Scala tutorial

# Overview

**Scala is object-oriented**

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

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

**Scala is functional**

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

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

**Scala is statically typed**

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

**Scala runs on the JVM**

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

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

**Scala can Execute Java Code**

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

Scala can do Concurrent & Synchronize processing

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

Scala vs Java

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

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

# 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.

# Useful

* In Scala, the \_ character is a “wildcard,” like \* in Java
* Scala has functions in addition to methods. It is simpler to use mathematical

functions such as min or pow in Scala than in Java—you need not call static

methods from a class.

sqrt(2) // Yields 1.4142135623730951

pow(2, 4) // Yields 16.0

min(3, Pi) // Yields 3.0

The mathematical functions are defined in the scala.math package

* Scala doesn’t have static methods, but it has a similar feature, called singleton

Objects.

* Java programmers use Javadoc to navigate the Java API. Scala has its own variant,

called **Scaladoc**

**“apply” method**

You can think of this as an overloaded form of the () operator. It is implemented

as a method with the name apply. For example, in the documentation of the StringOps

class, you will find a method

def apply(n: Int): Char

That is, "Hello"(4) is a shortcut for

"Hello".apply(4)

# Loops

If you need a loop, you have two choices. You can use a **while** loop. Or, you can use a **for** statement like this:

for (i <- 1 to n)

r = r \* i

When traversing a string or array, you often need a range from 0 to n – 1. In that case, use the until method instead of the to method. It returns a range that doesn’t include the upper bound.

val s = "Hello"

var sum = 0

for (i <- 0 until s.length) // Last value for i is s.length - 1

sum += s(i)

# Arithmetic Operators

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | | **Description** | | **Example** |
| + | | Adds two operands | | A + B will give 30 |
| - | | Subtracts second operand from the first | | A - B will give -10 |
| \* | | Multiplies both operands | | A \* B will give 200 |
| / | | Divides numerator by de-numerator | | B / A will give 2 |
| % | | Modulus operator finds the remainder after division of one number by another | | B % A will give 0 |
| **Operator** | **Description** | | **Example** | |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | | (A & B) will give 12, which is 0000 1100 | |
| | | Binary OR Operator copies a bit if it exists in either operand. | | (A | B) will give 61, which is 0011 1101 | |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | | (A ^ B) will give 49, which is 0011 0001 | |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | | (~A ) will give -61, which is 1100 0011 in 2's complement form due to a signed binary number. | |
| << | Binary Left Shift Operator. The bit positions of the left operands value is moved left by the number of bits specified by the right operand. | | A << 2 will give 240, which is 1111 0000 | |
| >> | Binary Right Shift Operator. The Bit positions of the left operand value is moved right by the number of bits specified by the right operand. | | A >> 2 will give 15, which is 1111 | |
| >>> | Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. | | A >>>2 will give 15 which is 0000 1111 | |

# Assignment Operators

|  |  |  |
| --- | --- | --- |
| **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 |

# Operations in Class mutable.Set

| **WHAT IT IS** | **WHAT IT DOES** |
| --- | --- |
| **Additions:** |  |
| xs += x | Adds element x to set xs as a side effect and returns xsitself. |
| xs += (x, y, z) | Adds the given elements to set xs as a side effect and returns xs itself. |
| xs ++= ys | Adds all elements in ys to set xs as a side effect and returns xs itself. |
| xs add x | Adds element x to xs and returns true if x was not previously contained in the set, false if it was. |
| **Removals:** |  |
| xs -= x | Removes element x from set xs as a side effect and returns xs itself. |
| xs -= (x, y, z) | Removes the given elements from set xs as a side effect and returns xs itself. |
| xs --= ys | Removes all elements in ys from set xs as a side effect and returns xs itself. |
| xs remove x | Removes element x from xs and returns true if x was previously contained in the set, false if it was not. |
| xs retain p | Keeps only those elements in xs that satisfy predicate p. |
| xs.clear() | Removes all elements from xs. |
| **Update:** |  |
| xs(x) = b | (or, written out, xs.update(x, b)). If boolean argument b is true, adds x to xs, otherwise removes x from xs. |
| **Cloning:** |  |
| xs.clone | A new mutable set with the same elements as xs. |

Just like an immutable set, a mutable set offers the + and ++ operations for element additions and the - and -- operations for element removals. But these are less often used for mutable sets since they involve copying the set. As a more efficient alternative, mutable sets offer the update methods += and -=

# Final and “sealed” class

A final class cannot be extended.

A sealed trait can only be extended in the same source file as it's declared. This is useful for creating ADTs (algebraic data types). An ADT is defined by the sum of its derived types.

# Functions

Functions are expressions that take parameters. Scala has functions in addition to methods. A method operates on an object, but a function doesn’t.

To define a function, you specify the function’s name, parameters, and body like this:

def abs(x: Double) = if (x >= 0) x else -x

You can define an anonymous function (i.e. no name) that returns a given integer plus one:

(x: Int) => x + 1

On the left of => is a list of parameters. On the right is an expression involving the parameters.

You can also name functions.

val addOne = (x: Int) => x + 1

println(addOne(1)) // 2

Functions may take multiple parameters.

val add = (x: Int, y: Int) => x + y

println(add(1, 2)) // 3

Or it can take no parameters.

val getTheAnswer = () => 42

println(getTheAnswer()) // 42

# Set

Sets are Iterables that contain no duplicate elements.

val fruit = Set("apple", "orange", "peach", "banana")

fruit: scala.collection.immutable.Set[java.lang.String] = Set(apple, orange, peach, banana)

# Seq

The **[Seq](http://www.scala-lang.org/api/current/scala/collection/Seq.html)** trait represents sequences. A sequence is a kind of iterable that has a length and whose elements have fixed index positions, starting from 0

val nums = Seq(1, 2, 3)

case class Person(name: String)

val people = Seq(

Person("Emily"),

Person("Hannah"),

Person("Mercedes")

)

# Scala ‘yield’

Scala offers a lightweight notation for expressing sequence comprehensions. Comprehensions have the form for (enumerators) yield e, where enumerators refers to a semicolon-separated list of enumerators. An enumerator is either a generator which introduces new variables, or it is a filter. A comprehension evaluates the body e for each binding generated by the enumerators and returns a sequence of these values

Example:

**case** **class** **User**(name: **String**, age: **Int**)

**val** userBase = **List**(**User**("Travis", 28),

**User**("Kelly", 33),

**User**("Jennifer", 44),

**User**("Dennis", 23))

**val** twentySomethings = **for** (user <- userBase **if** (user.age >=20 && user.age < 30))

**yield** user.name *// i.e. add this to a list*

twentySomethings.foreach(name => println(name)) *// prints Travis Dennis*

# Futures and Promises

Futures provide a way to reason about performing many operations in parallel– in an efficient and non-blocking way. A [Future](https://www.scala-lang.org/api/current/scala/concurrent/Future.html) is a placeholder object for a value that may not yet exist. Generally, the value of the Future is supplied concurrently and can subsequently be used. Composing concurrent tasks in this way tends to result in faster, asynchronous, non-blocking parallel code.

By default, futures and promises are non-blocking, making use of callbacks instead of typical blocking operations. To simplify the use of callbacks both syntactically and conceptually, Scala provides combinators such as flatMap, foreach, and filter used to compose futures in a non-blocking way. Blocking is still possible - for cases where it is absolutely necessary, futures can be blocked on (although this is discouraged).

A typical future looks like this:

val inverseFuture: Future[Matrix] = Future {

fatMatrix.inverse() // non-blocking long lasting computation

}(executionContext)

# The map function

The **map** function is applicable to both Scala's [Mutable](http://allaboutscala.com/tutorials/chapter-7-beginner-tutorial-using-scala-mutable-collection/) and [Immutable](http://allaboutscala.com/tutorials/chapter-6-beginner-tutorial-using-scala-immutable-collection/)collection data structures.

The **map** method takes a predicate function and applies it to every element in the collection. It creates a new collection with the result of the predicate function applied to each and every element of the collection.

As per the Scala documentation, the definition of the **map** method is as follows:

def map[B](f: (A) ⇒ B): Traversable[B]

**How to initialize a Sequence of donuts**

val donuts1: Seq[String] = Seq("Plain", "Strawberry", "Glazed")

println(s"Elements of donuts1 = $donuts1")

**How to append the word Donut to each element using the map function**

The code below shows how to use the **map** method to append the String Donut to each donut element in the Sequence.

println("\nStep 2: How to append the word Donut to each element using the map function")

val donuts2: Seq[String] = donuts1.map(\_ + " Donut")

println(s"Elements of donuts2 = $donuts2")

You should see the following output when you run your Scala application in IntelliJ:

Step 2: How to append the word Donut to each element using the map function

Elements of donuts2 = List(Plain Donut, Strawberry Donut, Glazed Donut)

# Scala “trait”

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

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

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

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.

# Scala “currying”

Currying is the technique of transforming a function with multiple arguments into a function with just one argument. The single argument is the value of the first argument from the original function and the function returns another single argument function. This in turn would take the second original argument and itself return another single argument function. This chaining continues over the number of arguments of the original. The last in the chain will have access to all of the arguments and so can do whatever it needs to do.

You can turn any function with multiple arguments into it’s curried equivalent.

In Scala, the regular uncurried function would look like this.

|  |
| --- |
| def add(x: Int, y: Int): Int = {  x + y  } |

As Scala supports curried functions, you can turn this into it’s curried version simply by separating out the arguments.

|  |
| --- |
| // shorthand  def add(x: Int)(y: Int): Int = {  x + y  } |

Which is shorthand for writing it out like this:

|  |
| --- |
| // longhand  def add(x: Int): (Int => Int) = {  (y: Int) => {  x + y  }  } |

Using the REPL to show how they’re called;

|  |
| --- |
| scala> def add(x: Int)(y: Int): Int = {  | x + y  | }  add: (x: Int)(y: Int)Int  scala> add(1) \_  res1: Int => Int = <Function>  scala> (add(1) \_).apply(1)  res2: Int = 2  scala> add(1)(1)  res3: Int = 2 |