Scala Tutorial

SCALA TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

ABOUT THE TUTORIAL

Scala Tutorial

Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. Scala has been created by Martin Odersky and he released the first version in 2003.

Scala smoothly integrates features of object-oriented and functional languages. This tutorial gives a great understanding on Scala.

Audience

This tutorial has been prepared for the beginners to help them understand programming Language Scala in simple and easy steps. After completing this tutorial, you will find yourself at a moderate level of expertise in using Scala from where you can take yourself to next levels.

Prerequisites

Scala Programming is based on Java, so if you are aware of Java syntax, then it's pretty easy to learn Scala. Further if you do not have expertise in Java but you know any other programming language like C, C++ or Python, then it will also help in grasping Scala concepts very quickly.

Copyright & Disclaimer Notice

Table of Content

CHAPTER

1

Scala Overview

Scala, short for Scalable Language, is a hybrid functional programming language. It was created by Martin Odersky and it was first released in 2003.

Scala smoothly integrates features of object-oriented and functional languages and 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 is the important list of features, which make Scala a first choice of the 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 because 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, 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 run-time 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's in Scala, and also your own, custom Java classes, or your favourite Java open source projects.

Scala vs Java:

Scala has a set of features, which 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 few of the most popular Scala web frameworks:

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

CHAPTER

2

Scala Environment Setup

The Scala language can be installed on any UNIX-like or Windows system. Before you start installing Scala on your machine, you must have Java 1.5 or greater installed on your computer.

Installing Scala on Windows:

STEP (1): JAVA SETUP:

First, you must set the JAVA_HOME environment variable and add the JDK's bin directory to your PATH variable. To verify if everything is fine, at command prompt, type **java -version** and press Enter. You should see something like the following:

```
C:\>java -version
java version "1.6.0_15"
Java(TM) SE Runtime Environment (build 1.6.0_15-b03)
Java HotSpot(TM) 64-Bit Server VM (build 14.1-b02, mixed mode)

C:\>
```

Next, test to see that the Java compiler is installed. Type **javac -version**. You should see something like the following:

```
C:\>javac -version
javac 1.6.0_15

C:\>
```

STEP (2): SCALA SETUP:

Next, you can download Scala from http://www.scala-lang.org/downloads. At the time of writing this tutorial, I downloaded *scala-2.9.0.1-installer.jar* and put it in C:/> directory. Make sure you have admin privilege to proceed. Now, execute the following command at command prompt:

```
C:\>java -jar scala-2.9.0.1-installer.jar

C:\>
```

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

```
C:\>scala -version
Scala code runner version 2.9.0.1 -- Copyright 2002-2011, LAMP/EPFL

C:\>
```

Congratulations, you have installed Scala on your Windows machine. Next section will teach you how to install scala on your Mac OS X and Unix/Linux machines.

Installing Scala on Mac OS X and Linux

STEP (1): JAVA SETUP:

Make sure you have got the Java JDK 1.5 or greater installed on your computer and set JAVA_HOME environment variable and add the JDK's bin directory to your PATH variable. To verify if everything is fine, at command prompt, type **java -version** and press Enter. You should see something like the following:

```
$java -version
java version "1.5.0_22"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_22-b03)
Java HotSpot(TM) Server VM (build 1.5.0_22-b03, mixed mode)
$
```

Next, test to see that the Java compiler is installed. Type **javac -version**. You should see something like the following:

```
$javac -version
javac 1.5.0_22
javac: no source files
Usage: javac <options> <source files>
.....................................................

$
```

STEP (2): SCALA SETUP:

Next, you can download Scala from http://www.scala-lang.org/downloads. At the time of writing this tutorial, I downloaded *scala-2.9.0.1-installer.jar* and put it in /tmp directory. Make sure you have admin privilege to proceed. Now, execute the following command at command prompt:

```
$java -jar scala-2.9.0.1-installer.jar
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:

```
$scala -version
Scala code runner version 2.9.0.1 -- Copyright 2002-2011, LAMP/EPFL

$
```

Congratulations, you have installed Scala on your UNIX/Linux machine.

CHAPTER

3

Scala Basic Syntax

If you have 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 others methods. Let us now briefly look into what do class, object, methods and instant variables mean.

- **Object -** Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, eating. An object is an instance of a class.
- **Class -** A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.
- **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 instant variables, which are called fields. An object's state is created by the values assigned to these fields.

First Scala Program:

INTERACTIVE MODE PROGRAMMING:

Invoking the interpreter without passing a script file as a parameter brings up the following prompt:

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

scala>
```

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

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

This will produce the following result:

```
Hello, Scala!
```

SCRIPT MODE PROGRAMMING:

Let us look at a simple code that would print the words *Hello, World!*

```
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
   }
}
```

Let's look at how to save the file, compile and run the program. Please follow the steps given below:

1. Open notepad and add the code as above.

2. Save the file as: HelloWorld.scala.

3. Open a command prompt window and go to the directory where you saved the program file. Assume it is C:\>

4. Type 'scalac HelloWorld.scala' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line.

5. Above command 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).

6. Now, type 'scala HelloWorld' to run your program.

7. You will be able to see 'Hello, World!' printed on the window.

```
C:\> scalac HelloWorld.scala
C:\> scala HelloWorld
Hello, World!
```

Basic Syntax:

About Scala programs, it is very important to keep in mind the following points.

- **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 several 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. There are following four types of identifiers supported by Scala:

ALPHANUMERIC IDENTIFIERS

An alphanumeric identifier starts with a letter or 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.

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:

```
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}
```

CHAPTER

4

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:

Data Type	Description
Byte	8 bit signed value. Range from -128 to 127
Short	16 bit signed value. Range -32768 to 32767
Int	32 bit signed value. Range -2147483648 to 2147483647
Long	64 bit signed value. -9223372036854775808 to 9223372036854775807
Float	32 bit IEEE 754 single-precision float
Double	64 bit IEEE 754 double-precision float
Char	16 bit unsigned Unicode character. Range from U+0000 to U+FFFF
String	A sequence of Chars
Boolean	Either the literal true or the literal false
Unit	Corresponds to no value
Null	null or empty reference
Nothing	The subtype of every other type; includes no values
Any	The supertype of any type; any object is of type *Any*
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.

INTEGER 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 LITERALS

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 consuctive 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."""
```

THE NULL VALUE

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:

```
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:

```
Hello   World
```

CHAPTER

5

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 the different syntax for the declaration of variables and they can be defined as value, i.e., constant or a variable. Following is the syntax to define a variable using **var** keyword:

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

Here, myVar is declared using the keyword var. This means that it is a variable that can change value and this is called mutable variable. Following is the syntax to define a variable using **val** keyword:

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

Here, myVal is declared using the keyword val. This means that it is a variable that can not be changed and this is called immutable variable.

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:

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

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

```
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:

```
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, the Tuple can be assigned to a val variable. [Note: We will study Tuple in subsequent chapters.]

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

And the type inference gets it right:

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

Variable Types:

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 or 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 mutable and 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 or immutable types and can be defined using either var or val.

CHAPTER

6

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:

```
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:

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

Public members:

Every member not labeled private or protected is public. There is no explicit modifier for public members. Such members can be accessed from anywhere. Following is the 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:

```
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).

CHAPTER

7

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:

There are the following arithmetic operators supported by Scala language:

Assume variable A holds 10 and variable B holds 20, then:

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 and remainder of after an integer division	B % A will give 0

Example:

Try the following example to understand all the arithmetic operators available in Scala Programming Language. Copy and paste the following Scala program in Test.scala file and compile and run this program.

```
object Test {
   def main(args: Array[String]) {
```

```
      var a = 10;
      var b = 20;
      var c = 25;
      var d = 25;
      println("a + b = " + (a + b) );
      println("a - b = " + (a - b) );
      println("a * b = " + (a * b) );
      println("b / a = " + (b / a) );
      println("b % a = " + (b % a) );
      println("c % a = " + (c % a) );

   }
}
```

This would produce the following result:

```
C:/>scalac Test.scala
C:/>scala Test
a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5

C:/>
```

Relational Operators:

There are the following relational operators supported by Scala language

Assume variable A holds 10 and variable B holds 20, then:

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.

Example:

The following simple example program demonstrates the relational operators. Copy and paste the following Scala program in Test.scala file and compile and run this program.

```
object Test {
   def main(args: Array[String]) {
      var a = 10;
      var b = 20;
      println("a == b = " + (a == b) );
      println("a != b = " + (a != b) );
      println("a > b = " + (a > b) );
      println("a < b = " + (a < b) );
      println("b >= a = " + (b >= a) );
      println("b <= a = " + (b <= a) );
   }
}
```

This would produce the following result:

```
C:/>scalac Test.scala
C:/>scala Test
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false

C:/>
```

Logical Operators:

There are the following logical operators supported by Scala language

Assume variable A holds 1 and variable B holds 0, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	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.

Example:

The following simple example program demonstrates the logical operators. Copy and paste the following Java program in Test.scala file and compile and run this program:

```
object Test {
   def main(args: Array[String]) {
      var a = true;
      var b = false;
```

```
      println("a && b = " + (a&&b) );

      println("a || b = " + (a||b) );

      println("!(a && b) = " + !(a && b) );
   }
}
```

This would produce the following result:

```
C:/>scalac Test.scala
C:/>scala Test
a && b = false
a || b = true
!(a && b) = true

C:/>
```

Bitwise Operators:

Bitwise operator works on bits and performs 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:

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 eather 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 efect of 'flipping' bits.	(~A) will give -60, which is 1100 0011
<<	Binary Left Shift Operator. 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 left operands 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

Example

The following simple example program demonstrates the bitwise operators. Copy and paste the following Scala program in Test.scala file and compile and run this program.

```
object Test {
   def main(args: Array[String]) {
      var a = 60;           /* 60 = 0011 1100 */
      var b = 13;           /* 13 = 0000 1101 */
      var c = 0;

      c = a & b;            /* 12 = 0000 1100 */
      println("a & b = " + c );

      c = a | b;            /* 61 = 0011 1101 */
      println("a | b = " + c );

      c = a ^ b;            /* 49 = 0011 0001 */
      println("a ^ b = " + c );

      c = ~a;               /* -61 = 1100 0011 */
      println("~a = " + c );

      c = a << 2;           /* 240 = 1111 0000 */
      println("a << 2 = " + c );

      c = a >> 2;           /* 215 = 1111 */
      println("a >> 2  = " + c );

      c = a >>> 2;          /* 215 = 0000 1111 */
      println("a >>> 2 = " + c );
   }
}
```

This would produce the following result:

```
C:/>scalac Test.scala
C:/>scala Test
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 15
```

```
a >>> 15

C:/>
```

Assignment Operators:

There are following assignment operators supported by Scala language:

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

Example:

The following simple example program demonstrates the assignment operators. Copy and paste the following Scala program in Test.scala file and compile and run this program.

```
object Test {
   def main(args: Array[String]) {
      var a = 10;
      var b = 20;
      var c = 0;

      c = a + b;
      println("c = a + b  = " + c );

      c += a ;
      println("c += a  = " + c );

      c -= a ;
      println("c -= a = " + c );

      c *= a ;
      println("c *= a = " + c );
```

```
      a = 10;
      c = 15;
      c /= a ;
      println("c /= a  = " + c );

      a = 10;
      c = 15;
      c %= a ;
      println("c %= a  = " + c );

      c <<= 2 ;
      println("c <<= 2  = " + c );

      c >>= 2 ;
      println("c >>= 2  = " + c );

      c >>= 2 ;
      println("c >>= a  = " + c );

      c &= a ;
      println("c &= 2  = " + c );

      c ^= a ;
      println("c ^= a  = " + c );

      c |= a ;
      println("c |= a  = " + c );
   }
}
```

This would produce the following result:

```
C:/>scalac Test.scala
C:/>scala Test
c = a + b  = 30
c += a  = 40
c -= a  = 30
c *= a  = 300
c /= a  = 1
c %= a  = 5
c <<= 2  = 20
c >>= 2  = 5
c >>= 2  = 1
c &= a  = 0
c ^= a  = 10
c |= a  = 10

C:/>
```

Operator 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.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

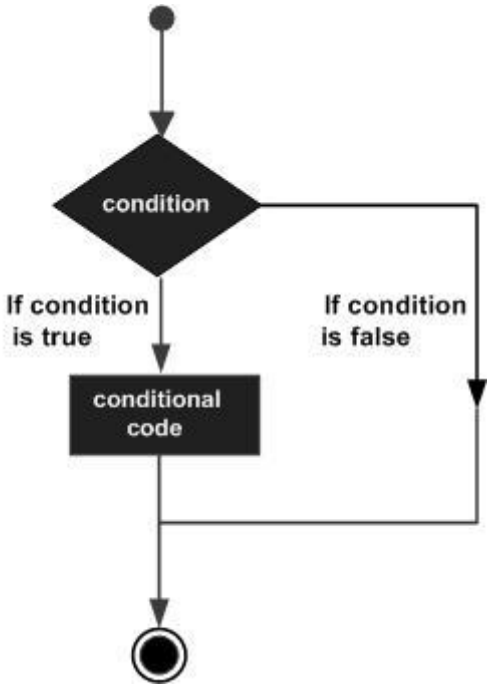
Category	Operator	Associativity
Postfix	() []	Left toright
Unary	! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Leftto right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise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

CHAPTER

8

Scala IF...ELSE Statements

Following is the general form of a typical decision making IF...ELSE structure found in most of the programming languages:



The if Statement:

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

SYNTAX:

The syntax of an if statement is:

```
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 statement will be executed. If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

EXAMPLE:

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

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

This would produce the following result:

```
C:/>scalac Test.scala
C:/>scala Test
This is if statement

C:/>
```

The 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
}
```

EXAMPLE:

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

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

This would produce the following result:

```
C:/>scalac Test.scala
C:/>scala Test
```

```
This is else statement

C:/>
```

The 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 syntax of an if...else if...else is:

```
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.
}
```

EXAMPLE:

```
object Test {
   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");
      }
   }
}
```

This would produce the following result:

```
C:/>scalac Test.scala
C:/>scala Test
Value of X is 30

C:/>
```

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
   }
}
```

You can nest *else if...else* in the similar way as we have nested *if* statement.

EXAMPLE:

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

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

This would produce the following result:

```
C:/>scalac Test.scala
C:/>scala Test
X = 30 and Y = 10

C:/>
```

CHAPTER

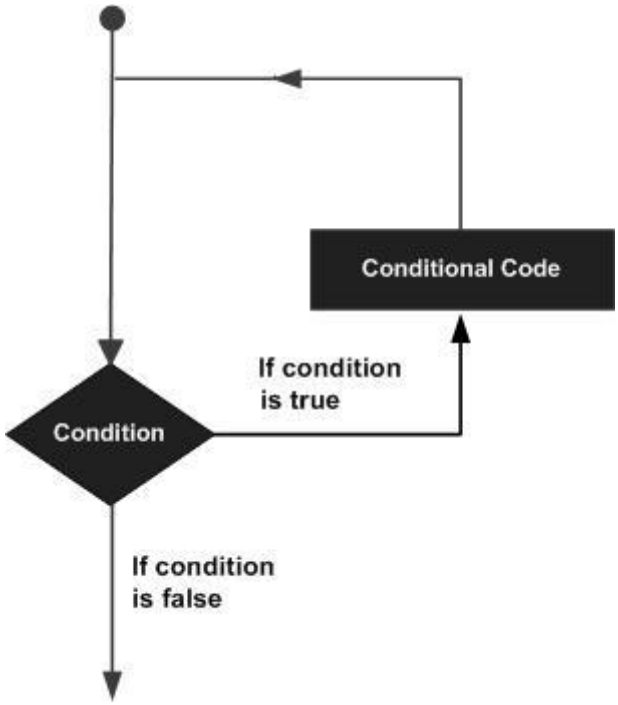
9

Scala Loop Types

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 from of a loop statement in most of the programming languages:



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

Loop Type	Description

while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

while loop

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

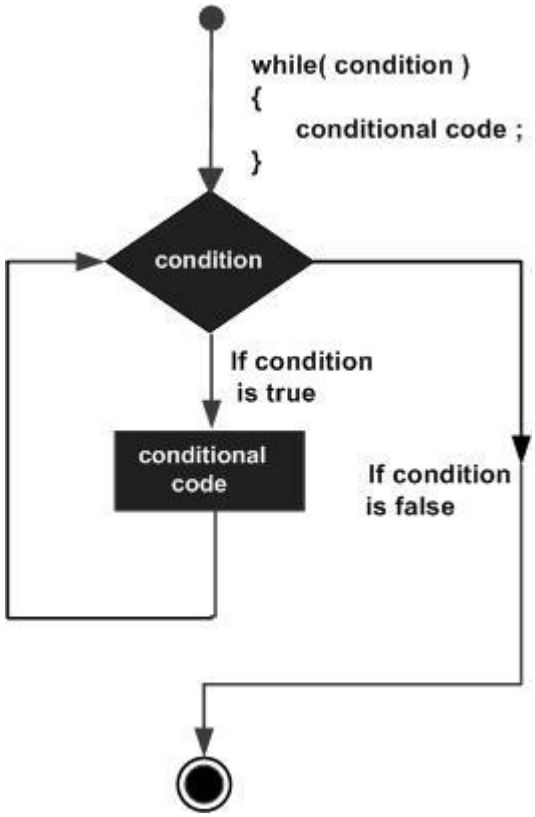
Syntax:

The syntax of a while loop in Scala is:

```
while(condition){
   statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram:



Here. key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
object Test {
   def main(args: Array[String]) {
      // Local variable declaration:
      var a = 10;

      // while loop execution
      while( a < 20 ){
         println( "Value of a: " + a );
         a = a + 1;
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

C:/>
```

do...while loop

Unlike **while** loop, which tests the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop. A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

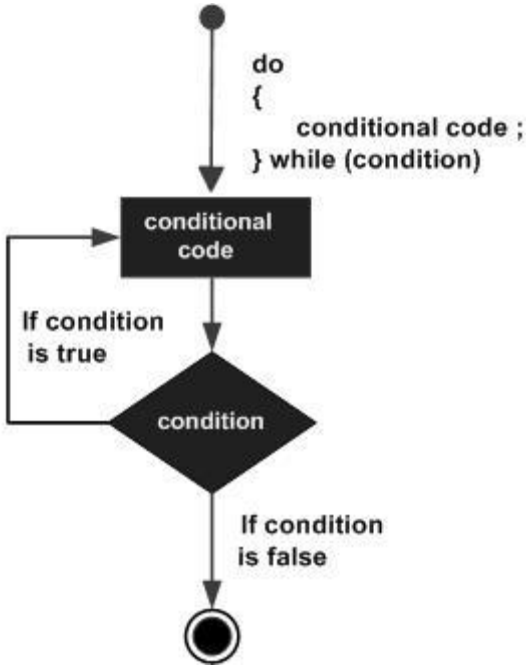
Syntax:

The syntax of a do...while loop in Scala is:

```
do{
   statement(s);
}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested. If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Flow Diagram:



Example:

```
object Test {
   def main(args: Array[String]) {
      // Local variable declaration:
      var a = 10;

      // do loop execution
      do{
         println( "Value of a: " + a );
         a = a + 1;
      }while( a < 20 )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

C:/>
```

for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. There are various forms of for loop in Scala which are described below:

The for Loop with Ranges

The simplest syntax of a for loop in Scala is:

```
for( var x <- Range ){
   statement(s);
}
```

Here, the **Range** could be a range of numbers and that is represented as **i to j** or sometime like **i until j**. The left-arrow <- operator is called a *generator*, so named because it's generating individual values from a range.

Example:

Following is the example of for loop with range using **i to j** syntax:

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      // for loop execution with a range
      for( a <- 1 to 10){
         println( "Value of a: " + a );
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9
value of a: 10

C:/>
```

Following is the example of for loop with range using **i until j** syntax:

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      // for loop execution with a range
      for( a <- 1 until 10){
         println( "Value of a: " + a );
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9

C:/>
```

You can use multiple ranges separated by semicolon (;) within a **for loop** and in that case loop will iterate through all the possible computations of the given ranges. Following is an example of using just two ranges, you can use more than two ranges as well.

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      var b = 0;
      // for loop execution with a range
      for( a <- 1 to 3; b <- 1 to 3){
         println( "Value of a: " + a );
         println( "Value of b: " + b );
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Value of a: 1
Value of b: 1
Value of a: 1
Value of b: 2
Value of a: 1
Value of b: 3
Value of a: 2
Value of b: 1
Value of a: 2
Value of b: 2
Value of a: 2
Value of b: 3
Value of a: 3
Value of b: 1
Value of a: 3
Value of b: 2
Value of a: 3
Value of b: 3

C:/>
```

The for Loop with Collections

The syntax of a for loop with collection is as follows:

```
for( var x <- List ){
   statement(s);
}
```

Here, the **List** variable is a collection type having a list of elements and *for loop* iterate through all the elements returning one element in x variable at a time.

Example:

Following is the example of for loop with a collection of numbers. Here we created this collection using*List()*. We will study collections in a separate chapter.

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6);

      // for loop execution with a collection
      for( a <- numList ){
         println( "Value of a: " + a );
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6

C:/>
```

The for Loop with Filters

Scala's for loop allows to filter out some elements using one or more **if** statement(s). Following is the syntax of *for loop* along with filters.

```
for( var x <- List
      if condition1; if condition2...
   ){
   statement(s);
}
```

To add more than one filter to a for expression, separate the filters with semicolons(;).

Example:

Following is the example of for loop along with filters:

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6,7,8,9,10);
```

```
      // for loop execution with multiple filters
      for( a <- numList
           if a != 3; if a < 8 ){
         println( "Value of a: " + a );
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
value of a: 1
value of a: 2
value of a: 4
value of a: 5
value of a: 6
value of a: 7

C:/>
```

The for Loop with yield:

You can store return values from a for loop in a variable or can return through a function. To do so, you prefix the body of the for expression by the keyword **yield** as follows:

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

Note the curly braces have been used to keep the variables and conditions and *retVal* is a variable where all the values of x will be stored in the form of collection.

Example:

Following is the example to show the usage of for loop along with yield:

```
object Test {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6,7,8,9,10);

      // for loop execution with a yield
      var retVal = for{ a <- numList
                        if a != 3; if a < 8
                     }yield a

      // Now print returned values using another loop.
      for( a <- retVal){
         println( "Value of a: " + a );
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
value of a: 1
```

```
value of a: 2
value of a: 4
value of a: 5
value of a: 6
value of a: 7

C:/>
```

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.

Control Statement	Description
break statement	Terminates the **loop** statement and transfers execution to the statement immediately following the loop.

break statement

As such there is no built-in break statement available in Scala, but if you are running Scala version 2.8, then there is a way to use *break* statement. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

Syntax:

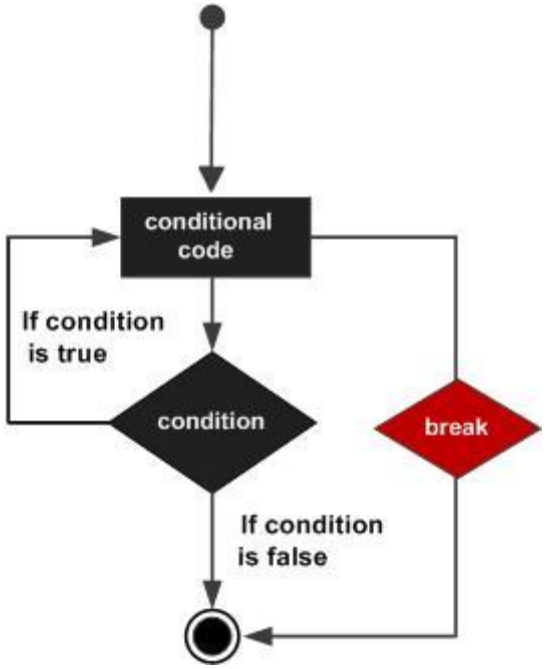
The syntax of a break statement is bit unusual but it works:

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

// create a Breaks object as follows
val loop = new Breaks;

// Keep the loop inside breakable as follows
loop.breakable{
    // Loop will go here
    for(...){
       ....
       // Break will go here
       loop.break;
   }
}
```

Flow Diagram:



Example:

```
import scala.util.control._

object Test {
   def main(args: Array[String]) {
      var a = 0;
      val numList = List(1,2,3,4,5,6,7,8,9,10);

      val loop = new Breaks;
      loop.breakable {
         for( a <- numList){
            println( "Value of a: " + a );
            if( a == 4 ){
               loop.break;
            }
         }
      }
      println( "After the loop" );
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Value of a: 1
Value of a: 2
Value of a: 3
Value of a: 4
After the loop

C:/>
```

Breaking Nested Loops:

Existing break has an issue while using for nested loops. So in case you have to use break for nested loops, then following is a way to proceed:

Example:

```
import scala.util.control._

object Test {
   def main(args: Array[String]) {
      var a = 0;
      var b = 0;
      val numList1 = List(1,2,3,4,5);
      val numList2 = List(11,12,13);

      val outer = new Breaks;
      val inner = new Breaks;

      outer.breakable {
         for( a <- numList1){
            println( "Value of a: " + a );
            inner.breakable {
               for( b <- numList2){
                  println( "Value of b: " + b );
                  if( b == 12 ){
                     inner.break;
                  }
               }
            } // inner breakable
         }
      } // outer breakable.
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Value of a: 1
Value of b: 11
Value of b: 12
Value of a: 2
Value of b: 11
Value of b: 12
Value of a: 3
Value of b: 11
Value of b: 12
Value of a: 4
Value of b: 11
Value of b: 12
Value of a: 5
Value of b: 11
Value of b: 12

C:/>
```

Infinite Loop:

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

```
object Test {
   def main(args: Array[String]) {
      var a = 10;
      // An infinite loop.
      while( true ){
         println( "Value of a: " + a );
      }
   }
}
```

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

CHAPTER

10

Scala Functions

A function is a group of statements that together 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 leave off the equals sign and method body. The enclosing type is then itself abstract.

Function Definitions:

A scala function definition has the following form:

```
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:

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

```
      return sum
   }
}
```

A function, which does not return anything, can return **Unit** which 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. Following 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 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 )
```

Following is the final example to define and then call the same function:

```
object Test {
   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
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Returned Value : 12

C:/>
```

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

Functions Call-by-Name

Typically, parameters to functions are by-value parameters; that is, the value of the parameter is determined before it is passed to the function. But what if we need to write a function that accepts as a parameter an expression that we don't want evaluated until it's called within our function? For this circumstance, Scala offers **call-by-name** parameters.

A call-by-name mechanism passes a code block to the callee and each time the callee accesses the parameter, the code block is executed and the value is calculated.

```
object Test {
   def main(args: Array[String]) {
        delayed(time());
   }

   def time() = {
      println("Getting time in nano seconds")
      System.nanoTime
   }
   def delayed( t: => Long ) = {
      println("In delayed method")
      println("Param: " + t)
      t
   }
}
```

Here, we declared the **delayed** method, which takes a call-by-name parameter by putting the => symbol between the variable name and the type. When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
In delayed method
Getting time in nano seconds
Param: 81303808765843
Getting time in nano seconds

C:/>
```

Here, delayed prints a message demonstrating that the method has been entered. Next, delayed prints a message with its value. Finally, delayed returns t.

Functions with Named Arguments

In a normal function call, the arguments in the call are matched one by one in the order of the parameters of the called function. Named arguments allow you to pass arguments to a function in a different order. The syntax is simply that each argument is preceded by a parameter name and an equals sign. Following is a simple example to show the concept:

```
object Test {
   def main(args: Array[String]) {
        printInt(b=5, a=7);
   }
   def printInt( a:Int, b:Int ) = {
      println("Value of a : " + a );
      println("Value of b : " + b );
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Value of a :  7
Value of b :  5

C:/>
```

Function with Variable Arguments

Scala allows you to indicate that the last parameter to a function may be repeated. This allows clients to pass variable length argument lists to the function. Following is a simple example to show the concept.

```
object Test {
   def main(args: Array[String]) {
        printStrings("Hello", "Scala", "Python");
   }
   def printStrings( args:String* ) = {
      var i : Int = 0;
      for( arg <- args ){
         println("Arg value[" + i + "] = " + arg );
         i = i + 1;
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Arg value[0] = Hello
Arg value[1] = Scala
Arg value[2] = Python

C:/>
```

Here, the type of args inside the printStrings function, which is declared as type "String*" is actually Array[String].

Recursive Functions

Recursion plays a big role in pure functional programming and Scala supports recursion functions very well. Recursion means a function can call itself repeatedly. Following is a good example of recursion where we calculate factorials of the passed number:

```
object Test {
   def main(args: Array[String]) {
      for (i <- 1 to 10)
         println( "Factorial of " + i + ": = " + factorial(i) )
   }

   def factorial(n: BigInt): BigInt = {
      if (n <= 1)
         1
      else
      n * factorial(n - 1)
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Factorial of 1: = 1
Factorial of 2: = 2
Factorial of 3: = 6
Factorial of 4: = 24
Factorial of 5: = 120
Factorial of 6: = 720
Factorial of 7: = 5040
Factorial of 8: = 40320
Factorial of 9: = 362880
Factorial of 10: = 3628800

C:/>
```

Default Parameter Values

Scala lets you specify default values for function parameters. The argument for such a parameter can optionally be omitted from a function call, in which case the corresponding argument will be filled in with the default. Following is an example of specifiying default parameters:

```
object Test {
   def main(args: Array[String]) {
        println( "Returned Value : " + addInt() );
   }
   def addInt( a:Int=5, b:Int=7 ) : Int = {
      var sum:Int = 0
      sum = a + b

      return sum
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Returned Value : 12

C:/>
```

If you specify one of the parameters, then first argument will be passed using that parameter and second will be taken from default value.

Higher-Order Functions

Scala allows the definition of **higher-order functions**. These are functions that take other functions as parameters, or whose result is a function. For example in the following code, apply() function takes another function **f** and a value **v** and applies function f to v:

```
object Test {
   def main(args: Array[String]) {

      println( apply( layout, 10) )

   }

   def apply(f: Int => String, v: Int) = f(v)

   def layout[A](x: A) = "[" + x.toString() + "]"
```

```
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
[10]

C:/>
```

Nested Functions

Scala allows you to define functions inside a function and functions defined inside other functions are called **local functions**. Here is an implementation of a factorial calculator, where we use a conventional technique of calling a second, nested method to do the work:

```
object Test {
   def main(args: Array[String]) {
      println( factorial(0) )
      println( factorial(1) )
      println( factorial(2) )
      println( factorial(3) )
   }

   def factorial(i: Int): Int = {
      def fact(i: Int, accumulator: Int): Int = {
         if (i <= 1)
            accumulator
         else
            fact(i - 1, i * accumulator)
      }
      fact(i, 1)
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
1
1
2
6

C:/>
```

Like a local variable declaration in many languages, a nested method is only visible inside the enclosing method. If you try to call **fact()** outside of **factorial()**, you will get a compiler error.

Anonymous Functions

Scala provides a relatively lightweight syntax for defining anonymous functions. Anonymous functions in source code are called **function literals** and at run time, function literals are instantiated into objects called **function values**.

Scala supports **first-class** functions, which means you can express functions in function literal syntax, i.e., (x: Int) => x + 1, and that functions can be represented by objects, which are called function values. The following expression creates a successor function for integers:

```
var inc = (x:Int) => x+1
```

Variable inc is now a function that can be used the usual way:

