Introduction to Scala

Programming Paradigms

There are four different broad types or styles of programming called paradigms.

– The imperative programming paradigm is defined by explicit instructions telling the machine what to do and mutable state where values are changed over time.

– The functional paradigm is based on Church's lambda calculus and uses functions in a very mathematical sense. Mathematical functions do not involve mutable state.

– Object-orientation is highlighted by combining data and functionality together into objects.

– Logic programming is extremely declarative, meaning that you say what solution you want, but not how to do it.

– Scala is purely object-oriented with support for both functional and imperative programming styles.

About Scala

High-level language for the JVM

- Object oriented + functional programming

Statically typed

- Comparable in speed to Java*
- Type inference saves us from having to write explicit types most of the time

Interoperates with Java

- Can use any Java class (inherit from, etc.)
- Can be called from Java code

A Scalable Language

The name Scala stands for "scalable language."

The language is so named because it was designed to grow with the demands of its users.

You can apply Scala to a wide range of programming tasks, from writing small scripts to building large systems

Scala is easy to get into. It runs on the standard Java platform and interoperates seamlessly with all Java libraries. It's quite a good language for writing scripts that pull together Java components

What is Scala?

- Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It seamlessly integrates features of object-oriented and functional languages.

Scala is object-oriented

- Scala is a pure object-oriented language in the sense that every value is an object. Types and behaviors of objects are described by classes and traits. Classes can be extended by subclassing, and by using 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. Scala provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and it supports currying.

Scala's case classes and its built-in support for pattern matching provide the functionality of algebraic types, which are used in many functional languages. Singleton objects provide a convenient way to group functions that aren't members of a class.

Scala is extensible

In practice, the development of domain-specific applications often requires domain-specific language extensions. Scala provides a unique combination of language mechanisms that make it straightforward to add new language constructs in the form of libraries.

- In many cases, this can be done without using meta-programming facilities such as macros. For example:
- Implicit classes allow adding extension methods to existing types.
- String interpolation is user-extensible with custom interpolators.

Scala interoperates

Scala is designed to interoperate well with the popular Java Runtime Environment (JRE). In particular, the interaction with the mainstream object-oriented Java programming language is as seamless as possible. Newer Java features like SAMs, lambdas, annotations, and generics have direct analogues in Scala.

Important features of Scala

- It’s a high-level language
- It’s statically typed
- Its syntax is concise but still readable — we call it *expressive*
- It supports the object-oriented programming (OOP) paradigm
- It supports the functional programming (FP) paradigm
- It has a sophisticated type inference system
- Scala code results in *.class* files that run on the Java Virtual Machine (JVM)
- It’s easy to use Java libraries in Scala

Scala Basics - Expressions

Expressions are computable statements. You can output the results of expressions using **`println`**:

```
scala> 1 + 1
res1: Int = 2

scala> println(1)
1

scala> println(1+1)
2

scala> println("Hello!")
Hello!

scala> println("Hello" + " world!")
Hello world!
```

Objects & Methods (Scala Basics)

```
scala> 5.6
res0: Double = 5.6

scala> 5.6.toInt
res1: Int = 5

scala> res0
res2: Double = 5.6

scala> res1.
!=   <=          ceil          isInfinity      isWhole       toBinaryString   toOctalString
%    ==          compare       isNaN           longValue     toByte           toRadians
&    >           compareTo     isNegInfinity   max           toChar           toShort
*    >=          doubleValue   isPosInfinity   min           toDegrees        unary_+
+    >>          floatValue    isValidByte     round         toDouble         unary_-
-    >>>         floor         isValidChar     self          toFloat          unary_~
/    ^           getClass      isValidInt      shortValue    toHexString      underlying
<    abs         intValue      isValidLong     signum        toInt            until
<<   byteValue   isInfinite    isValidShort    to            toLong           |

scala> 4+5
res3: Int = 9

scala> true
res4: Boolean = true

scala> false
res5: Boolean = false

scala>
```

Scala is an object-oriented language in pure form: every value is an object and every operation is a method call.

