BASICS

In this page, we will cover the basics of Scala.

Trying Scala in the Browser

You can run Scala in your browser with *ScalaFiddle*. This is an easy, zero-setup way to experiment with pieces of Scala code:

1. Go to https://scalafiddle.io.
2. Paste `println("Hello, world!")` in the left pane.
3. Click **Run**. The output appears in the right pane.

ScalaFiddle is integrated with some of the code examples in this documentation; if you see a **Run** button in a code example below, click it to directly experiment with the code.

Expressions

Expressions are computable statements:

```
1 + 1
```

You can output the results of expressions using `println`:

```
println(1) // 1
println(1 + 1) // 2
println("Hello!") // Hello!
println("Hello," + " world!") // Hello, world!
```

Values

You can name the results of expressions using the `val` keyword:

```
val x = 1 + 1
println(x) // 2
```

Named results, such as `x` here, are called values. Referencing a value does not re-compute it.

Values cannot be re-assigned:

```
x = 3 // This does not compile.
```

The type of a value can be omitted and inferred, or it can be explicitly stated:

```
val x: Int = 1 + 1
```

Notice how the type declaration `Int` comes after the identifier `x`. You also need a `:`.

Variables

Variables are like values, except you can re-assign them. You can define a variable with the `var` keyword.

```
  var x = 1 + 1
x = 3 // This compiles because "x" is declared with the "var" keyword.
println(x * x) // 9
```

As with values, the type of a variable can be omitted and inferred, or it can be explicitly stated:

```
  var x: Int = 1 + 1
```

Blocks

You can combine expressions by surrounding them with `{}`. We call this a block.

The result of the last expression in the block is the result of the overall block, too:

```
  println({
  val x = 1 + 1
  x + 1
}) // 3
```

Functions

Functions are expressions that have parameters, and take arguments.

You can define an anonymous function (i.e., a function that has 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
```

A function can have multiple parameters:

```
  val add = (x: Int, y: Int) => x + y
println(add(1, 2)) // 3
```

Or it can have no parameters at all:

```
  val getTheAnswer = () => 42
println(getTheAnswer()) // 42
```

Methods

Methods look and behave very similar to functions, but there are a few key differences between them.

Methods are defined with the `def` keyword. `def` is followed by a name, parameter list(s), a return type, and a body:

```
  def add(x: Int, y: Int): Int = x + y
println(add(1, 2)) // 3
```

Notice how the return type `Int` is declared *after* the parameter list and a `:`.

A method can take multiple parameter lists:

```
  def addThenMultiply(x: Int, y: Int)(multiplier: Int): Int = (x + y) * multiplier
println(addThenMultiply(1, 2)(3)) // 9
```

Or no parameter lists at all:

```
  def name: String = System.getProperty("user.name")
println("Hello, " + name + "!")
```

There are some other differences, but for now, you can think of methods as something similar to functions.

Methods can have multi-line expressions as well:

```
  def getSquareString(input: Double): String = {
  val square = input * input
  square.toString
}
println(getSquareString(2.5)) // 6.25
```

The last expression in the body is the method's return value. (Scala does have a `return` keyword, but it is rarely used.)

Classes

You can define classes with the `class` keyword, followed by its name and constructor parameters:

```
  class Greeter(prefix: String, suffix: String) {
  def greet(name: String): Unit =
    println(prefix + name + suffix)
}
```

The return type of the method `greet` is `Unit`, which signifies that there is nothing meaningful to return. It is used similarly to `void` in Java and C. (A difference is that, because every Scala expression must have some value, there is actually a singleton value of type Unit, written (). It carries no information.)

You can make an instance of a class with the `new` keyword:

```
  val greeter = new Greeter("Hello, ", "!")
greeter.greet("Scala developer") // Hello, Scala developer!
```

We will cover classes in depth later.

Case Classes

Scala has a special type of class called a "case" class. By default, instances of case classes are immutable, and they are compared by value (unlike classes, whose instances are compared by reference). This makes them additionally useful for pattern matching.

You can define case classes with the `case class` keywords:

```
  case class Point(x: Int, y: Int)
```

You can instantiate case classes without the `new` keyword:

```
  val point = Point(1, 2)
val anotherPoint = Point(1, 2)
val yetAnotherPoint = Point(2, 2)
```

Instances of case classes are compared by value, not by reference:

```
  if (point == anotherPoint) {
  println(s"$point and $anotherPoint are the same.")
} else {
  println(s"$point and $anotherPoint are different.")
} // Point(1,2) and Point(1,2) are the same.

if (point == yetAnotherPoint) {
  println(s"$point and $yetAnotherPoint are the same.")
} else {
  println(s"$point and $yetAnotherPoint are different.")
} // Point(1,2) and Point(2,2) are different.
```

There is a lot more to case classes that we would like to introduce, and we are convinced you will fall in love with them! We will cover them in depth later.

Objects

Objects are single instances of their own definitions. You can think of them as singletons of their own classes.

You can define objects with the `object` keyword:

```
  object IdFactory {
  private var counter = 0
  def create(): Int = {
    counter += 1
    counter
  }
}
```

You can access an object by referring to its name:

```
  val newId: Int = IdFactory.create()
println(newId) // 1
val newerId: Int = IdFactory.create()
println(newerId) // 2
```

We will cover objects in depth later.

Traits

Traits are abstract data types containing certain fields and methods. In Scala inheritance, a class can only extend one other class, but it can extend multiple traits.

You can define traits with the `trait` keyword:

```
  trait Greeter {
  def greet(name: String): Unit
}
```

Traits can also have default implementations:

```
  trait Greeter {
  def greet(name: String): Unit =
    println("Hello, " + name + "!")
}
```

You can extend traits with the `extends` keyword and override an implementation with the `override` keyword:

```
  class DefaultGreeter extends Greeter

class CustomizableGreeter(prefix: String, postfix: String) extends Greeter {
  override def greet(name: String): Unit = {
    println(prefix + name + postfix)
  }
}

val greeter = new DefaultGreeter()
greeter.greet("Scala developer") // Hello, Scala developer!

val customGreeter = new CustomizableGreeter("How are you, ", "?")
customGreeter.greet("Scala developer") // How are you, Scala developer?
```

Here, `DefaultGreeter` extends only one single trait, but it could extend multiple traits.

We will cover traits in depth later.

Main Method

The main method is the entry point of a Scala program. The Java Virtual Machine requires a main method, named `main`, that takes one argument: an array of strings.

Using an object, you can define the main method as follows:

```
  object Main {
  def main(args: Array[String]): Unit =
    println("Hello, Scala developer!")
}
```

More resources

- Scala book overview

← previous next →

Contributors to this page: