Scala

* short for Scalable Language
* a hybrid functional programming language - both object oriented and functional
  + Scala is a pure object-oriented language in the sense that every value is an object.
  + Scala is also a functional language in the sense that every function is a value and every value is an object so ultimately every function is an object.
  + Scala provides a lightweight syntax for defining **anonymous functions**, it supports **higher-order functions**, it allows functions to be **nested**, and supports **currying**.
* It is a strong static type language.
* In scala, everything is an object whether it is a function or a number. It does not have concept of primitive data.
* File extension of scala source file may be either .scala or .sc.
* Scala runs on the JVM:
  + Scala is compiled into Java Byte Code which is executed by the Java Virtual Machine (JVM). This means that Scala and Java have a common runtime platform.
  + 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.
* You can create any kind of application like web application, enterprise application, mobile application, desktop based application etc.
* ; is optional at the end

**Features**

**Type Inference**

In Scala, you don't require to mention data type and function return type explicitly. Scala is enough smart to deduce the type of data. The return type of function is determined by the type of last expression present in the function.

**Singleton object**

In Scala, there are no static variables or methods. Scala uses singleton object, which is essentially class with only one object in the source file. Singleton object is declared by using object instead of class keyword.

**Immutability**

Scala uses immutability concept. Each declared variable is immutable by default. Immutable means you can't modify its value. You can also create mutable variables which can be changed.

**Lazy Computation**

In Scala, computation is lazy by default. Scala evaluates expressions only when they are required. You can declare a lazy variable by using lazy keyword. It is used to increase performance.

**Case classes and Pattern matching**

Scala case classes are just regular classes which are immutable by default and decomposable through pattern matching.

All the parameters listed in the case class are public and immutable by default.

Case classes support pattern matching. So, you can write more logical code.

**Concurrency control**

Scala provides standard library which includes the actor model. You can write concurrency code by using actor. Scala provides one more platform and tool to deal with concurrency known as Akka. Akka is a separate open source framework that provides actor-based concurrency. Akka actors may be distributed or combined with software transactional memory.

**String Interpolation**

Since Scala 2.10.0, Scala offers a new mechanism to create strings from your data. It is called string interpolation. String interpolation allows users to embed variable references directly in processed string literals. Scala provides three string interpolation methods: s, f and raw.

**Higher Order Functions**

Higher order function is a function that either takes a function as argument or returns a function. In other words, we can say a function which works with another function is called higher order function. Higher order function allows you to create function composition, lambda function or anonymous function etc.

**Traits**

A trait is like an interface with a partial implementation. In Scala, trait is a collection of abstract and non-abstract methods. You can create trait that can have all abstract methods or some abstract and some non-abstract methods.

Traits are compiled into Java interfaces with corresponding implementation classes that hold any methods implemented in the traits.

**Rich Set of Collection**

Scala provides rich set of collection library. It contains classes and traits to collect data. These collections can be mutable or immutable. You can use it according to your requirement. Scala.collection.mutable package contains all the mutable collections. You can add, remove and update data while using this package.

Scala.collection.immutable package contains all the immutable collections. It does not allow you to modify data.

**Programming:**

**Two modes**

1. **Interactive:** open cmd, and type scala. You can now directly run commands on it. (like pig interactive)

println("Hello, Scala!")

1. **Scripting:** create a .sc or .scala file (using notepad), compile on cmd using scalac, run using scala (Like java commandline)

object HelloWorld {

def main(args: Array[String]) {

println("Hello, world!") // prints Hello World

}

}

## Basic Syntax

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

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

**Example** − class MyFirstScalaClass.

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

**Example** − def myMethodName()

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

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

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

## Scala Identifiers

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

### Alphanumeric Identifiers

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

Following are **legal alphanumeric identifiers** −

age, salary, \_value, \_\_1\_value

Following are **illegal identifiers** −

$salary, 123abc, -salary

### Operator Identifiers

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

Following are legal operator identifiers −

+ ++ ::: <?> :>

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

### Mixed Identifiers

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

Following are legal mixed identifiers −

unary\_+, myvar\_=

Here, unary\_+ used as a method name defines a unary + operator and myvar\_= used as method name defines an assignment operator (operator overloading).

### Literal Identifiers

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

Following are legal literal identifiers −

`x` `<clinit>` `yield`

## Scala Keywords

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

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

## Comments in Scala

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

object HelloWorld {

/\* This is my first java program.

\* This will print 'Hello World' as the output

\* This is an example of multi-line comments.

\*/

def main(args: Array[String]) {

// Prints Hello World

// This is also an example of single line comment.

println("Hello, world!")

