1
Introduction

Scala is a scalable object-oriented programming language with features found in functional programming languages. Nowadays, the object-oriented approach to software construction is considered the most succesful methodology for software design, mainly because it makes software reuse extremely easy. On the other hand, functional programming offers some very elegant tools which when combined with an object-oriented program development philosophy define a really powerful programming methodology. Generally, any programming language that can be extended seamlessly is called scalable. When all these ideas are combined in a single tool, then the result is a particularly powerful programming language.

1.1 Object orientation

The first object-oriented programming language was SIMULA [18], which was designed and implemented by Ole-Johan Dahl and Kristen Nygaard. The SIMUlation LAnguage was designed "to facilitate formal description of the layout and rules of operation of systems with discrete events (changes of state)." In other words, SIMULA was designed as a simulation tool of discrete systems. Roughly, a simulation involves the representation of the functioning of one system or process by means of the functioning of another. In order to achieve its design goal, the designers equipped the language with structures that would make easy the correspondence between a software simulation and the physical system itself. The most important of these structures is the *process*. A process "is intended as an aid for decomposing a discrete event system into components, which are separately describable." Processes, which nowadays are called classes, consist of two parts: a data part and a code part. In the data part, programmers can declare and/or define variables, while in the code part they can define actions (procedures) to process the data. Processes can be combined to describe the functionality of some system. *Elements*, which nowadays are called *objects*, are *instances* of processes, thus, for a single process there may

be different instances. It turns out that these simple ideas are the core of what is now known as object-orientation. Nevertheless, the next major milestone in this technology was the design and implementation of Smalltalk (see [27] and [40] for an elegant and concise presentation of Smalltalk).

Smalltalk is an object-oriented programming language[1] designed and implemented by Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler, Scott Wallace, and several other people working at Xeror PARC during the 1970s. The basic design principle of Smalltalk was the idea that all data manipulated by a program are *objects*, that is, software entities capable of interacting with other similar objects. According to this view, the operation `3+4` is viewed as if the object `3` is sending the message `+` to object `4`. Then, if object `4` understands the message `+`, it starts the execution of a method that specifies how to respond to this particular message. In this case, the method responds by sending back the object `7`.

The paradigm shift pioneered by SIMULA and Smalltalk shaped the whole industry and this is evident in the number of object-oriented languages that emerged and their use in industry. For example, today any software engineer is fluent in at least one of the following object-oriented programming languages: C++ [70], Java [28], Eiffel [52], Self [74], Ruby [22], Python [49], Objective C [16], and Oberon [66]. But what are the reasons for the success of the object-oriented programming paradigm?

The reason for this success is that object-oriented languages implement a number of principles that make the software design and construction process much simpler and elegant when compared to "traditional" approaches. The four basic principles of object-orientation are described briefly below.

Abstraction Objects lie at the heart of object-oriented program design. A software object is an *abstraction* of a real-world object. An object has the essential characteristics of the real-world object that distinguish it from all other kinds of object. Thus, it is important to classify the various characteristics as essential or insignificant. This way the software becomes simpler and easier to understand.

Encapsulation An object is a software component that is characterized by its *state* and its *behavior*. Fields (think of them as placeholders that may hold numbers, words, etc.) are used to store its state, while its behavior depends on the actions its methods may take. Typically, the fields of an object are accessible only through its methods. In other words, one can either change the state of an object or become aware of its current state by invoking specific methods. This implies that the internal state of an object is not visible to anyone, thus providing a data protection mechanism. This property is known as *data encapsulation*.

Inheritance In general, objects are not independent software components. Usually, objects are related with an "isa" relationship, that is, if *A* and *B* are two objects such

[1] In fact, the designers of Smalltalk were the first to introduce the widely used object-oriented parlance that includes terms such as *object-oriented*, *method*, etc.

that *B* extends the functionality of *A*, then we say that *B is a A*. Object *B* may extend the functionality of *A* either by defining new fields and/or methods or by changing the actions taken by some methods. It is customary to say that *B* inherits *A* when *B* is an *A*. Objects may inherit characteristics from more than one object and in this case we talk about multiple inheritance, while if each object may inherit characteristics from only one object, we talk about single inheritance. When building new systems, it is not necessary to design all objects from scratch. Instead, one may opt to use existing objects and extend their functionality to suit one's own needs by designing new objects that inherit existing objects. In a nutshell, this is the essence of software reuse.