For example, when you say 1 + 2 in Scala, you are actually invoking a method named + defined in class Int.

You can define methods with operator-like names that clients of your API can then use in operator notation.

Character type

```
scala> 'a'
res6: Char = a

scala> 'a'.
!=    ==          compare       isNegInfinity   min               toDouble         unary_~
%     >           compareTo     isPosInfinity   round             toFloat          underlying
&     >=          doubleValue   isValidByte     self              toHexString      until
*     >>          floatValue    isValidChar     shortValue        toInt            |
+     >>>         floor         isValidInt      signum            toLong
-     ^           getClass      isValidLong     to                toOctalString
/     abs         intValue      isValidShort    toBinaryString    toRadians
<     byteValue   isInfinite    isWhole         toByte            toShort
<<    ceil        isInfinity    longValue       toChar            unary_+
<=    charValue   isNaN         max             toDegrees         unary_-

scala> 'a'.toInt
res7: Int = 97

scala> 'a'+1
res8: Int = 98

scala> 'a'-1
res9: Int = 96

scala> ('a'+1).toChar
res10: Char = b
```

Character and String type

```
scala> 'b'-'a'
res11: Int = 1

scala> "Hello world"
res12: String = Hello world

scala> 'ab'
<console>:1: error: unclosed character literal
'ab'
   ^

scala> "a"+"b"
res13: String = ab

scala> "hi"+" there"
res14: String = hi there

scala> "hi"+5
res15: String = hi5

scala> "hi"+5.2
res16: String = hi5.2

scala> "hi"-5
<console>:12: error: value - is not a member of String
       "hi"-5
           ^
```

Other Integer types

```
scala> Int.MinValue
res0: Int = -2147483648

scala> Int.MaxValue
res1: Int = 2147483647

scala> Int.MaxValue+1
res2: Int = -2147483648

scala> Byte.MinValue
res3: Byte = -128

scala> Byte.MaxValue
res4: Byte = 127

scala> 2000000000 + 2000000000
res5: Int = -294967296

scala> 2000000000L + 2000000000L
res6: Long = 4000000000
```

Package math

- The package object `scala.math` contains methods for performing basic numeric operations such as elementary exponential, logarithmic, root and trigonometric functions.

Floating Point Numbers

```
scala> 1.0 - 0.9 - 0.1
res0: Double = -2.7755575615628914E-17

scala> 1.0f -0.9f -0.1f
res1: Float = 2.2351742E-8

scala> Math.
E                copySign         hypot              nextAfter    sqrt
IEEEremainder    cos              incrementExact     nextDown     subtractExact
PI               cosh             log                nextUp       tan
abs              decrementExact   log10              pow          tanh
acos             exp              log1p              random       toDegrees
addExact         expm1            max                rint         toIntExact
asin             floor            min                round        toRadians
atan             floorDiv         multiplyExact      scalb        ulp
atan2            floorMod         multiplyFull       signum
cbrt             fma              multiplyHigh       sin
ceil             getExponent      negateExact        sinh

scala> Math.PI
res2: Double = 3.141592653589793

scala> Math.random
res3: Double = 0.0014043948853560417
```

Hello world example

```
object Hello extends App {
  println("Hello, world")
}
```

Save to **Hello.scala**

At linux command prompt, compile with scalac as

$ scalac Hello.scala

Observe the class files Hello.class and Hello$.class. The class file can be run in JVM

Run the Hello application with the scala command:

$ scala Hello

The Scala REPL

The Scala REPL ("Read-Evaluate-Print-Loop") is a command-line interpreter that you use as a "playground" area to test your Scala code.