```
var x = inc(7)-1
```

It is also possible to define functions with multiple parameters as follows:

```
var mul = (x: Int, y: Int) => x*y
```

Variable mul is now a function that can be used the usual way:

```
println(mul(3, 4))
```

It is also possible to define functions with no parameter as follows:

```
var userDir = () => { System.getProperty("user.dir") }
```

Variable userDir is now a function that can be used the usual way:

```
println( userDir )
```

Partially Applied Functions

When you invoke a function, you're said to be applying the function to the arguments. If you pass all the expected arguments, you have fully applied it. If you send only a few arguments, then you get back a partially applied function. This gives you the convenience of binding some arguments and leaving the rest to be filled in later. Following is a simple example to show the concept:

```
import java.util.Date

object Test {
   def main(args: Array[String]) {
      val date = new Date
      log(date, "message1" )
      log(date, "message2" )
      log(date, "message3" )
   }

   def log(date: Date, message: String)  = {
     println(date + "----" + message)
   }
}
```

Here, the log( ) method takes two parameters: *date* and *message*. We want to invoke the method multiple times, with the same value for date but different values for message. We can eliminate the noise of passing the date to each call by partially applying that argument to the log( ) method. To do so, we first bind a value to the date parameter and leave the second parameter unbound by putting an underscore at its place. The result is a partially applied function that we've stored in a variable. We can now invoke this new method with only the unbound argument message as follows:

```
import java.util.Date

object Test {
   def main(args: Array[String]) {
      val logWithDateBound = log(new Date, _ : String)
```

```
            logWithDateBound("message1" )
      logWithDateBound("message2" )
      logWithDateBound("message3" )
   }

   def log(date: Date, message: String)  = {
     println(date + "----" + message)
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Thu Aug 18 01:41:07 GST 2011----message1
Thu Aug 18 01:41:08 GST 2011----message2
Thu Aug 18 01:41:08 GST 2011----message3

C:/>
```

Currying Functions

Currying transforms a function that takes multiple parameters into a chain of functions, each taking a single parameter. Curried functions are defined with multiple parameter lists, as follows:

```
def strcat(s1: String)(s2: String) = s1 + s2
```

Alternatively, you can also use the following syntax to define a curried function:

```
def strcat(s1: String) = (s2: String) => s1 + s2
```

Following is the syntax to call a curried function:

```
strcat("foo")("bar")
```

You can define more than two parameters on a curried function based on your requirement. Let us take a complete example to show currying concept:

```
object Test {
   def main(args: Array[String]) {
      val str1:String = "Hello, "
      val str2:String = "Scala!"
      println( "str1 + str2 = " +  strcat(str1)(str2) )
   }

   def strcat(s1: String)(s2: String) = {
      s1 + s2
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
str1 + str2 = Hello, Scala!

C:/>
```

CHAPTER

11

Scala Closures

A **closure** is a function, whose return value depends on the value of one or more variables declared outside this function. Consider the following piece of code with anonymous function:

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

Here, the only variable used in the function body, i * 0, is i, which is defined as a parameter to the function. Now, let us take another piece of 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. Let us try the following example:

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

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

```
C:/>scalac Test.scala
C:/>scala Test
muliplier(1) value = 3
muliplier(2) value = 6

C:/>
```

Above 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.

CHAPTER

12

Scala Strings

Consider the following simple example where we assign a string in a variable of type val:

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

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

Here, the type of the value above is **java.lang.String** borrowed from Java, because Scala strings are also Java strings. It is very good point to note that every Java class is available in Scala. As such, Scala does not have a String class and makes use of Java Strings. So this chapter has been written keeping Java String as a base.

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. Since strings are very useful objects, in the rest of this section, we present the most important methods class java.lang.String defines.

Creating Strings:

The most direct way to create a string is to write:

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

or

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

Whenever it encounters a string literal in your code, the compiler creates a String object with its value, in this case, "Hello world!', but if you like, you can give String keyword as I have shown you in alternate declaration.

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

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

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

```
C:/>scalac Test.scala
C:/>scala Test
Hello, world!

C:/>
```

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

String Length:

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

After the following two lines of code have been executed, len equals 17:

```
object Test {
   def main(args: Array[String]) {
      var palindrome = "Dot saw I was Tod";
      var len = palindrome.length();
      println( "String Length is : " + len );
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
String Length is : 17

C:/>
```

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!"
```

Let us look at the following example:

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

```
      var str2 =  "Tod";
      println("Dot " + str1 + str2);
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Dot Dot saw I was Tod

C:/>
```

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. Let us look at the following example, which makes use of printf() method:

```
object Test {
   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 the integer " +
                   "variable is %d, and the string " +
                   "is %s", floatVar, intVar, stringVar)
      println(fs)
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
The value of the float variable is 12.456000, while the
value of the integer var iable is 2000, and the
string is Hello, Scala!()

C:/>
```

String Methods:

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

SN	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 ooffset, int len)** Tests if two string regions are equal.
28	**boolean regionMatches(int toffset, String other, int ooffset, 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.

CHAPTER

13

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. Here is the syntax for declaring an array variable:

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

or

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

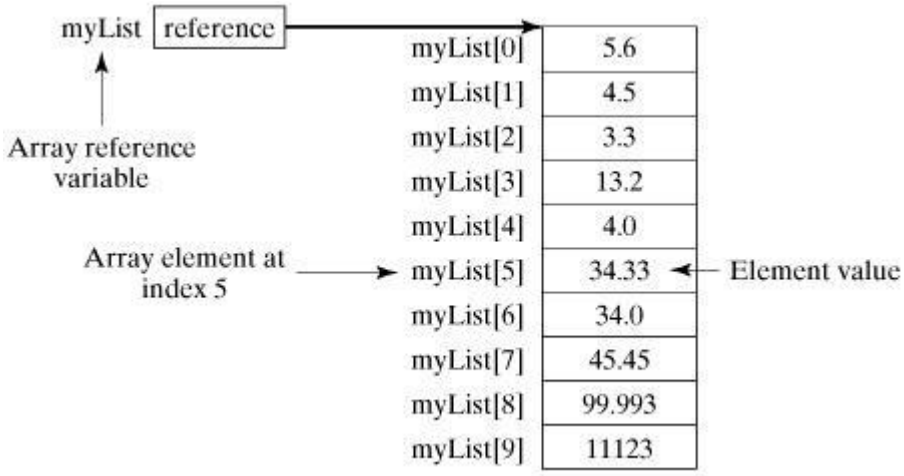
Here, z is declared as an array of Strings that may hold up to three elements. You can assign values to individual elements or get access to individual elements; it can be done by using commands like the following:

```
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 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 either for loop because all of the elements in an array are of the same type and the size of the array is known. Here is a complete example of showing how to create, initialize and process arrays:

```
object Test {
   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);

   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

```
C:/>
```

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.

Scala does not directly support multi-dimensional arrays and provides various methods to process arrays in any dimension. 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. The code that follows shows how one can process a multi-dimensional array:

```
import Array._

object Test {
   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();
      }

   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
0 1 2
0 1 2
0 1 2

C:/>
```

Concatenate Arrays:

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

```
import Array._

object Test {
```

```
   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 )
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
1.9
2.9
3.4
3.5
8.9
7.9
0.4
1.5

C:/>
```

Create Array with Range:

Following is the example, which makes 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.

```
import Array._

object Test {
   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 )
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
10 12 14 16 18
10 11 12 13 14 15 16 17 18 19
```

```
C:/>
```

Scala Arrays 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.

SN	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.

.

CHAPTER

14

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 gives details of the most commonly used collection types and most frequently used operations over those collections.

SN	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.

Collections are explained here individually:

Scala Lists

Scala Lists are quite similar to arrays which means, all the elements of a list have the same type, but there are two important differences. First, lists are immutable, which means elements of a list cannot be changed by assignment. Second, lists represent a linked list whereas arrays are flat.

The type of a list that has elements of type T is written as **List[T]**. For example, here are few lists defined for various data types:

```
// List of Strings
val fruit: List[String] = List("apples", "oranges", "pears")

// List of Integers
val nums: List[Int] = List(1, 2, 3, 4)

// Empty List.
val empty: List[Nothing] = List()

// Two dimensional list
val dim: List[List[Int]] =
   List(
      List(1, 0, 0),
      List(0, 1, 0),
      List(0, 0, 1)
   )
```

All lists can be defined using two fundamental building blocks, a tail **Nil** and **::,** which is pronounced **cons**. Nil also represents the empty list. All the above lists can be defined as follows:

```
// List of Strings
val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))

// List of Integers
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))

// Empty List.
val empty = Nil

// Two dimensional list
val dim = (1 :: (0 :: (0 :: Nil))) ::
          (0 :: (1 :: (0 :: Nil))) ::
          (0 :: (0 :: (1 :: Nil))) :: Nil
```

Basic Operations on List:

All operations on lists can be expressed in terms of the following three methods:

Methods	Description
Head	This method returns the first element of a list.
Tail	This method returns a list consisting of all elements except the first.
isEmpty	This method returns true if the list is empty otherwise false.

Following is the example showing usage of the above methods:

```
object Test {
   def main(args: Array[String]) {
```

```
      val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
      val nums = Nil

      println( "Head of fruit : " + fruit.head )
      println( "Tail of fruit : " + fruit.tail )
      println( "Check if fruit is empty : " + fruit.isEmpty )
      println( "Check if nums is empty : " + nums.isEmpty )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Head of fruit : apples
Tail of fruit : List(oranges, pears)
Check if fruit is empty : false
Check if nums is empty : true

C:/>
```

Concatenating Lists:

You can use either **:::** operator or **List.:::()** method or **List.concat()** method to add two or more lists. Following is the example:

```
object Test {
   def main(args: Array[String]) {
      val fruit1 = "apples" :: ("oranges" :: ("pears" :: Nil))
      val fruit2 = "mangoes" :: ("banana" :: Nil)

      // use two or more lists with ::: operator
      var fruit = fruit1 ::: fruit2
      println( "fruit1 ::: fruit2 : " + fruit )

      // use two lists with Set.:::() method
      fruit = fruit1.:::(fruit2)
      println( "fruit1.:::(fruit2) : " + fruit )

      // pass two or more lists as arguments
      fruit = List.concat(fruit1, fruit2)
      println( "List.concat(fruit1, fruit2) : " + fruit  )


   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
fruit1 ::: fruit2 : List(apples, oranges, pears, mangoes, banana)
fruit1.:::(fruit2) : List(mangoes, banana, apples, oranges, pears)
List.concat(fruit1, fruit2) : List(apples, oranges, pears, mangoes, banana)

C:/>
```

Creating Uniform Lists:

You can use **List.fill()** method creates a list consisting of zero or more copies of the same element as follows:

```
object Test {
   def main(args: Array[String]) {
      val fruit = List.fill(3)("apples") // Repeats apples three times.
      println( "fruit : " + fruit  )

      val num = List.fill(10)(2)         // Repeats 2, 10 times.
      println( "num : " + num  )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
fruit : List(apples, apples, apples)
num : List(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)

C:/>
```

Tabulating a Function:

You can use a function along with **List.tabulate()** method to apply on all the elements of the list before tabulating the list. Its arguments are just like those of List.fill: the first argument list gives the dimensions of the list to create, and the second describes the elements of the list. The only difference is that instead of the elements being fixed, they are computed from a function:

```
object Test {
   def main(args: Array[String]) {
      // Creates 5 elements using the given function.
      val squares = List.tabulate(6)(n => n * n)
      println( "squares : " + squares  )

      //
      val mul = List.tabulate( 4,5 )( _ * _ )
      println( "mul : " + mul  )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
squares : List(0, 1, 4, 9, 16, 25)
mul : List(List(0, 0, 0, 0, 0), List(0, 1, 2, 3, 4),
           List(0, 2, 4, 6, 8), List(0, 3, 6, 9, 12))

C:/>
```

Reverse List Order:

You can use **List.reverse** method to reverse all elements of the list. Following is the example to show the usage:

```
object Test {
   def main(args: Array[String]) {
```

```
      val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
      println( "Before reverse fruit : " + fruit )

      println( "After reverse fruit : " + fruit.reverse )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Before reverse fruit : List(apples, oranges, pears)
After reverse fruit : List(pears, oranges, apples)

C:/>
```

Scala List Methods:

Following are the important methods, which you can use while playing with Lists. For a complete list of methods available, please check official documentation of Scala.

SN	Methods with Description
1	**def +(elem: A): List[A]** Prepends an element to this list
2	**def ::(x: A): List[A]** Adds an element at the beginning of this list.
3	**def :::(prefix: List[A]): List[A]** Adds the elements of a given list in front of this list.
4	**def ::(x: A): List[A]** Adds an element x at the beginning of the list
5	**def addString(b: StringBuilder): StringBuilder** Appends all elements of the list to a string builder.
6	**def addString(b: StringBuilder, sep: String): StringBuilder** Appends all elements of the list to a string builder using a separator string.
7	**def apply(n: Int): A** Selects an element by its index in the list.
8	**def contains(elem: Any): Boolean** Tests whether the list contains a given value as an element.
9	**def copyToArray(xs: Array[A], start: Int, len: Int): Unit** Copies elements of the list to an array. Fills the given array xs with at most len elements of this list, starting at position start.
10	**def distinct: List[A]** Builds a new list from the list without any duplicate elements.
11	**def drop(n: Int): List[A]** Returns all elements except first n ones.
12	**def dropRight(n: Int): List[A]** Returns all elements except last n ones.

13	**def dropWhile(p: (A) => Boolean): List[A]** Drops longest prefix of elements that satisfy a predicate.
14	**def endsWith[B](that: Seq[B]): Boolean** Tests whether the list ends with the given sequence.
15	**def equals(that: Any): Boolean** The equals method for arbitrary sequences. Compares this sequence to some other object.
16	**def exists(p: (A) => Boolean): Boolean** Tests whether a predicate holds for some of the elements of the list.
17	**def filter(p: (A) => Boolean): List[A]** Returns all elements of the list which satisfy a predicate.
18	**def forall(p: (A) => Boolean): Boolean** Tests whether a predicate holds for all elements of the list.
19	**def foreach(f: (A) => Unit): Unit** Applies a function f to all elements of the list.
20	**def head: A** Selects the first element of the list.
21	**def indexOf(elem: A, from: Int): Int** Finds index of first occurrence of some value in the list after or at some start index.
22	**def init: List[A]** Returns all elements except the last.
23	**def intersect(that: Seq[A]): List[A]** Computes the multiset intersection between the list and another sequence.
24	**def isEmpty: Boolean** Tests whether the list is empty.
25	**def iterator: Iterator[A]** Creates a new iterator over all elements contained in the iterable object.
26	**def last: A** Returns the last element.
27	**def lastIndexOf(elem: A, end: Int): Int** Finds index of last occurrence of some value in the list before or at a given end index.
28	**def length: Int** Returns the length of the list.
29	**def map[B](f: (A) => B): List[B]** Builds a new collection by applying a function to all elements of this list.
30	**def max: A** Finds the largest element.
31	**def min: A** Finds the smallest element.
32	**def mkString: String** Displays all elements of the list in a string.
33	**def mkString(sep: String): String**

	Displays all elements of the list in a string using a separator string.
34	**def reverse: List[A]** Returns new list with elements in reversed order.
35	**def sorted[B >: A]: List[A]** Sorts the list according to an Ordering.
36	**def startsWith[B](that: Seq[B], offset: Int): Boolean** Tests whether the list contains the given sequence at a given index.
37	**def sum: A** Sums up the elements of this collection.
38	**def tail: List[A]** Returns all elements except the first.
39	**def take(n: Int): List[A]** Returns first n elements.
40	**def takeRight(n: Int): List[A]** Returns last n elements.
41	**def toArray: Array[A]** Converts the list to an array.
42	**def toBuffer[B >: A]: Buffer[B]** Converts the list to a mutable buffer.
43	**def toMap[T, U]: Map[T, U]** Converts this list to a map.
44	**def toSeq: Seq[A]** Converts the list to a sequence.
45	**def toSet[B >: A]: Set[B]** Converts the list to a set.
46	**def toString(): String** Converts the list to a string.

Scala Sets

Scala Set is a collection of pairwise different elements of the same type. In other words, a Set is a collection that contains no duplicate elements. There are two kinds of Sets, the **immutable** and the**mutable**. The difference between mutable and immutable objects is that when an object is immutable, the object itself can't be changed.

By default, Scala uses the immutable Set. If you want to use the mutable Set, you'll have to import **scala.collection.mutable.Set** class explicitly. If you want to use both mutable and immutable sets in the same, then you can continue to refer to the immutable Set as **Set** but you can refer to the mutable Set as **mutable.Set**. Following is the example to declare immutable Sets as follows:

```
// Empty set of integer type
var s : Set[Int] = Set()

// Set of integer type
var s : Set[Int] = Set(1,3,5,7)

or
```

```
var s = Set(1,3,5,7)
```

While defining empty set, the type annotation is necessary as the system needs to assign a concrete type to variable.

Basic Operations on Set:

All operations on sets can be expressed in terms of the following three methods:

Methods	Description
Head	This method returns the first element of a set.
Tail	This method returns a set consisting of all elements except the first.
isEmpty	This method returns true if the set is empty otherwise false.

Following is the example showing usage of the above methods:

```
object Test {
   def main(args: Array[String]) {
      val fruit = Set("apples", "oranges", "pears")
      val nums: Set[Int] = Set()

      println( "Head of fruit : " + fruit.head )
      println( "Tail of fruit : " + fruit.tail )
      println( "Check if fruit is empty : " + fruit.isEmpty )
      println( "Check if nums is empty : " + nums.isEmpty )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Head of fruit : apples
Tail of fruit : Set(oranges, pears)
Check if fruit is empty : false
Check if nums is empty : true

C:/>
```

Concatenating Sets:

You can use either **++** operator or **Set.++()** method to concatenate two or more sets, but while adding sets it will remove duplicate elements. Following is the example to concatenate two sets:

```
object Test {
   def main(args: Array[String]) {
      val fruit1 = Set("apples", "oranges", "pears")
      val fruit2 = Set("mangoes", "banana")

      // use two or more sets with ++ as operator
      var fruit = fruit1 ++ fruit2
      println( "fruit1 ++ fruit2 : " + fruit )

      // use two sets with ++ as method
      fruit = fruit1.++(fruit2)
      println( "fruit1.++(fruit2) : " + fruit )
```

```
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
fruit1 ++ fruit2 : Set(banana, apples, mangoes, pears, oranges)
fruit1.++(fruit2) : Set(banana, apples, mangoes, pears, oranges)

C:/>
```

Find max, min elements in Set:

You can use **Set.min** method to find out the minimum and **Set.max** method to find out the maximum of the elements available in a set. Following is the example to show the usage:

```
object Test {
   def main(args: Array[String]) {
      val num = Set(5,6,9,20,30,45)

      // find min and max of the elements
      println( "Min element in Set(5,6,9,20,30,45) : " + num.min )
      println( "Max element in Set(5,6,9,20,30,45) : " + num.max )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Min element in Set(5,6,9,20,30,45) : 5
Max element in Set(5,6,9,20,30,45) : 45

C:/>
```

Find common values in Sets:

You can use either **Set.&** method or **Set.intersect** method to find out the common values between two sets. Following is the example to show the usage:

```
object Test {
   def main(args: Array[String]) {
      val num1 = Set(5,6,9,20,30,45)
      val num2 = Set(50,60,9,20,35,55)

      // find common elements between two sets
      println( "num1.&(num2) : " + num1.&(num2) )
      println( "num1.intersect(num2) : " + num1.intersect(num2) )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
num1.&(num2) : Set(20, 9)
num1.intersect(num2) : Set(20, 9)
```

```
C:/>
```

Scala Set Methods:

Following are the important methods which you can use while playing with Sets. For a complete list of methods available, please check official documentation of Scala.

SN	Methods with Description
1	**def +(elem: A): Set[A]** Creates a new set with an additional element, unless the element is already present.
2	**def -(elem: A): Set[A]** Creates a new set with a given element removed from this set.
3	**def contains(elem: A): Boolean** Returns true if elem is contained in this set, false otherwise.
4	**def &(that: Set[A]): Set[A]** Returns a new set consisting of all elements that are both in this set and in the given set that.
5	**def &~(that: Set[A]): Set[A]** Returns the difference of this set and another set.
6	**def +(elem1: A, elem2: A, elems: A*): Set[A]** Creates a new immutable set with additional elements from the passed sets
7	**def ++(elems: A): Set[A]** Concatenates this immutable set with the elements of another collection to this immutable set.
8	**def -(elem1: A, elem2: A, elems: A*): Set[A]** Returns a new immutable set that contains all elements of the current immutable set except one less occurrence of each of the given argument elements.
9	**def addString(b: StringBuilder): StringBuilder** Appends all elements of this immutable set to a string builder.
10	**def addString(b: StringBuilder, sep: String): StringBuilder** Appends all elements of this immutable set to a string builder using a separator string.
11	**def apply(elem: A)** Tests if some element is contained in this set.
12	**def count(p: (A) => Boolean): Int** Counts the number of elements in the immutable set which satisfy a predicate.
13	**def copyToArray(xs: Array[A], start: Int, len: Int): Unit** Copies elements of this immutable set to an array.
14	**def diff(that: Set[A]): Set[A]** Computes the difference of this set and another set.
15	**def drop(n: Int): Set[A]]** Returns all elements except first n ones.
16	**def dropRight(n: Int): Set[A]** Returns all elements except last n ones.
17	**def dropWhile(p: (A) => Boolean): Set[A]**

	Drops longest prefix of elements that satisfy a predicate.
18	**def equals(that: Any): Boolean** The equals method for arbitrary sequences. Compares this sequence to some other object.
19	**def exists(p: (A) => Boolean): Boolean** Tests whether a predicate holds for some of the elements of this immutable set.
20	**def filter(p: (A) => Boolean): Set[A]** Returns all elements of this immutable set which satisfy a predicate.
21	**def find(p: (A) => Boolean): Option[A]** Finds the first element of the immutable set satisfying a predicate, if any.
22	**def forall(p: (A) => Boolean): Boolean** Tests whether a predicate holds for all elements of this immutable set.
23	**def foreach(f: (A) => Unit): Unit** Applies a function f to all elements of this immutable set.
24	**def head: A** Returns the first element of this immutable set.
25	**def init: Set[A]** Returns all elements except the last.
26	**def intersect(that: Set[A]): Set[A]** Computes the intersection between this set and another set.
27	**def isEmpty: Boolean** Tests if this set is empty.
28	**def iterator: Iterator[A]** Creates a new iterator over all elements contained in the iterable object.
29	**def last: A** Returns the last element.
30	**def map[B](f: (A) => B): immutable.Set[B]** Builds a new collection by applying a function to all elements of this immutable set.
31	**def max: A** Finds the largest element.
32	**def min: A** Finds the smallest element.
33	**def mkString: String** Displays all elements of this immutable set in a string.
34	**def mkString(sep: String): String** Displays all elements of this immutable set in a string using a separator string.
35	**def product: A** Returns the product of all elements of this immutable set with respect to the * operator in num.
36	**def size: Int** Returns the number of elements in this immutable set.
37	**def splitAt(n: Int): (Set[A], Set[A])** Returns a pair of immutable sets consisting of the first n elements of this immutable set, and the other

	elements.
38	**def subsetOf(that: Set[A]): Boolean** Returns true if this set is a subset of that, i.e. if every element of this set is also an element of that.
39	**def sum: A** Returns the sum of all elements of this immutable set with respect to the + operator in num.
40	**def tail: Set[A]** Returns a immutable set consisting of all elements of this immutable set except the first one.
41	**def take(n: Int): Set[A]** Returns first n elements.
42	**def takeRight(n: Int):Set[A]** Returns last n elements.
43	**def toArray: Array[A]** Returns an array containing all elements of this immutable set.
44	**def toBuffer[B >: A]: Buffer[B]** Returns a buffer containing all elements of this immutable set.
45	**def toList: List[A]** Returns a list containing all elements of this immutable set.
46	**def toMap[T, U]: Map[T, U]** Converts this immutable set to a map
47	**def toSeq: Seq[A]** Returns a seq containing all elements of this immutable set.
48	**def toString(): String** Returns a String representation of the object.

Scala Maps

Scala map is a collection of key/value pairs. Any value can be retrieved based on its key. Keys are unique in the Map, but values need not be unique. Maps are also called Hash tables. There are two kinds of Maps, the **immutable** and the **mutable**. The difference between mutable and immutable objects is that when an object is immutable, the object itself can't be changed.

By default, Scala uses the immutable Map. If you want to use the mutable Set, you'll have to import**scala.collection.mutable.Map** class explicitly. If you want to use both mutable and immutable Maps in the same, then you can continue to refer to the immutable Map as **Map** but you can refer to the mutable set as **mutable.Map**. Following is the example to declare immutable Maps as follows:

```
// Empty hash table whose keys are strings and values are integers:
var A:Map[Char,Int] = Map()

// A map with keys and values.
val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")
```

While defining empty map, the type annotation is necessary as the system needs to assign a concrete type to variable. If we want to add a key-value pair to a Map, we can use the operator + as follows:

```
A += ('I' -> 1)
A += ('J' -> 5)
A += ('K' -> 10)
A += ('L' -> 100)
```

Basic Operations on Map:

All operations on maps can be expressed in terms of the following three methods:

Methods	Description
keys	This method returns an iterable containing each key in the map.
values	This method returns an iterable containing each value in the map.
isEmpty	This method returns true if the map is empty otherwise false.

Following is the example showing usage of the above methods:

```
object Test {
   def main(args: Array[String]) {
      val colors = Map("red" -> "#FF0000",
                       "azure" -> "#F0FFFF",
                       "peru" -> "#CD853F")

      val nums: Map[Int, Int] = Map()

      println( "Keys in colors : " + colors.keys )
      println( "Values in colors : " + colors.values )
      println( "Check if colors is empty : " + colors.isEmpty )
      println( "Check if nums is empty : " + nums.isEmpty )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Keys in colors : Set(red, azure, peru)
Values in colors : MapLike(#FF0000, #F0FFFF, #CD853F)
Check if colors is empty : false
Check if nums is empty : true

C:/>
```

Concatenating Maps

You can use either **++** operator or **Map.++()** method to concatenate two or more Maps, but while adding Maps it will remove duplicate keys. Following is the example to concatenate two Maps:

```
object Test {
   def main(args: Array[String]) {
      val colors1 = Map("red" -> "#FF0000",
                        "azure" -> "#F0FFFF",
                        "peru" -> "#CD853F")
      val colors2 = Map("blue" -> "#0033FF",
                        "yellow" -> "#FFFF00",
                        "red" -> "#FF0000")

      // use two or more Maps with ++ as operator
      var colors = colors1 ++ colors2
      println( "colors1 ++ colors2 : " + colors )
```

```
      // use two maps with ++ as method
      colors = colors1.++(colors2)
      println( "colors1.++(colors2)) : " + colors )

   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
colors1 ++ colors2 : Map(blue -> #0033FF, azure -> #F0FFFF,
                     peru -> #CD853F, yellow -> #FFFF00, red -> #FF0000)
colors1.++(colors2)) : Map(blue -> #0033FF, azure -> #F0FFFF,
                     peru -> #CD853F, yellow -> #FFFF00, red -> #FF0000)

C:/>
```

Print Keys and Values from a Map:

You can iterate through the keys and values of a Map using foreach loop. Following is the example to show the usage:

```
object Test {
   def main(args: Array[String]) {
      val colors = Map("red" -> "#FF0000",
                       "azure" -> "#F0FFFF",
                       "peru" -> "#CD853F")

      colors.keys.foreach{ i =>
                           print( "Key = " + i )
                           println(" Value = " + colors(i) )}
   }
}
```

Here, we used method **foreach** associated with iterator to walk through the keys. When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
Key = red Value = #FF0000
Key = azure Value = #F0FFFF
Key = peru Value = #CD853F

C:/>
```

Check for a Key in Map:

You can use either **Map.contains** method to test if a given key exists in the map or not. Following is the example to show the usage:

```
object Test {
   def main(args: Array[String]) {
      val colors = Map("red" -> "#FF0000",
                       "azure" -> "#F0FFFF",
                       "peru" -> "#CD853F")

      if( colors.contains( "red" )){
```

```
            println("Red key exists with value :"  + colors("red"))
        }else{
            println("Red key does not exist")
        }
        if( colors.contains( "maroon" )){
            println("Maroon key exists with value :"  + colors("maroon"))
        }else{
            println("Maroon key does not exist")
        }
    }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Red key exists with value :#FF0000
Maroon key does not exist

C:/>
```

Scala Map Methods:

Following are the important methods which you can use while playing with Maps. For a complete list of methods available, please check official documentation of Scala.

SN	Methods with Description
1	**def ++(xs: Map[(A, B)]): Map[A, B]** Returns a new map containing mappings of this map and those provided by xs.
2	**def -(elem1: A, elem2: A, elems: A*): Map[A, B]** Returns a new map containing all the mappings of this map except mappings with a key equal to elem1, elem2 or any of elems.
3	**def --(xs: GTO[A]): Map[A, B]** Returns a new map with all the key/value mappings of this map except mappings with a key equal to a key from the traversable object xs.
4	**def get(key: A): Option[B]** Optionally returns the value associated with a key.
5	**def iterator: Iterator[(A, B)]** Creates a new iterator over all key/value pairs of this map
6	**def addString(b: StringBuilder): StringBuilder** Appends all elements of this shrinkable collection to a string builder.
7	**def addString(b: StringBuilder, sep: String): StringBuilder** Appends all elements of this shrinkable collection to a string builder using a separator string.
8	**def apply(key: A): B** Returns the value associated with the given key, or the result of the map's default method, if none exists.
9	**def clear(): Unit** Removes all bindings from the map. After this operation has completed, the map will be empty.
10	**def clone(): Map[A, B]** Creates a copy of the receiver object.

11	**def contains(key: A): Boolean** Returns true if there is a binding for key in this map, false otherwise.
12	**def copyToArray(xs: Array[(A, B)]): Unit** Copies values of this shrinkable collection to an array. Fills the given array xs with values of this shrinkable collection.
13	**def count(p: ((A, B)) => Boolean): Int** Counts the number of elements in the shrinkable collection which satisfy a predicate.
14	**def default(key: A): B** Defines the default value computation for the map, returned when a key is not found.
15	**def drop(n: Int): Map[A, B]** Returns all elements except first n ones.
16	**def dropRight(n: Int): Map[A, B]** Returns all elements except last n ones
17	**def dropWhile(p: ((A, B)) => Boolean): Map[A, B]** Drops longest prefix of elements that satisfy a predicate.
18	**def empty: Map[A, B]** Returns the empty map of the same type as this map.
19	**def equals(that: Any): Boolean** Returns true if both maps contain exactly the same keys/values, false otherwise.
20	**def exists(p: ((A, B)) => Boolean): Boolean** Returns true if the given predicate p holds for some of the elements of this shrinkable collection, otherwise false.
21	**def filter(p: ((A, B))=> Boolean): Map[A, B]** Returns all elements of this shrinkable collection which satisfy a predicate.
22	**def filterKeys(p: (A) => Boolean): Map[A, B]** Returns an immutable map consisting only of those key value pairs of this map where the key satisfies the predicate p.
23	**def find(p: ((A, B)) => Boolean): Option[(A, B)]** Finds the first element of the shrinkable collection satisfying a predicate, if any.
24	**def foreach(f: ((A, B)) => Unit): Unit** Applies a function f to all elements of this shrinkable collection.
25	**def init: Map[A, B]** Returns all elements except the last.
26	**def isEmpty: Boolean** Tests whether the map is empty.
27	**def keys: Iterable[A]** Returns an iterator over all keys.
28	**def last: (A, B)** Returns the last element.
29	**def max: (A, B)** Finds the largest element.
30	**def min: (A, B)**

	Finds the smallest element.
31	**def mkString: String** Displays all elements of this shrinkable collection in a string.
32	**def product: (A, B)** Returns the product of all elements of this shrinkable collection with respect to the * operator in num.
33	**def remove(key: A): Option[B]** Removes a key from this map, returning the value associated previously with that key as an option.
34	**def retain(p: (A, B) => Boolean): Map.this.type** Retains only those mappings for which the predicate p returns true.
35	**def size: Int** Return the number of elements in this map.
36	**def sum: (A, B)** Returns the sum of all elements of this shrinkable collection with respect to the + operator in num.
37	**def tail: Map[A, B]** Returns all elements except the first.
38	**def take(n: Int): Map[A, B]** Returns first n elements.
39	**def takeRight(n: Int): Map[A, B]** Returns last n elements.
40	**def takeWhile(p: ((A, B)) => Boolean): Map[A, B]** Takes longest prefix of elements that satisfy a predicate.
41	**def toArray: Array[(A, B)]** Converts this shrinkable collection to an array.
42	**def toBuffer[B >: A]: Buffer[B]** Returns a buffer containing all elements of this map.
43	**def toList: List[A]** Returns a list containing all elements of this map.
44	**def toSeq: Seq[A]** Returns a seq containing all elements of this map.
45	**def toSet: Set[A]** Returns a set containing all elements of this map.
46	**def toString(): String** Returns a String representation of the object.

Scala Tuples

Scala tuple combines a fixed number of items together so that they can be passed around as a whole. Unlike an array or list, a tuple can hold objects with different types but they are also immutable. Here is an example of a tuple holding an integer, a string, and the console:

```
val t = (1, "hello", Console)
```

Which is syntactic sugar (short cut) for the following:

```
val t = new Tuple3(1, "hello", Console)
```

The actual type of a tuple depends upon the number and of elements it contains and the types of those elements. Thus, the type of (99, "Luftballons") is Tuple2[Int, String]. The type of ('u', 'r', "the", 1, 4, "me") is Tuple6[Char, Char, String, Int, Int, String]

Tuples are of type Tuple1, Tuple2, Tuple3 and so on. There currently is an upper limit of 22 in the Scala if you need more, then you can use a collection, not a tuple. For each TupleN type, where 1 <= N <= 22, Scala defines a number of element-access methods. Given the following definition:

```
val t = (4,3,2,1)
```

To access elements of a tuple t, you can use method t._1 to access the first element, t._2 to access the second, and so on. For example, the following expression computes the sum of all elements of t:

```
val sum = t._1 + t._2 + t._3 + t._4
```

You can use Tupel to write a method that takes a List[Double] and returns the count, the sum, and the sum of squares returned in a three-element Tuple, a Tuple3[Int, Double, Double]. They are also useful to pass a list of data values as messages between actors in concurrent programming. Following is the example showing usage of a tuple:

```
object Test {
   def main(args: Array[String]) {
      val t = (4,3,2,1)

      val sum = t._1 + t._2 + t._3 + t._4

      println( "Sum of elements: "  + sum )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Sum of elements: 10

C:/>
```

Iterate over the Tuple:

You can use **Tuple.productIterator()** method to iterate over all the elements of a Tuple. Following is the example to concatenate two Maps:

```
object Test {
   def main(args: Array[String]) {
      val t = (4,3,2,1)

      t.productIterator.foreach{ i =>println("Value = " + i )}
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Value = 4
Value = 3
Value = 2
```

```
Value = 1

C:/>
```

Convert to String:

You can use **Tuple.toString()** method to concatenate all the elements of the tuple into a string. Following is the example to show the usage:

```
object Test {
   def main(args: Array[String]) {
      val t = new Tuple3(1, "hello", Console)

      println("Concatenated String: " + t.toString() )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Concatenated String: (1,hello,scala.Console$@281acd47)

C:/>
```

Swap the Elements:

You can use **Tuple.swap** method to swap the elements of a Tuple2. Following is the example to show the usage:

```
object Test {
   def main(args: Array[String]) {
      val t = new Tuple2("Scala", "hello")

      println("Swapped Tuple: " + t.swap )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Swapped tuple: (hello,Scala)

C:/>
```

Scala Options

Scala Option[T] is a container for zero or one element of a given type. An Option[T] can be either**Some[T]** or **None** object, which represents a missing value. For instance, the get method of Scala's Map produces Some(value) if a value corresponding to a given key has been found, or **None** if the given key is not defined in the Map. The Option type is used frequently in Scala programs and you can compare this to **null** value available in Java which indicate no value. For example, the get method of java.util.HashMap returns either a value stored in the HashMap, or null if no value was found.

Let's say we have a method that retrieves a record from the database based on a primary key:

```
def findPerson(key: Int): Option[Person]
```

The method will return Some[Person] if the record is found but None if the record is not found. Let us see a real example:

```
object Test {
   def main(args: Array[String]) {
      val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")

      println("capitals.get( \"France\" ) : " +  capitals.get( "France" ))
      println("capitals.get( \"India\" ) : " +  capitals.get( "India" ))
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
capitals.get( "France" ) : Some(Paris)
capitals.get( "India" ) : None

C:/>
```

The most common way to take optional values apart is through a pattern match. For instance:

```
object Test {
   def main(args: Array[String]) {
      val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")

      println("show(capitals.get( \"Japan\")) : " +
                                          show(capitals.get( "Japan")) )
      println("show(capitals.get( \"India\")) : " +
                                          show(capitals.get( "India")) )
   }

   def show(x: Option[String]) = x match {
      case Some(s) => s
      case None => "?"
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
show(capitals.get( "Japan")) : Tokyo
show(capitals.get( "India")) : ?

C:/>
```

Using getOrElse() Method:

Following is the example of showing how to use getOrElse() to access a value or a default when no value is present:

```
object Test {
   def main(args: Array[String]) {
      val a:Option[Int] = Some(5)
      val b:Option[Int] = None

      println("a.getOrElse(0): " + a.getOrElse(0) )
      println("b.getOrElse(10): " + b.getOrElse(10) )
```

```
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
a.getOrElse(0): 5
b.getOrElse(10): 10

C:/>
```

Using isEmpty() Method:

Following is the example of showing how to use isEmpty() to check if the option is None or not:

```
object Test {
   def main(args: Array[String]) {
      val a:Option[Int] = Some(5)
      val b:Option[Int] = None

      println("a.isEmpty: " + a.isEmpty )
      println("b.isEmpty: " + b.isEmpty )
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
a.isEmpty: false
b.isEmpty: true

C:/>
```

Scala Option Methods:

Following are the important methods which you can use while playing with Options. For a complete list of methods available, please check official documentation of Scala.

SN	Methods with Description
1	**def get: A** Returns the option's value.
2	**def isEmpty: Boolean** Returns true if the option is None, false otherwise.
3	**def productArity: Int** The size of this product. For a product A(x_1, ..., x_k), returns k
4	**def productElement(n: Int): Any** The nth element of this product, 0-based. In other words, for a product A(x_1, ..., x_k), returns x_(n+1) where 0 < n < k.
5	**def exists(p: (A) => Boolean): Boolean** Returns true if this option is nonempty and the predicate p returns true when applied to this Option's value. Otherwise, returns false.

6	**def filter(p: (A) => Boolean): Option[A]** Returns this Option if it is nonempty and applying the predicate p to this Option's value returns true. Otherwise, return None.
7	**def filterNot(p: (A) => Boolean): Option[A]** Returns this Option if it is nonempty and applying the predicate p to this Option's value returns false. Otherwise, return None.
8	**def flatMap[B](f: (A) => Option[B]): Option[B]** Returns the result of applying f to this Option's value if this Option is nonempty. Returns None if this Option is empty.
9	**def foreach[U](f: (A) => U): Unit** Apply the given procedure f to the option's value, if it is nonempty. Otherwise, do nothing.
10	**def getOrElse[B >: A](default: => B): B** Returns the option's value if the option is nonempty, otherwise return the result of evaluating default.
11	**def isDefined: Boolean** Returns true if the option is an instance of Some, false otherwise.
12	**def iterator: Iterator[A]** Returns a singleton iterator returning the Option's value if it is nonempty, or an empty iterator if the option is empty.
13	**def map[B](f: (A) => B): Option[B]** Returns a Some containing the result of applying f to this Option's value if this Option is nonempty. Otherwise return None.
14	**def orElse[B >: A](alternative: => Option[B]): Option[B]** Returns this Option if it is nonempty, otherwise return the result of evaluating alternative.
15	**def orNull** Returns the option's value if it is nonempty, or null if it is empty.

Scala Iterators

An iterator is not a collection, but rather a way to access the elements of a collection one by one. The two basic operations on an iterator it are **next** and **hasNext**. A call to **it.next()** will return the next element of the iterator and advance the state of the iterator. You can find out whether there are more elements to return using Iterator's **it.hasNext** method.

The most straightforward way to "step through" all the elements returned by an iterator is to use a while loop. Let us see a real example:

```
object Test {
   def main(args: Array[String]) {
      val it = Iterator("a", "number", "of", "words")

      while (it.hasNext){
         println(it.next())
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
a
number
```

```
of
words

C:/>
```

Find Min & Max valued Element:

You can use **it.min** and **it.max** methods to find out the minimum and maximum valued elements from an iterator. Following is the usage:

```
object Test {
   def main(args: Array[String]) {
      val ita = Iterator(20,40,2,50,69, 90)
      val itb = Iterator(20,40,2,50,69, 90)

      println("Maximum valued element " + ita.max )
      println("Minimum valued element " + itb.min )
   }
}
```

Here, we used ita and itb to perform two different operations because iterator can be traversed only once. When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
Maximum valued element 90
Minimum valued element 2

C:/>
```

Find the length of the Iterator:

You can use either **it.size** or **it.length** methods to find out the number of elements available in an iterator. Following is the usage:

```
object Test {
   def main(args: Array[String]) {
      val ita = Iterator(20,40,2,50,69, 90)
      val itb = Iterator(20,40,2,50,69, 90)

      println("Value of ita.size : " + ita.size )
      println("Value of itb.length : " + itb.length )

   }
}
```

Here, we used ita and itb to perform two different operations because iterator can be traversed only once. When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
Value of ita.size : 6
Value of itb.length : 6