Polymorphism Seemingly different real-life structures may actually differ only in the items they process. So instead of defining an object for each instance of the real-life structure (something that is practically not possible), one can design a generic software module and then instantiate it to model particular real-life structures. For example, a stack consists of items that are put one atop the other and one can remove and/or add items only from/to the top of the stack. Thus, if we want a stack of integers or a stack of software modules modeling books, we can create a generic software module that will implement the functionality of any stack and then use particular instances of this software module to simulate stacks of integers and/or books. This marvelous capability is known as polymorphism. To put it very simply, a polymorphic software module is one that may have different instances with identical behavior.

Without worrying about the details, let us see by means of an example how these principles are realized in the language that is presented in this book.

Assume that we want to build a system simulating a zoo. In order to achieve this goal we need to build a hierarchy of classes that will describe the species living in the zoo. Naturally, we do not need to build a different class for each species since, for example, a bee is an insect and all insects are arthropods. Let us start by defining a class that describes arthropods:

```
class arthropod (NumberOfEyes: Int, NumberOfFeet : Int) {
  def numberOfFeet () = println(NumberOfFeet)
  def numberOfEyes () = println(NumberOfEyes)
}
```

We are not interested in every aspect of what makes an animal an arthropod. Instead, we center upon two quite important things: the number of eyes and the number of feet. Obviously, our choice is *subjective*, but it depends on the task we are trying to accomplish and this is exactly the essence of abstraction. Note that the values stored in the fields `NumberOfEyes` and `NumberOfFeet` cannot be changed.

An ant has six legs and let us assume it has two eyes. The declaration that follows creates an `ant` object that corresponds to an ant:

```
val ant = new arthropod(2,6)
```

Although we cannot alter the number of feet or the number of eyes of an `ant` object, we can inspect these values. Indeed, the commands

```
ant.numberOfFeet()
ant.numberOfEyes()
```

print the number of feet and eyes of an ant, correspondingly. Although we have used an indirect way to access the values of each field, one can use the fields directly to access or modify the corresponding values, However, one can also declare the fields in such a way that such operations are not directly possible, and this is a simple example of data encapsulation.

An insect is an arthropod with six feet. Instead of defining a new class for insects from scratch, we can *extend* the functionality of class `arthropod` to define a class for insects:

```
class insect (NumberOfEyes: Int)
   extends arthropod (NumberOfEyes, 6){ }
```

Creating and using `insect` is easy. The commands that follow

```
val bee = new insect(4)
bee.numberOfFeet()
bee.numberOfEyes()
```

create a new `insect` object (stored in variable `bee`) and print the numbers of feet and eyes of a bee. In this particular case, the numbers six and four will be printed on the computer screen. This very simple code shows the essence of inheritance. We extend the functionality of existing software modules by creating new software modules that inherit the properties of these existing modules and add new features making the resulting module more expressive. Although the examples presented are very simple, nevertheless, any real-world application uses inheritance in exactly the same way. The important benefit of the introduction of inheritance is that software modules become reusable. Thus, there is no need to invent the wheel every time one tries to solve a particular problem. And when a programming language is equipped with a huge library of such software modules, then it attracts many users. After all, this is just one of the reasons that the Java programming language has become so popular.

Although there are animals that change their forms entirely during their lifetime (think of butterflies for example), still it makes no sense merely to demonstrate polymorphism using such a complex example. Instead, we will use stacks to demonstrate

polymorphism. As noted already, a stack is a structure where one can add/remove elements from its top. Let us first define a class that simulates a stack of integers:

```
class IntStack (n: Int) {
   private var S = new Array[Int](n)
   private var top = 0;
   private var TopElement;
   def push(elem: Int) {
     top = top + 1
     S(top) = elem
   }
  def pop () : Q = {
    var oldtop = top
    top = top - 1
    S(oldtop)
  }
}
```

Note that we have intentionally left out various checks that should be performed (for example, we cannot pop something from an empty stack) just to keep things simple. Creating new stacks is easy. We just specify the height of the stack as shown below:

```
var x = new IntStack(3)
x.push(3)
x.push(4)
println(x.pop())
```

