

**Scala application using AKKA**

***Marta Doberschuetz***

**T00171021**

# Introduction

1.1. Software chasing hardware

Hardware has been developing very quickly in the past few years, in fact so quickly that software has not been able to keep up with it. The most popular programming languages like Java or C# cannot use multicore processors to the fullest of its abilities. That’s why it has become very important to create languages that would be able to cope with modern hardware.

One of those languages is Scala, created by Martin Odersky, a German computer scientist.

1.2. Programming paradigms

A paradigm, in science, is defined as a set of concepts or thought patterns in a given discipline. (ref. Odersky – 1st course). In computer science a programming paradigm is a way of creating the structure and components of computer applications. Computer scientists distinguish between several programming paradigms, the main ones are:

* imperative
* functional
* object-oriented
* logic
* symbolic.

Some, like Martin Odersky, the creator of Scala, name only the first 3 as main programming paradigms and claim for the object-oriented to be orthogonal to the 3 paradigms and one that combines the features of the 3 paradigms.

Imperative programming uses statements for computation. Statements change the state of the program. In imperative programming, the mutable variables are modified, assignments are used, and so are control structures such as if-then-else, loops, break, continue, return. Programs written using imperative programming paradigm specify a list of tasks that the computer is to perform. Imperative programming indicates how the program should achieve the end result (in what sequence the tasks should be executed) and not what the program should accomplish, like in declarative programming, which is the opposite to imperative programming. Functional and logic programming are examples of declarative programming.

Functional programming uses functions to perform calculations. It does not use mutable variables, assignments, loops and other imperative control structures. In functional programming functions can act like values and those values can be produced, consumed and composed. Additionally functions can be defined in other functions, they can be passed as parameters and returned as results.

Logic programming uses mathematical logic to create programs. Programs written using programming paradigm consist of sequences of logical statements that present facts and rules of the domain problem and an inference algorithm.

Object-oriented programming uses objects to describe the current world state. Objects are described by data fields and the behaviour of the objects is described by methods. Objects then, as instances of classes, interact with one another to create applications and programs.

# Scala

3.1. Scalability

The name “Scala” stands for “scalability”. The term scalability means for the network, process or system to be able to manage when the demands of the users grow and effectively grow with those demands. (Book).

Scala runs on a Java platform and uses Java libraries. Scala combines functional and object-oriented concepts. The combination of the two programming styles also makes the code more concise. (Book). Let’s look at the two code snippets to visualise it:

//enter code here

3.2. Functional and object-oriented languages

In Scala, the value of a function is an object! (Book, page 45). But first things first.

In object-oriented programming, objects structure programs. They are containers for both data and operations performed on that data. Object are also values themselves and can be stored in other object or passed as parameters in operations. (book, page 45). Although object-oriented languages are based on objects, they also have such thing as primitive data types (like in Java), that are not objects and are used to store values, or some methods do not belong to any objects. All of that seems to limit scalability. Whereas in Scala there is no such limitation as everything is an object: every value, every operation. (Book, page 46). For example, if you perform a calculation like: 1+2 in Scala, you actually invoke a method called: + from class Int. (Book, page 46).

In functional programming, functions are first-class values, which means they have the same status as primitive data types or String objects. Functions can be passed as arguments to other functions, functions can be returned as a result or stored in variables. A function can be defined inside another function same way an integer variable can be defined inside a function. (Book, page 47). “Functions that are first-class values provide a convenient means for abstracting over operations and creating new control structures. This generalization of functions provides great expressiveness, which often leads to very legible and concise programs. It also plays an important role for scalability.” (Book, page 47).

“Immutable data structures are one of cornerstones of functional programming.” (Book, page 48). In Java strings are objects of a class String, they are immutable, which means that when you create an object String and then you want to change that object, what you in fact do, is you create another object. If we just look at strings in Java we can say that Java is a functional language. In this case Scala is like Java – it treats strings in a mathematical sense and not as an array of characters – Scala does not allow mutability. Scala also introduces immutable lists, tuples, maps and sets.

Another characteristic of a functional language is the fact that its methods should have no side effects. If a function or expression has a side effect it means that not only does it return a value, it also modifies state or it interacts with the outside world. Functional languages encourage methods with no side effect, Scala gives programmers a choice: they can either write in an imperative style (with mutable data and side effects) or they can avoid it as Scala makes it easy to do so.

Methods with no side effects are called: referentially transparent, which means that a method call can be replaced by its result and the program’s semantics will not be affected. (Book, page 48). In fact, if an expression in Scala has no side effects it can be evaluated using a method called a “substitution model”, which idea is to reduce an expression to a value.

//add from the course here about the evaluate by value and evaluate by name

3.3. Scala – Java’s cousin

“Scala programs compile to JVM bytecodes.” (Book, page 48). When writing programs in Scala one can avail of the richness that Java provides: libraries, methods, classes, fields, interfaces. In fact, Scala has “borrowed” a lot of the Java types: an integer in Java is an Integer in Scala, and on top of that, Scala has introduced some additional methods connected to those types. For example, there is a method called: toInt in Scala that parses a string to an integer. So instead of:

String someNumber = "12";

**int** number = Integer.*parseInt*(someNumber);

is what we would have written in Java, in Scala we simply write:

**def** someNumber = "12"

someNumber.toInt.

