Here are just some examples that should be enough as refreshers −

Following is the table listing down all the regular expression Meta character syntax available in Java.

Subexpression	Matches
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
\\A	Beginning of entire string
\\z	End of entire string
\\Z	End of entire string except allowable final line terminator.
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more of the previous thing
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a|b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.
(?: re)	Groups regular expressions without remembering matched text.
(?> re)	Matches independent pattern without backtracking.
\\w	Matches word characters.
\\W	Matches nonword characters.
\\s	Matches whitespace. Equivalent to [\t\n\r\f].
\\S	Matches nonwhitespace.
\\d	Matches digits. Equivalent to [0-9].
\\D	Matches nondigits.
\\A	Matches beginning of string.

\\Z	Matches end of string. If a newline exists, it matches just before newline.
\\z	Matches end of string.
\\G	Matches point where last match finished.
\\n	Back-reference to capture group number "n"
\\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\\B	Matches nonword boundaries.
\\n, \\t, etc.	Matches newlines, carriage returns, tabs, etc.
\\Q	Escape (quote) all characters up to \\E
\\E	Ends quoting begun with \\Q

Regular-Expression Examples

Example	Description
.	Match any character except newline
[Rr]uby	Match "Ruby" or "ruby"
rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of the above
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit
\\d	Match a digit: [0-9]
\\D	Match a nondigit: [^0-9]
\\s	Match a whitespace character: [\t\r\n\f]
\\S	Match nonwhitespace: [^ \t\r\n\f]
\\w	Match a single word character: [A-Za-z0-9_]
\\W	Match a nonword character: [^A-Za-z0-9_]
ruby?	Match "rub" or "ruby": the y is optional
ruby*	Match "rub" plus 0 or more ys
ruby+	Match "rub" plus 1 or more ys
\\d{3}	Match exactly 3 digits
\\d{3,}	Match 3 or more digits
\\d{3,5}	Match 3, 4, or 5 digits
\\D\\d+	No group: + repeats \\d
(\\D\\d)+/	Grouped: + repeats \\D\d pair
([Rr]uby(,)?)+	Match "Ruby", "Ruby, ruby, ruby", etc.

Note − that every backslash appears twice in the string above. This is because in Java and Scala a single backslash is an escape character in a string literal, not a regular character that shows up in the string. So instead of '\', you need to write '\\' to get a single backslash in the string.

Try the following example program.

Example

```
import scala.util.matching.Regex

object Demo {
   def main(args: Array[String]) {
      val pattern = new Regex("abl[ae]\\d+")
      val str = "ablaw is able1 and cool"

      println((pattern findAllIn str).mkString(","))
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
able1
```

Scala - Exception Handling

Scala's exceptions work like exceptions in many other languages like Java. Instead of returning a value in the normal way, a method can terminate by throwing an exception. However, Scala doesn't actually have checked exceptions.

When you want to handle exceptions, you use a try{...}catch{...} block like you would in Java except that the catch block uses matching to identify and handle the exceptions.

Throwing Exceptions

Throwing an exception looks the same as in Java. You create an exception object and then you throw it with the **throw** keyword as follows.

```
throw new IllegalArgumentException
```

Catching Exceptions

Scala allows you to **try/catch** any exception in a single block and then perform pattern matching against it using **case** blocks. Try the following example program to handle exception.

Example

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object Demo {
   def main(args: Array[String]) {
      try {
         val f = new FileReader("input.txt")
      } catch {
         case ex: FileNotFoundException =>{
            println("Missing file exception")
         }

         case ex: IOException => {
            println("IO Exception")
         }
      }
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Missing file exception
```

The behavior of this **try-catch** expression is the same as in other languages with exceptions. The body is executed, and if it throws an exception, each **catch** clause is tried in turn.

The finally Clause

You can wrap an expression with a **finally** clause if you want to cause some code to execute no matter how the expression terminates. Try the following program.

Example

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object Demo {
   def main(args: Array[String]) {
      try {
         val f = new FileReader("input.txt")
      } catch {
         case ex: FileNotFoundException => {
            println("Missing file exception")
         }

         case ex: IOException => {
            println("IO Exception")
         }
      } finally {
         println("Exiting finally...")
      }
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Missing file exception
Exiting finally...
```

Scala - Extractors

An extractor in Scala is an object that has a method called **unapply** as one of its members. The purpose of that unapply method is to match a value and take it apart. Often, the extractor object also defines a dual method **apply** for building values, but this is not required.

Example

Let us take an example of object defines both **apply** and **unapply** methods. The apply method has the same meaning as always: it turns Test into an object that can be applied to arguments in parentheses in the same way a method is applied. So you can write Test ("Zara", "gmail.com") to construct the string "Zara@gmail.com".

The **unapply** method is what turns Test class into an **extractor** and it reverses the construction process of **apply**. Where apply takes two strings and forms an email address string out of them, unapply takes an email address and returns potentially two strings: the **user** and the **domain** of the address.

The **unapply** must also handle the case where the given string is not an email address. That's why unapply returns an Option-type over pairs of strings. Its result is either **Some (user, domain)** if the string str is an email address with the given user and domain parts, or None, if str is not an email address. Here are some examples as follows.

Syntax

```
unapply("Zara@gmail.com") equals Some("Zara", "gmail.com")
unapply("Zara Ali") equals None
```

Following example program shows an extractor object for email addresses.

Example

```
object Demo {
   def main(args: Array[String]) {
      println ("Apply method : " + apply("Zara", "gmail.com"));
      println ("Unapply method : " + unapply("Zara@gmail.com"));
      println ("Unapply method : " + unapply("Zara Ali"));
   }

   // The injection method (optional)
   def apply(user: String, domain: String) = {
      user +"@"+ domain
   }

   // The extraction method (mandatory)
   def unapply(str: String): Option[(String, String)] = {
      val parts = str split "@"

      if (parts.length == 2){
         Some(parts(0), parts(1))
      } else {
         None
      }
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Apply method : Zara@gmail.com
Unapply method : Some((Zara,gmail.com))
Unapply method : None
```

Pattern Matching with Extractors

When an instance of a class is followed by parentheses with a list of zero or more parameters, the compiler invokes the **apply** method on that instance. We can define apply both in objects and in classes.

As mentioned above, the purpose of the **unapply** method is to extract a specific value we are looking for. It does the opposite operation **apply** does. When comparing an extractor object using the **match** statement the **unapply** method will be automatically executed.

Try the following example program.

Example

```
object Demo {
   def main(args: Array[String]) {
      val x = Demo(5)
      println(x)

      x match {
         case Demo(num) => println(x+" is bigger two times than "+num)

         //unapply is invoked
         case _ => println("i cannot calculate")
      }
   }
   def apply(x: Int) = x*2
   def unapply(z: Int): Option[Int] = if (z%2==0) Some(z/2) else None
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
10
10 is bigger two times than 5
```

Scala - Files I/O

Scala is open to make use of any Java objects and **java.io.File** is one of the objects which can be used in Scala programming to read and write files.

The following is an example program to writing to a file.

Example

```
import java.io._

object Demo {
   def main(args: Array[String]) {
      val writer = new PrintWriter(new File("test.txt" ))

      writer.write("Hello Scala")
      writer.close()
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

It will create a file named **Demo.txt** in the current directory, where the program is placed. The following is the content of that file.

Output

```
Hello Scala
```

Reading a Line from Command Line

Sometime you need to read user input from the screen and then proceed for some further processing. Following example program shows you how to read input from the command line.

Example

```
object Demo {
   def main(args: Array[String]) {
      print("Please enter your input : " )
      val line = Console.readLine

      println("Thanks, you just typed: " + line)
   }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Please enter your input : Scala is great
Thanks, you just typed: Scala is great
```

Reading File Content

Reading from files is really simple. You can use Scala's **Source** class and its companion object to read files. Following is the example which shows you how to read from **"Demo.txt"** file which we created earlier.

Example

```
import scala.io.Source

object Demo {
   def main(args: Array[String]) {
      println("Following is the content read:" )

      Source.fromFile("Demo.txt" ).foreach {
         print
      }
```

```
    }
}
```

Save the above program in **Demo.scala**. The following commands are used to compile and execute this program.

Command

```
\>scalac Demo.scala
\>scala Demo
```

Output

```
Following is the content read:
Hello Scala
```