**Everything is an Object**

Scala is a pure object-oriented language in the sense that *everything* is an object, including numbers or functions. It differs from Java in that respect, since Java distinguishes primitive types (such as boolean and int) from reference types, and does not enable one to manipulate functions as values.

### Numbers are objects

Since numbers are objects, they also have methods. And in fact, an arithmetic expression like the following:

1 + 2 \* 3 / x

consists exclusively of method calls, because it is equivalent to the following expression, as we saw in the previous section:

(1).+(((2).\*(3))./(x))

This also means that +, \*, etc. are valid identifiers in Scala.

### Functions are objects

Perhaps more surprising for the Java programmer, functions are also objects in Scala. It is therefore possible to pass functions as arguments, to store them in variables, and to return them from other functions. This ability to manipulate functions as values is one of the cornerstone of a very interesting programming paradigm called **functional programming**.

## Classes

As we have seen above, Scala is an object-oriented language, and as such it has a concept of class. (For the sake of completeness, it should be noted that some object-oriented languages do not have the concept of class, but Scala is not one of them.) Classes in Scala are declared using a syntax which is close to Java’s syntax. One important difference is that classes in Scala can have parameters. This is illustrated in the following definition of complex numbers.

class Complex(real: Double, imaginary: Double) {

def re() = real

def im() = imaginary

}

This Complex class takes two arguments, which are the real and imaginary part of the complex. These arguments must be passed when creating an instance of class Complex, as follows: new Complex(1.5, 2.3). The class contains two methods, called re and im, which give access to these two parts.

It should be noted that the return type of these two methods is not given explicitly. It will be inferred automatically by the compiler, which looks at the right-hand side of these methods and deduces that both return a value of type Double.

### Methods without arguments

A small problem of the methods re and im is that, in order to call them, one has to put an empty pair of parenthesis after their name, as the following example shows:

object ComplexNumbers {

def main(args: Array[String]) {

val c = new Complex(1.2, 3.4)

println("imaginary part: " + c.im())

}

}

It would be nicer to be able to access the real and imaginary parts like if they were fields, without putting the empty pair of parenthesis. This is perfectly doable in Scala, simply by defining them as methods without arguments. Such methods differ from methods with zero arguments in that they don’t have parenthesis after their name, neither in their definition nor in their use. Our Complex class can be rewritten as follows:

class Complex(real: Double, imaginary: Double) {

def re = real

def im = imaginary

}

### Inheritance and overriding

All classes in Scala inherit from a super-class. When no super-class is specified, as in the Complex example of previous section, scala.AnyRef is implicitly used.

It is possible to override methods inherited from a super-class in Scala. It is however mandatory to explicitly specify that a method overrides another one using the override modifier, in order to avoid accidental overriding. As an example, our Complex class can be augmented with a redefinition of the toString method inherited from Object.

class Complex(real: Double, imaginary: Double) {

def re = real

def im = imaginary

override def toString() =

"" + re + (if (im < 0) "" else "+") + im + "i"

}

## Case Classes and Pattern Matching

{ case "x" => 5 }

This notation defines a function which, when given the string "x" as argument, returns the integer 5, and fails with an exception otherwise.

## Genericity

The last characteristic of Scala we will explore in this tutorial is genericity. Java programmers should be well aware of the problems posed by the lack of genericity in their language, a shortcoming which is addressed in Java 1.5.

Genericity is the ability to write code parametrized by types. For example, a programmer writing a library for linked lists faces the problem of deciding which type to give to the elements of the list. Since this list is meant to be used in many different contexts, it is not possible to decide that the type of the elements has to be, say, Int. This would be completely arbitrary and overly restrictive.

Java programmers resort to using Object, which is the super-type of all objects. This solution is however far from being ideal, since it doesn’t work for basic types (int, long, float, etc.) and it implies that a lot of dynamic type casts have to be inserted by the programmer.

Scala makes it possible to define generic classes (and methods) to solve this problem. Let us examine this with an example of the simplest container class possible: a reference, which can either be empty or point to an object of some type.

class Reference[T] {

private var contents: T = \_

def set(value: T) { contents = value }

def get: T = contents

}

The class Reference is parametrized by a type, called T, which is the type of its element. This type is used in the body of the class as the type of the contents variable, the argument of the set method, and the return type of the get method.

The above code sample introduces variables in Scala, which should not require further explanations. It is however interesting to see that the initial value given to that variable is \_, which represents a default value. This default value is 0 for numeric types, false for the Boolean type, () for the Unit type and null for all object types.