To start a REPL session, just type scala at your operating system command line, and you'll see something like this:

```
$ scala
Welcome to Scala 2.13.0 (Java HotSpot(TM) 64-Bit Server VM, Java 1.
Type in expressions for evaluation. Or try :help.

scala> _
```

Two types of variables

Scala has two types of variables:

- val is an immutable variable — like final in Java — and should be preferred
- var creates a mutable variable, and should only be used when there is a specific reason to use it
- Examples:

```
val x = 1    //immutable; Values cannot be re-assigned.
x = 3  //this does not compile
var y = 0    //mutable
```

Declaring variable types

In Scala, you typically create variables without declaring their type:

```
val x = 1
val s = "a string“
val p = new Person("Regina")
```

When you do this, Scala can usually infer the data type for you, as shown in these REPL examples:

```
scala> val x = 1
       val x: Int = 1

scala> val s = "a string"
       val s: String = a string
```

This feature is known as type inference

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

```
val x: Int = 1
val s: String = "a string"
val p: Person = new Person("Regina")
```

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

As you can see, that code looks unnecessarily verbose.

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
```

You may have noticed that there were no semicolons after variable declarations or assignments. In Scala, semicolons are only required if you have multiple statements on the same line.

You can declare multiple values or variables together:

```
val xmax, ymax = 100 // Sets xmax and ymax to 100
var greeting, message: String = null
// greeting and message are both strings, initialized with null
```

Scala has seven numeric types:

Byte, Char, Short, Int, Long, Float, and Double, and a Boolean type. These types are classes. There is no distinction between primitive types and class types in Scala.

You can invoke methods on numbers, for example:

```
1.toString() // Yields the string "1" or, more interestingly,
1.to(10) // Yields Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
// The Range class can be viewed as a collection of numbers.
```

```
scala> 1.
!=   <=          ceil         isInfinity     isWhole       toBinaryString   toOctalString
%    ==          compare      isNaN          longValue     toByte           toRadians
&    >           compareTo    isNegInfinity  max           toChar           toShort
*    >=          doubleValue  isPosInfinity  min           toDegrees        unary_+
+    >>          floatValue   isValidByte    round         toDouble         unary_-
-    >>>         floor        isValidChar    self          toFloat          unary_~
/    ^           getClass     isValidInt     shortValue    toHexString      underlying
<    abs         intValue     isValidLong    signum        toInt            until
<<   byteValue   isInfinite   isValidShort   to            toLong           |

scala> 1.to(10)
res1: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> 1 to 10
res2: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> "Hello".intersect("World")
res3: String = lo

scala> "Bonjour".sorted
res4: String = Bjnooru
```

math package

Use import packageName._ whenever you need to import a particular package. For invoking mathematical methods import the math package.

```
import scala.math._ // In Scala, the _ character is a "wildcard," like * in Java
sqrt(2) //  Yields 1.4142135623730951
pow(2, 4) //  Yields 16.0
min(3, Pi) //  Yields 3.0
If you don't import the scala.math package, add the package name:
scala.math.sqrt(2)
```

In Scala, it is common to use a syntax that looks like a function call.

For example, if s is a string, then s(i) is the ith character of the string.