The last command will print the number 4 on the computer screen. Suppose that we also need a stack of strings. The most "natural" thing to do is to define a `StringStack` by replacing all but the first occurrence of `Int` with `String`. Here the words `Int` and `String` are *data types* or just types. Roughly, a type is defined by prescribing how its elements are formed as well as when two elements are equal (see [64] for a practical account of type theory and [36] and the references therein for an account more suitable for theoretical computer scientists). With types one can distinguish between one as a natural number and one as a real number. In the simplest case, types may be seen as sets of data values. Thus, when one says $x : \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers, one means that x can assume any value that is an integer number. Note that `Int` and `String` denote (a system dependent range of) integer numbers and finite character sequences, respectively. After this brief but necessary explanation, let us continue with our example. If one wants yet another stack structure, it can be defined in a similar way. Nevertheless, a far more elegant

solution would be to define a parametric structure in which the type of its elements would be specified when a new instance of the structure is declared. Consider the following generic definition:

```
class Stack [γ] (n: Int) {
  private var S = new Array[γ](n)
  private var top = 0;
  def push(elem: γ) {
    top = top + 1
    S(top) = elem
   }
  def pop () : γ = {
    var oldtop = top
    top = top - 1
    S(oldtop)
  }
}
```

Here γ is a type variable, in other words, a variable whose values can be any type. This means that types are treated as values of the type of all types, usually called *Type*, and `Stack[γ]` is a generic type, that is, roughly a type pattern that can be used to specify particular types and, therefore, define particular objects of these particular types. In order to create a stack of integers, we need to declare an identifier to be an instance of `Stack[Int]`. In other words, by replacing γ with the name of a specific type (for example `Int`), we create a stack with elements of this particular type. Let us give some concrete examples:

```
var x = new Stack[Int](3)
x.push(3)
x.push(4)
println(x.pop())
var y = new Stack[String](4)
y.push("C++")
y.push("Java")
println(y.pop())
```

The really great benefit of polymorphism is that programmers do not have to spend time and energy defining similar things. On the other hand, finding the similarities between seemingly different structures is another problem that depends on the mathematical maturity of each person.

1.2 An overview of functional programming

A function can be viewed as a black box that maps elements drawn from a set (i.e., a collection of similar objects), which is called the *domain* of the function, into elements drawn from another set known as the *codomain* of the function. However, there is one restriction: no domain element can be mapped simultaneously to two or more different codomain elements. Let us consider a simple function that maps any integer number to an element of a set that consists of the words *minus*, *zero*, and *plus*. Obviously, the domain of the function is $\mathbb{Z}$ and its codomain is the three-word set {plus, zero, minus}. Function sign will map all negative integers to minus and all positive integers to plus. Finally, it will map 0 to zero. Verbal descriptions are not precise enough, so we need a more formal method to describe functions. One simple method is to write down a set of equations that specify which domain element is mapped to which codomain element. For example, the following equations can be considered to define function sign:

$$
\begin{aligned}
&\vdots \\
\text{sign}(-3) &= \text{minus} \\
\text{sign}(-2) &= \text{minus} \\
\text{sign}(-1) &= \text{minus} \\
\text{sign}(0) &= \text{zero} \\
\text{sign}(1) &= \text{plus} \\
\text{sign}(2) &= \text{plus} \\
\text{sign}(3) &= \text{plus} \\
&\vdots
\end{aligned}
$$

Another method to describe a function is to specify a single rule:

$$
\text{sign}(x) = \begin{cases} \text{minus} & \text{if } x < 0 \\ \text{zero} & \text{if } x = 0 \\ \text{plus} & \text{if } x > 0. \end{cases}
$$

The second definition can be easily coded into a Scala function:

```
def sign(x: Int) = if (x > 0)
                     "plus"
                   else if (x == 0)
```

```
  "zero"
else
  "minus"
```

Using the function is straightforward:

```
println(sign(4))
println(sign(-4))
```

Let us consider one more function. Assume that we want to define a function that computes the maximum of its two arguments. Clearly, if the first argument is greater than the second, then the first argument is the maximum. Otherwise, the second argument is the maximum. This function can be easily encoded as a Scala function as shown below:[2]

```
def max(x: Int, y: Int) = if (x > y) x else y
```

Let us make our life a little bit more difficult and let us try to define a function that finds the maximum of three numbers. In order to solve this problem we need to check the various cases – if the first argument is greater than the second and the second is greater than the third, then the first argument is the greatest of all three, etc. Although this computes what we want, it does it in a very complicated way. A simpler approach is to compute the maximum of the second and the third argument and then the maximum of the first argument and the maximum of the second and the third argument, or in Scala

```
def max3(x : Int, y : Int, z : Int) = max(x,max(y,z))
```