}

}

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

|  |  |
| --- | --- |
| **Sr.No** | **Data Type & Description** |
| 1 | **Byte**  8 bit signed value. Range from -128 to 127 |
| 2 | **Short**  16 bit signed value. Range -32768 to 32767 |
| 3 | **Int**  32 bit signed value. Range -2147483648 to 2147483647 |
| 4 | **Long**  64 bit signed value. -9223372036854775808 to 9223372036854775807 |
| 5 | **Float**  32 bit IEEE 754 single-precision float |
| 6 | **Double**  64 bit IEEE 754 double-precision float |
| 7 | **Char**  16 bit unsigned Unicode character. Range from U+0000 to U+FFFF |
| 8 | **String**  A sequence of Chars |
| 9 | **Boolean**  Either the literal true or the literal false |
| 10 | **Unit**  Corresponds to no value |
| 11 | **Null**  null or empty reference |
| 12 | **Nothing**  The subtype of every other type; includes no values |
| 13 | **Any**  The supertype of any type; any object is of type *Any* |
| 14 | **AnyRef**  The supertype of any reference type |

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

## Scala Basic Literals

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

### Integral Literals

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

0

035

21

0xFFFFFFFF

0777L

### Floating Point Literal

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

0.0

1e30f

3.14159f

1.0e100

.1

### Boolean Literals

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

### Symbol Literals

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

package scala

final case class Symbol private (name: String) {

override def toString: String = "'" + name

}

### Character Literals

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

'a'

'\u0041'

'\n'

'\t'

### String Literals

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

"Hello,\nWorld!"

"This string contains a \" character."

### Multi-Line Strings

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

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

"""the present string

spans three

lines."""

### Null Values

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

## Escape Sequences

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

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

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

**Variables**

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

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

## Variable Declaration

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

### Syntax

var myVar : String = "Foo"

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

### Syntax

val myVal : String = "Foo"

## Variable Data Types

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

### Syntax

val or val VariableName : DataType = [Initial Value]

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

### Syntax

var myVar :Int;

val myVal :String;

## Variable Type Inference

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

### Syntax

var myVar = 10;

val myVal = "Hello, Scala!";

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

## Multiple assignments

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

### Syntax

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

And the type inference gets it right −

### Syntax

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

## Example Program

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

### Example

object Demo {

def main(args: Array[String]) {

var myVar :Int = 10;

val myVal :String = "Hello Scala with datatype declaration.";

var myVar1 = 20;

val myVal1 = "Hello Scala new without datatype declaration.";

println(myVar); println(myVal); println(myVar1);

println(myVal1);

}

}

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

### Command

\>scalac Demo.scala

\>scala Demo

### Output

10

Hello Scala with datatype declaration.

20

Hello Scala without datatype declaration.

## Variable Scope

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

### Fields

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

### Method Parameters

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

### Local Variables

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

## Basic Class

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

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

### Syntax

class Point(xc: Int, yc: Int) {

var x: Int = xc

var y: Int = yc

def move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

println ("Point x location : " + x);

println ("Point y location : " + y);

}

}

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

### Example

import java.io.\_

class Point(val xc: Int, val yc: Int) {

var x: Int = xc

var y: Int = yc

def move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

println ("Point x location : " + x);

println ("Point y location : " + y);

}

}

object Demo {

def main(args: Array[String]) {

val pt = new Point(10, 20);

// Move to a new location

pt.move(10, 10);

}

}

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

### Command

\>scalac Demo.scala

\>scala Demo

### Output

Point x location : 20

Point y location : 30

## Extending a Class

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

### Example

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

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

Try the following example program to implement inheritance.

import java.io.\_

class Point(val xc: Int, val yc: Int) {

var x: Int = xc

var y: Int = yc

def move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

println ("Point x location : " + x);

println ("Point y location : " + y);

}

}

class Location(override val xc: Int, override val yc: Int,

val zc :Int) extends Point(xc, yc){

var z: Int = zc

def move(dx: Int, dy: Int, dz: Int) {

x = x + dx

y = y + dy

z = z + dz

println ("Point x location : " + x);

println ("Point y location : " + y);

println ("Point z location : " + z);

}

}

object Demo {

def main(args: Array[String]) {

val loc = new Location(10, 20, 15);

// Move to a new location

loc.move(10, 10, 5);

}

}

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

### Command

\>scalac Demo.scala

\>scala Demo

### Output

Point x location : 20

Point y location : 30

Point z location : 20

## Implicit Classes

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

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

### Syntax

object <object name> {

implicit class <class name>(<Variable>: Data type) {

def <method>(): Unit =

}

}

### Example

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

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

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

object Run {

implicit class IntTimes(x: Int) {

def times [A](f: =>A): Unit = {

def loop(current: Int): Unit =

if(current > 0){

f

loop(current - 1)

}

loop(x)

}

}

}

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

import Run.\_

object Demo {

def main(args: Array[String]) {

4 times println("hello")

}

}

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

Hello

Hello

Hello

Hello

**Note** −

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

## Singleton Objects

Scala is more object-oriented than Java because in Scala, we cannot have static members. Instead, Scala has **singleton objects**. A singleton is a class that can have only one instance, i.e., Object. You create singleton using the keyword **object** instead of class keyword. Since you can't instantiate a singleton object, you can't pass parameters to the primary constructor. You already have seen all the examples using singleton objects where you called Scala's main method.

Following is the same example program to implement singleton.

### Example

import java.io.\_

class Point(val xc: Int, val yc: Int) {

var x: Int = xc

var y: Int = yc

def move(dx: Int, dy: Int) {

x = x + dx

y = y + dy

}

}

object Demo {

def main(args: Array[String]) {

val point = new Point(10, 20)

printPoint

def printPoint{

println ("Point x location : " + point.x);

println ("Point y location : " + point.y);

}

}

}

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

### Command

\>scalac Demo.scala

\>scala Demo

### Output

Point x location : 10

Point y location : 20