```
val s = "Hello"
s(4) // Yields 'o'
```

Scaladoc

Scala programmers can use Scaladoc to navigate the Scala API

You can browse Scaladoc online at **www.scala-lang.org/api**, but it is a good idea to download a copy from

http://scala-lang.org/download/all.html and install it locally.

Unlike Javadoc, which presents an alphabetical listing of classes, Scaladoc is organized by packages. If you know a class or method name, don't bother navigating to the package. Simply use the search bar on the top of the entry page https://www.scala-lang.org/api/current/

Scala Standard Library 2.13.4

Search

root package

Packages

root

scala

`package root`

This is the documentation for the Scala standard library.

Package structure

The scala package contains core types like `Int`, `Float`, `Array` or `Option` which are accessible in all Scala compilation units without explicit qualification or imports.

Notable packages include:

- `scala.collection` and its sub-packages contain Scala's collections framework
  - `scala.collection.immutable` - Immutable, sequential data-structures such as `Vector`, `List`, `Range`, `HashMap` or `HashSet`
  - `scala.collection.mutable` - Mutable, sequential data-structures such as `ArrayBuffer`, `StringBuilder`, `HashMap` or `HashSet`
  - `scala.collection.concurrent` - Mutable, concurrent data-structures such as `TrieMap`
- `scala.concurrent` - Primitives for concurrent programming such as `Futures` and `Promises`
- `scala.io` - Input and output operations
- `scala.math` - Basic math functions and additional numeric types like `BigInt` and `BigDecimal`
- `scala.sys` - Interaction with other processes and the operating system
- `scala.util.matching` - Regular expressions

Other packages exist. See the complete list on the right.

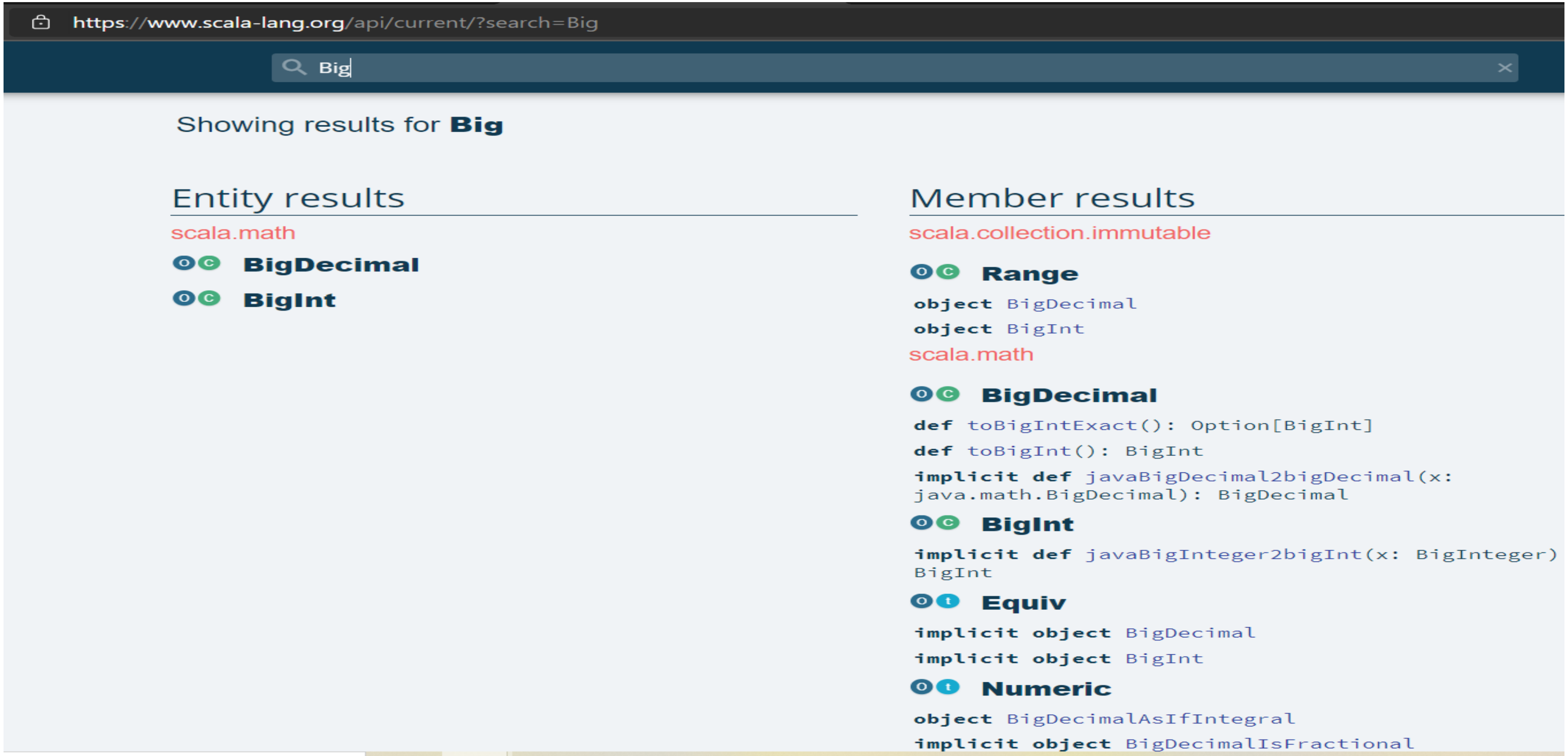
Additional parts of the standard library are shipped as separate libraries. These include:

- `scala.reflect` - Scala's reflection API (scala-reflect.jar)
- `scala.xml` - XML parsing, manipulation, and serialization (scala-xml.jar)
- `scala.collection.parallel` - Parallel collections (scala-parallel-collections.jar)
- `scala.util.parsing` - Parser combinators (scala-parser-combinators.jar)
- `scala.swing` - A convenient wrapper around Java's GUI framework called Swing (scala-swing.jar)

Automatic imports

Identifiers in the scala package and the `scala.Predef` object are always in scope by default.

The search bar in Scaladoc



```
scala> val a = 5
a: Int = 5

scala> var b = 6
b: Int = 6

scala> val s = "hi"
s: String = hi

scala> val x:Double = 0
x: Double = 0.0

scala> val y:Int = 3.6
<console>:11: error: type mismatch;
 found   : Double(3.6)
 required: Int
       val y:Int = 3.6
                   ^

scala> val theAnswer = 42
theAnswer: Int = 42

scala> b = 7
b: Int = 7

scala> a = 8
<console>:12: error: reassignment to val
       a = 8
         ^
```

Tuples

```
scala> (4, "hi", 4.7)
res0: (Int, String, Double) = (4,hi,4.7)

scala> res0.
_1   canEqual   hashCode       productElement    toString
_2   copy       invert         productIterator   x
_3   equals     productArity   productPrefix     zipped

scala> res0._1
res1: Int = 4

scala> res0._2
res2: String = hi

scala> val (age,word,price) = res0
age: Int = 4
word: String = hi
price: Double = 4.7
```

String Interpolation

Scala offers a mechanism to create strings from your data: String Interpolation. String Interpolation allows users to embed variable references directly in processed string literals. Here's an example:

```
val name = "James"
println(s"Hello, $name")  // Hello, James
```

In the above, the literal s"Hello, $name" is a processed string literal.

String Methods

```
scala> "hi there"
res0: String = hi there

scala> res0(0)
res1: Char = h

scala> res0(1)
res2: Char = i

scala> res0.indexOf("h")
res4: Int = 0

scala> res0.lastIndexOf("h")
res5: Int = 4

scala> res0.indexOf("t")
res6: Int = 3

scala> res0.indexOf("th")
res7: Int = 3
```

String Methods

```
scala> res0
res8: String = hi there

scala> res0.substring(3)
res9: String = there

scala> res0.substring(3,6)
res10: String = the

scala> res0.splitAt(2)
res11: (String, String) = (hi," there")

scala> res0.splitAt(3)
res12: (String, String) = ("hi ",there)

scala> res12._1.trim
res13: String = hi

scala> res12._1.trim.length
res14: Int = 2
```

Arithmetic and Operator Overloading

Arithmetic operators in Scala work just as you would expect in Java or C++:

```
val answer = 8 * 5 + 2
```

The + - * / % operators do their usual job, as do the bit operators & | ^ >> <<.

These operators are actually methods. For example,

`a + b` is a shorthand for **`a.+(b)`**

Here, + is the name of the method

In general, you can write **`a method b`** as a shorthand for **`a.method(b)`**

where method is a method with two parameters (one implicit, one explicit).

Conditions, Loops & Functions in Scala

In this section, you will learn how to implement conditions, loops, and functions in Scala. A fundamental difference between Scala and other programming languages is that in Java or C++, we differentiate between expressions (such as 3 + 4) and statements (for example, an if statement). An expression has a value; a statement carries out an action. In Scala, almost all constructs have values. This feature can make programs more concise and easier to read.

Conditional Expressions

Scala has an if/else construct with the same syntax as in Java or C++. However, in Scala, an if/else has a value, namely the value of the expression that follows the if or else.

For example, **`if (x > 0) 1 else -1`** has a value of 1 or -1, depending on the value of x.

You can put that value in a variable:

```
val s = if (x > 0) 1 else -1
```

This has the same effect as

```
if (x > 0) s = 1 else s = -1
```

However, the first form is better because it can be used to initialize a val. In the second form, s needs to be a var.

Conditional Expressions

If the else part is omitted, for example in

`if (x > 0) 1` then it is possible that the if statement yields no value. However, in Scala, every expression is supposed to have some value. This is finessed by introducing a class Unit that has one value, written as ().

The if statement without an else is equivalent to

```
if (x > 0) 1 else ()
```

Think of () as a placeholder for “no useful value,” and of Unit as an analog of void in Java or C++.

(Technically speaking, void has no value whereas Unit has one value that signifies “no value.”)

Statement Termination

In Java and C++, every statement ends with a semicolon. In Scala—like in JavaScript and other scripting languages—a semicolon is never required if it falls just before the end of the line.

A semicolon is also optional before an }, an else, and similar locations where it is clear from context that the end of a statement has been reached.

However, if you want to have more than one statement on a single line, you need to separate them with semicolons. For example,

```
if (n > 0) { r = r * n; n -= 1 }
```

(A semicolon is needed to separate r = r * n and n -= 1. Because of the }, no semicolon is needed after the second statement).

If you want to continue a long statement over two lines, make sure that the first line ends in a symbol that cannot be the end of a statement. An operator is often a good choice:

```
s = s0 + (v - v0) * t + // The + tells the parser that this is not the end
  0.5 * (a - a0) * t * t
```

In practice, long expressions usually involve function or method calls, and then you don’t need to worry much—after an opening (, the compiler won’t infer the end of a statement until it has seen the matching).

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
```

Code Block example

```
import io.StdIn._
{
   println(“Enter a number:”)
   val num = readInt
   num*2
}
```

Block Expressions and Assignments

In Scala, a { } block contains a sequence of expressions, and the result is also an expression. The value of the block is the value of the last expression. This feature can be useful if the initialization of a val takes more than one step.

For example,

`val distance = { val dx = x - x0; val dy = y - y0; sqrt(dx * dx + dy * dy) }`

The value of the { } block is the last expression, shown here in bold. The variables dx and dy, which were only needed as intermediate values in the computation, are neatly hidden from the rest of the program.

Functions

Functions are expressions that have parameters, and take arguments. A function can have multiple parameters:

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

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
```

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.)

Technically, Scala is a blend of object-oriented and functional programming concepts in a statically typed language

Functional Programming - Definition

In computer science, functional programming is a programming paradigm where programs are constructed by applying and composing functions. It is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program.

Source - Wikepaedia

Programs of different sizes tend to require different programming constructs. Consider, the following Scala program:

```scala
var capital = Map("US" -> "Washington", "France" -> "Paris")

capital += ("Japan" -> "Tokyo")

println(capital("France"))
```

This program sets up a map from countries to their capitals, modifies the map by adding a new binding ("Japan" -> "Tokyo"), and prints the capital associated with the country France.

The notation in this example is high level, to the point, and not cluttered with extraneous semicolons or type annotations.

The feel is that of a modern "scripting" language like Perl, Python, or Ruby. One common characteristic of these languages, which is relevant for the example above, is that they each support an "associative map" construct in the syntax of the language.