This is a form of function *composition*, that is, a process by means of which one can generate a new function from two or more other functions. In addition, *functional programming* can be defined as a programming discipline where programs are usually composite functions.[3] And this is the reason why functional programming

[2] This function is predefined in Scala, but we use it to demonstrate the notion of function composition.
[3] In a sense, this is similar to the *divide and conquer* programming methodology, that is, the decomposition of a particular problem to two or more simpler problems and the subsequent decomposition of these problems until we have problems that are simple enough to be solved directly. Then the composition of these solutions gives a solution to the original problem.

is particularly elegant. In order to ensure that procedure functions can be composed as their mathematical counterparts, one must avoid the so-called *side effects*. To understand what we mean by side effects, consider the following code:

```
var flag = true // a switch: can be either true or false
def f(n : Int) = {
  var k = 0 // local variable
  if (flag) k=n else k=2*n
  flag = ! flag // destructive assignment!
  k  // what the function yields
}
println(f(1) + f(2))
println(f(2) + f(1))
```

This code will print the numbers 5 and 4 on the computer screen. If `f` was a pure function, then the two commands would print exactly the same. The problem with this code is the *destructive assignment*, that is, a command that modifies the value of a variable. Programming languages that allow the use of such assignments are called *referentially opaque*. On the other hand, languages that do not permit the use of destructive assignments are called *referentially transparent*. In general, languages that are referentially transparent are purely functional languages like Haskell [38] and Erlang [5]. Obviously, one can keep side effects out of the programs in a referentially opaque language by deliberately avoiding the use of destructive assignments. Nevertheless, functional programming languages provide a number of tools (for example, pattern matching, algebraic types, that is, the disjoint union of several types) that greatly facilitate programming in these languages. But these are not the fundamental differences between an *imperative* language (i.e., nonfunctional for our purposes) and a functional programming language. The fundamental difference lies in the way solutions to problems are expressed. Typically, an imperative program is a sequence of "imperatives which describe *how* the computer must solve a problem in terms of state changes (updates to assignable variables)" while "a functional program describes *what* is to be computed, that is the program is just an expression, defined in terms of the predefined and user-defined functions, the value of which constitues the result of the program" [21].

1.3 Extendable languages

In October 1998, at the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Guy L. Steele Jr. advocated that "[A] language

design can no longer be a thing. It must be a pattern – a pattern for growth – a pattern for growing the pattern for defining the patterns that programmers can use for their real work and their main goal" [69]. According to Steele there are two kinds of *growth* in a language – one should be able to change either the vocabulary or the rules that say what a sequence of words means (i.e., the *semantics* of a sequence of words). The essence of these two kinds of growth is that one should be able either to define new keywords or to change the meaning of operators and/or keywords. Similarly, there two ways by which a language can grow – either this can be done by a person, or a small group of persons (for example, a committee), or by a whole community. In the second case, members of the user community can actively participate in the extention of a language. Nevertheless, the development cannot be anarchical. For this reason a person or a small group of persons act as project coordinators. But how can a language be designed to be extendable?

Steele argues that the best way to make a language extendable is to include generic types, operator redefinition, and user-defined types of light weight, which could be used to define numeric and related types. In Section 1.1 we have discussed generic types, but we have said nothing about operator overloading and light weight user-defined types.

Instead of providing different predefined types for different kinds of numbers (for example, complex numbers, fractions, etc.), it is far better to provide an infrastructure by means of which one can easily implement such types. Many engineers need to be able to manipulate complex numbers easily, thus, the availability of a numeric type providing the functionality of complex numbers is a key factor in their choice of programming language. Defining a light weight user-defined type where ordinary arithmetic operators are redefined while their original meaning is not lost solves this problem, see Figure 1.1. Here a complex number is simulated by a class with two fields that can assume as values real numbers of double precision. Also, we (re)define the meaning of the operators `+`, `-`, `*`, and `/`. This way, we can write things like the following:

```
var a = new Complex (1.0, 3.0)
var b = new Complex (4.5, -2.5)
println ("a + b = " + (a + b) )
```

The last command will print "`a + b = 5.5+0.5i`" on the computer screen. Note also that in the last command the first `+` is used to concatenate character sequences and the second to add complex variables. And this is the reason we need the extra parentheses, or else we will get the following "erroneous" output

```
a + b = 1.0+3.0i4.5-2.5i
```