What is Scala

Scala is a statically typed general-purpose programming language, which supports object-oriented programming and functional programming. The code is compiled into .class files, written in Java bytecode, that are run on the Java Virtual Machine (JVM).

Scala Basics

When used as an object-oriented language, like Java, the basic building block of a Scala program are classes, which contain methods. This includes the main method, which is in the entry point of the program. There's a special type of classes in Scala, called objects, which only ever have one instance. This is well suited for the main method, as there is only one instance of it running in the program.

Code is indented in Scala, similar to Python, and unlike Java, semicolons are not used at the end of lines. According to the Scala style guide, indentation in Scala is two spaces, not 4 or tab.

This is how you would implement a Hello World program in Scala:

```
object MainObject {
  def main(args: Array[String]) = {
    println("Hello, world")
  }
}
```

When running the program, the main-method is run. It, in turn, can create other objects and call other methods.

Methods are defined with the def -keyword, and their arguments are written in the order "name: type" such as "x: Int". The return type, in this case Int, is written similarly after the method:

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

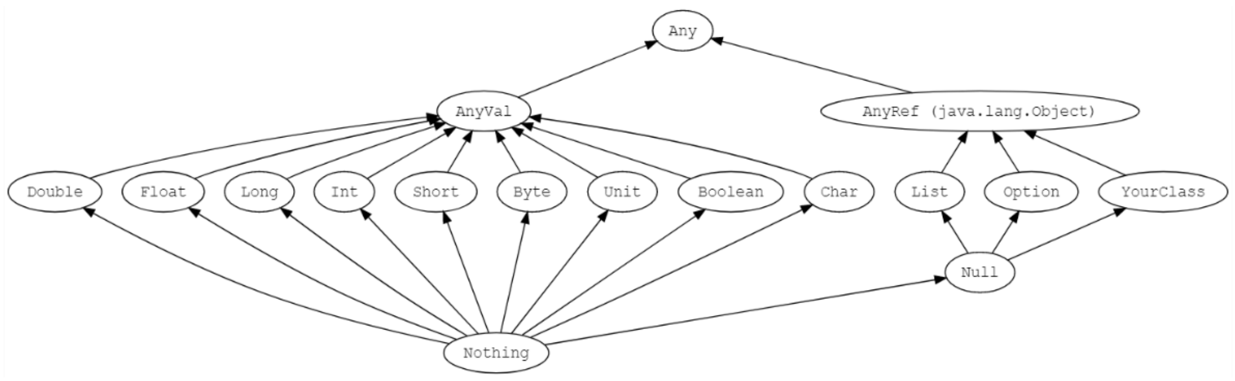
The return value is the last executed line in the function, and return -keyword is not used. If it is stored in a variable, you can type out the variable name. It is good practice to define the type of the return, when defining the method.

Scala has variables and constant values. Variables are mutable, while values are immutable. Variables are declared with keyword var and values with the keyword val. The types can be inferred from the variable, but you can also specify them:

```
var height = 4 // Will be int
val g_earth = 9.81 // Will be double
var greeting: String = "Hello"
```

The type inference also works when leaving out the return type of a method. Specifying the types explicitly is safer, as type inference will fail in some instances, and you may also want to use long instead of int for example.

Being statically typed, Scala enforces the use of its type system with tests at compile-time. This means that you need to explicitly use the correct types, when dealing with variables. Otherwise, your program will not compile, and you are not able to run it. The type hierarchy in Scala is like this:



As you can see, not all types are subtypes of AnyVal, nor AnyRef. Null is not an AnyVal for example, and AnyVal type values can therefore not be assigned null.

As seen in the type system, some classes extend other classes. As an example, you can have Pet, which is a subtype of Animal, and Cat, which is a subtype of pet. You can also have abstract types, when defining a class. For example, designing a container for all classes that are a subtype of Pet would work like this:

```
abstract class Animal {
 def name: String
}

abstract class Pet extends Animal {}

class Cat extends Pet {
  override def name: String = "Cat"
}

class PetContainer[P <: Pet](p: P) {
  def pet: P = p
}
```

You can read more about type bounds in Scala docs: https://docs.scala-lang.org/tour/upper-type-bounds.html, https://docs.scala-lang.org/tour/lower-type-bounds.html

The compiler will stop you from implicitly converting variables from one type to another. This can lead to issues, where you know you have a variable of the correct type, but the compiler does not know that. Here’s an example of that, and how to resolve it:

```
var x: String = "text "
var y: AnyRef = x

var stringArray: Array[String] = Array[String]()


// This will result in an error as y is not a String
stringArray = stringArray :+ y

// You can convert y into a String like this
stringArray = stringArray :+ y.asInstanceOf[String]
```

Scala contains the typical programming control structures, such as loops and if-else structures. An example of their usage is shown here:

```
if (true)
  println("true")
else
  println("false")

for (i <- 0 to 10) {
  println(s"Loop iteration ${i}")
}

while (true) {
  println("This never stops!")
}

var news = "good"

news match {
  case "good" => println("Good news!")
  case "bad" => println("Bad news!")
}
```

These examples follow the style guide’s usage of braces.

In Scala, it is also possible to pass methods as arguments. This makes it possible for a class or a method to call that passed method. Here's an example of that:

```scala
class Caller (callback: Int => Int) {
  def method(x: Int) = {
    callback(x)
  }
}

object MathFunctions {
  def square(x: Int): Int = x*x
  def double(x: Int): Int = 2*x
}

val squarer = new Caller(MathFunctions.square)
val doubler = new Caller(MathFunctions.double)

squarer.method(3) // Returns 9
doubler.method(3) // Returns 6
```

Passing callback methods can be done for multiple reasons. They can be used to give a method limited access to a method of another class it would typically not have access to. They can also be called at the end of the method in an asynchronous program to inform that the method is completed. When given as builder arguments, they can be used to create specialized versions of the same class.

Testing Scala

In addition to the tests conducted by the compiler, it is possible to write your own tests for the code. On the course, you are provided with some scalatest -tests that test functionalities of the class you have written. It is possible to augment those tests with some of your own, to increase the test coverage. The basic model for the tests is the following:

```scala
// First you have the package name. This gives the testing suite
// access to the class you are testing.
package mypackage

// Then you import the testing suite and other libraries you may
// require in your testing.
import org.scalatest.funsuite.AnyFunSuite
import org.scalatest.concurrent.TimeLimitedTests
import org.scalatest.time.{Seconds, Span}

// The tests are run in a test class. Extend the test style you want
// to create, and include style traits you want to enable
class MyTest extends AnyFunSuite with TimeLimitedTests {

  // Inside the class you write your tests. In this case, it includes
  // a time limit of 10 seconds for running the tests.
  override def timeLimit: Span = Span(10, Seconds)

  // Then you define the test environment, including variables
  // you want to test, such as instances of your class
  val testedInstanceOne = new MyClass()
  val testedInstanceTwo = new MyClass()


  // Finally, you define your tests. The exact details depend on which
  // style you use, but the basic model is the same: first you define
  // a test, then you assert what the outcome should be.
  test("Instances of the class should behave similarly") {
    assert(testedInstanceOne.method() == testedInstanceTwo.method())
  }
}
```

There are multiple styles for writing tests, and not all the tests provided with the assignments use the same style. Read more about testing styles in scalatest here:
https://www.scalatest.org/user_guide/selecting_a_style