C:/>
```

Scala Iterator Methods:

Following are the important methods which you can use while playing with Iterator. For a complete list of methods available, please check official documentation of Scala.

SN	Methods with Description
1	**def hasNext: Boolean** Tests whether this iterator can provide another element.
2	**def next(): A** Produces the next element of this iterator.
3	**def ++(that: => Iterator[A]): Iterator[A]** Concatenates this iterator with another.
4	**def ++[B >: A](that :=> GenTraversableOnce[B]): Iterator[B]** Concatenates this iterator with another.
5	**def addString(b: StringBuilder): StringBuilder** Returns the string builder b to which elements were appended.
6	**def addString(b: StringBuilder, sep: String): StringBuilder** Returns the string builder b to which elements were appended using a separator string.
7	**def buffered: BufferedIterator[A]** Creates a buffered iterator from this iterator.
8	**def contains(elem: Any): Boolean** Tests whether this iterator contains a given value as an element.
9	**def copyToArray(xs: Array[A], start: Int, len: Int): Unit** Copies selected values produced by this iterator to an array.
10	**def count(p: (A) => Boolean): Int** Counts the number of elements in the traversable or iterator which satisfy a predicate.
11	**def drop(n: Int): Iterator[A]** Advances this iterator past the first n elements, or the length of the iterator, whichever is smaller.
12	**def dropWhile(p: (A) => Boolean): Iterator[A]** Skips longest sequence of elements of this iterator which satisfy given predicate p, and returns an iterator of the remaining elements.
13	**def duplicate: (Iterator[A], Iterator[A])** Creates two new iterators that both iterate over the same elements as this iterator (in the same order).
14	**def exists(p: (A) => Boolean): Boolean** Returns true if the given predicate p holds for some of the values produced by this iterator, otherwise false.
15	**def filter(p: (A) => Boolean): Iterator[A]** Returns an iterator over all the elements of this iterator that satisfy the predicate p. The order of the elements is preserved.
16	**def filterNot(p: (A) => Boolean): Iterator[A]** Creates an iterator over all the elements of this iterator which do not satisfy a predicate p.
17	**def find(p: (A) => Boolean): Option[A]** Finds the first value produced by the iterator satisfying a predicate, if any.

18	**def flatMap[B](f: (A) => GenTraversableOnce[B]): Iterator[B]** Creates a new iterator by applying a function to all values produced by this iterator and concatenating the results.
19	**def forall(p: (A) => Boolean): Boolean** Returns true if the given predicate p holds for all values produced by this iterator, otherwise false.
20	**def foreach(f: (A) => Unit): Unit** Applies a function f to all values produced by this iterator.
21	**def hasDefiniteSize: Boolean** Returns true for empty Iterators, false otherwise.
22	**def indexOf(elem: B): Int** Returns the index of the first occurrence of the specified object in this iterable object.
23	**def indexWhere(p: (A) => Boolean): Int** Returns the index of the first produced value satisfying a predicate, or -1.
24	**def isEmpty: Boolean** Returns true if hasNext is false, false otherwise.
25	**def isTraversableAgain: Boolean** Tests whether this Iterator can be repeatedly traversed.
26	**def length: Int** Returns the number of elements in this iterator. The iterator is at its end after this method returns.
27	**def map[B](f: (A) => B): Iterator[B]** Returns a new iterator which transforms every value produced by this iterator by applying the function f to it.
28	**def max: A** Finds the largest element. The iterator is at its end after this method returns.
29	**def min: A** Finds the minumum element. The iterator is at its end after this method returns.
30	**def mkString: String** Displays all elements of this traversable or iterator in a string.
31	**def mkString(sep: String): String** Displays all elements of this traversable or iterator in a string using a separator string.
32	**def nonEmpty: Boolean** Tests whether the traversable or iterator is not empty.
33	**def padTo(len: Int, elem: A): Iterator[A]** Appends an element value to this iterator until a given target length is reached.
34	**def patch(from: Int, patchElems: Iterator[B], replaced: Int): Iterator[B]** Returns this iterator with patched values.
35	**def product: A** Multiplies up the elements of this collection.
36	**def sameElements(that: Iterator[_]): Boolean** Returns true, if both iterators produce the same elements in the same order, false otherwise.
37	**def seq: Iterator[A]** Returns a sequential view of the collection.

38	**def size: Int** Returns the number of elements in this traversable or iterator.
39	**def slice(from: Int, until: Int): Iterator[A]** Creates an iterator returning an interval of the values produced by this iterator.
40	**def sum: A** Returns the sum of all elements of this traversable or iterator with respect to the + operator in num.
41	**def take(n: Int): Iterator[A]** Returns an iterator producing only of the first n values of this iterator, or else the whole iterator, if it produces fewer than n values.
42	**def toArray: Array[A]** Returns an array containing all elements of this traversable or iterator.
43	**def toBuffer: Buffer[B]** Returns a buffer containing all elements of this traversable or iterator.
44	**def toIterable: Iterable[A]** Returns an Iterable containing all elements of this traversable or iterator. This will not terminate for infinite iterators.
45	**def toIterator: Iterator[A]** Returns an Iterator containing all elements of this traversable or iterator. This will not terminate for infinite iterators.
46	**def toList: List[A]** Returns a list containing all elements of this traversable or iterator.
47	**def toMap[T, U]: Map[T, U]** Returns a map containing all elements of this traversable or iterator.
48	**def toSeq: Seq[A]** Returns a sequence containing all elements of this traversable or iterator.
49	**def toString(): String** Converts this iterator to a string.
50	**def zip[B](that: Iterator[B]): Iterator[(A, B)** Returns a new iterator containing pairs consisting of corresponding elements of this iterator and that. The number of elements returned by the new iterator is the minimum of the number of elements returned by this iterator and that.

Example:

Following code snippet is a simple example to define all the above type of collections:

```
// Define List of integers.
val x = List(1,2,3,4)

// Define a set.
var x = Set(1,3,5,7)

// Define a map.
val x = Map("one" -> 1, "two" -> 2, "three" -> 3)

// Create a tuple of two elements.
val x = (10, "Scala")
```

```
// Define an option
val x:Option[Int] = Some(5)
```

CHAPTER

15

Scala Classes & Objects

A class is a blueprint for objects. Once you define a class, you can create objects from the class blueprint with the keyword **new**. Following is a simple syntax to define a class in Scala:

```
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);
   }
}
```

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.

As mentioned earlier, you can create objects using a keyword **new** and then you can access class fields and methods as shown below in the 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 Test {
   def main(args: Array[String]) {
      val pt = new Point(10, 20);

      // Move to a new location
```

```
      pt.move(10, 10);
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Point x location : 20
Point y location : 30

C:/>
```

Extending a Class:

You can extend a base scala class in similar way you can do it in Java 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:

```
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);
   }
}
```

Such an **extends** clause has two effects: it makes class Location inherit all non-private members from class Point, 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. Let us take complete example showing inheirtance:

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

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

Note that methods move and **move** 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). When the above code is compiled and executed, it produces the following result:

```
C:/>scalac Test.scala
C:/>scala Test
Point x location : 20
Point y location : 30
Point z location : 20

C:/>
```

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 of showing singleton:

```
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 Test {
   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);
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Point x location : 10
Point y location : 20

C:/>
```

CHAPTER

16

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** as follows:

```
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. Below is a complete example to show the concept of traits:

```
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 == x
}

object Test {
   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))
```

```
    }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
false
true
true

C:/>
```

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.

CHAPTER

17

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 for 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. Here is a small example, which shows how to match against an integer value:

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

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

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

```
C:/>scalac Test.scala
C:/>scala Test
many

C:/>
```

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. Following is a second example, which matches a value against patterns of different types:

```
object Test {
   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"
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
2
many
one

C:/>
```

The first case matches if x refers to the integer value 1. The second case matches if x is equal to the string"two". The third case consists of a typed pattern; it matches against any integer and binds the selector value xto the variable y of type integer. Following is another form of writing same match...case expressions with the help of braces {...}:

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

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

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**. Following is a simple pattern matching example using case class:

```
object Test {
   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, empty one.
   case class Person(name: String, age: Int)
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Hi Alice!
Hi Bob!
Age: 32 year, name: Charlie?

C:/>
```

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 is gone because there are no methods that we need to define!

CHAPTER

18

Scala Regular Expressions

Scala supports regular expressions through **Regex** class available in the scala.util.matching package. Let us check an example where we will try to find out word **Scala** from a statement:

```
import scala.util.matching.Regex

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

      println(pattern findFirstIn str)
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Some(Scala)

C:/>
```

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 as follows:

```
import scala.util.matching.Regex

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

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

```
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Scala,scala

C:/>
```

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

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

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

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

```
C:/>scalac Test.scala
C:/>scala Test
Java is scalable and cool

C:/>
```

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:

Here is the table listing down all the regular expression metacharacter 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. Check the following example:

```
import scala.util.matching.Regex

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

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

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

```
C:/>scalac Test.scala
```

```
C:/>scala Test
able1

C:/>
```

CHAPTER

19

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:

```
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 as shown below:

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

object Test {
   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")
         }
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Missing file exception

C:/>
```

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.

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

object Test {
   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...")
      }
   }
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
Missing file exception
Exiting finally...

C:/>
```

CHAPTER

20

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.

Following example shows an extractor object for email addresses:

```
object Test {
   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
      }
   }
}
```

This 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:

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

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

```
C:/>scalac Test.scala
C:/>scala Test
Apply method : Zara@gmail.com
Unapply method : Some((Zara,gmail.com))
Unapply method : None

C:/>
```

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 as shown below:

```
object Test {
   def main(args: Array[String]) {

      val x = Test(5)
      //apply is invoked... the value 10 is returend
      println(x)

      x match
      {
         case Test(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
}
```

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

```
C:/>scalac Test.scala
C:/>scala Test
10
10 is bigger two times than 5

C:/>
```

CHAPTER

21

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. Following is an example of writing to a file:

```
import java.io._

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

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

When the above code is compiled and executed, it creates a file with "Hello Scala" content which you can check yourself.

```
C:/>scalac Test.scala
C:/>scala Test

C:/>
```

Reading line from Screen:

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

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

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

When the above code is compiled and executed, it prompts you to enter your input and it continues until you press ENTER key.

```
C:/>scalac Test.scala
```

```
C:/>scala Test
scala Test
Please enter your input : Scala is great
Thanks, you just typed: Scala is great

C:/>
```

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 "test.txt" file which we created earlier:

```
import scala.io.Source

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

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

When the above code is compiled and executed, it will read test.txt file and display the content on the screen:

```
C:/>scalac Test.scala
C:/>scala Test
scala Test
Following is the content read:
Hello Scala

C:/>
```