Another advantage to Scala programming language is that its code doesn’t take up as many lines. On average Scala program contains half the amount of lines of what the same Java program would. And it is not just about less typing, which means less reading and trying to understand code, but also less errors. As an example the code in Java would look like that:

**public** **class** MyClass {

**private** **int** index;

**private** String name;

**public** MyClass(**int** index, String name){

**this**.index = index;

**this**.name = name;

}

}

(code from the Book, page 50),

Whereas the code in Scala would look like that:

**class** MyClass(index: Integer, name: String) {

}

The code in Scala is obviously more concise, therefore easier to read and understand. Also less space for potential errors.

Semicolons are optional in Scala. Many experienced Java programmers would say they would prefer to know exactly where the statement ends. The rules for semicolon inference are actually simple. There is no need to put a semicolon at the end of the line. The end of the line is the end of a statement. If the statement consists of several lines, you can either wrap them in braces:

(x

+ )

or leave the + sign at the end of the line and the compiler will know the next line also belongs to the same statement:

x +

y.

Scala is statistically typed. A computer program consists of variables, expressions, functions that are of a certain type. There are interfaces defined between different parts of the computer program and the main idea behind a types system is to eliminate bugs. The parts of the computer program are connected and the compiler can check if those parts are connected correctly at either compile time or run time. When the checking for bugs is done at compile time we say it happens statically, when it happens at run time we say it happens dynamically. (Wikipedia).

We talk about compile time when we are in the process of building our program and we talk about run time when we actually execute our program. So in static type system the compiler will inform us of an error just when we are building a program, so since Java is statically typed, when we enter something like this:

String s = 123;

The error message will read: Type mismatch, cannot convert from int to String.

In dynamic programs, the programmer does not indicate type, like in Javascript:

var s = 123

s = "abc".

And when we perform the following calculation:

var x = s \* 10

an error will occur at run time, not at compile time.

Scala is statically typed, like Java, but Scala has a system of type that are “inferred”, so Scala has the ability to “guess” what type the variable is without the programmer having to indicate the type. So when we type:

**val** s = "abc"

Scala will figure out that s is in fact a String.

Of course it is correct to type in:

**val** s:String = "abc",

it may even increase readability or when it comes to numbers, sometimes we define a number that is of floating type, although we don’t actually give any floating point.

In the case below an error will occur:

**val** z:Int = "abc".

So because of Scala has this inferred type it behaves a bit like a dynamic language and dynamic typed system enthusiasts claim the speed of writing code to be one of its big advantages. So there you go – Scala is statically typed programming language with dynamic elements.

Scala differentiate between two types of variables: val and var. A val is similar to final in Java:

**val** s:String = "abc".

Once initialized it cannot be reassigned:

,

the error reads:



and the code will not compile,

whereas var is like a non-final variable in Java and can be reassigned:

.

Types in Scala can be inferred. It means that you don’t always have to specify what type the variable is and the compiler will infer it, guess it. The following statement:

**var** bla:String = "bla"

is the same as:

**var** bla = "bla".

Functions in Scala have the following structure:

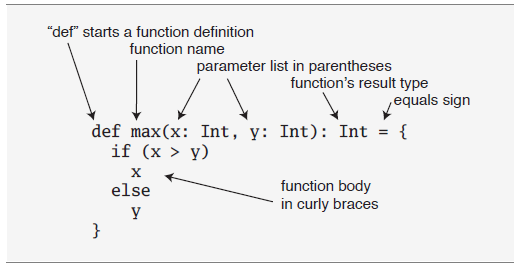
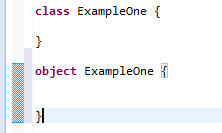


Figure 1.1 Function structure in Scala.

*Source: Odersky, Spoon, Venners, 2010*

The return type is optional and if not indicated a function will return the last value computed by the compiler. The only place where the type has to be specified is the parameters. It is also worth mentioning that the default access modifier in Scala is public.

Scala cannot have static members. Scala has singleton objects instead of static members. The object definition looks the same as class definition:

.

If the name of the object has the same name as the class, it is called a companion object. They both have to be defined in the same source file. The advantage here is that companion object and its companion class can access their private members. “If you are a Java programmer, one way to think of singleton objects is as the home for any static methods you might have written in Java.” (Odersky M., Spoon L., Venners B., 2010, p.67). Those static methods can be invoked the similar way we invoke them in Java – using the name of the object followed by the dot and the name of the method. Singleton objects cannot be instantiated, therefore they cannot take parameters. Companion classes can take parameters.

If an object does not have the same name as the class it is called a standalone object. Standalone object are used to ………………………….

//add here

To run a Scala application a standalone object has to have a main method, the same way to run a Java application you have to a main method defined. The signature of a main method is the following:

**object** Summer {

**def** main(args: Array[String]): Unit = {

}

}.

There is another way of running a singleton object. You can use the Application trait. What you do is you simply put: extends Application, after the name of the object and you don’t have to type the main method definition after that:

**object** Summer **extends** Application{

}

and then you put the body of the application as normal. Saves typing.

In Java the file name has to be exactly the same as the class name. In Scala it doesn’t.

In Java we can define a class that has a method and then we can write another class that extends the first class and therefore inherits all the methods of that first class. In Scala we have traits. A trait can be “mixed in” to a class. We say that we “mix in” a trait, we do not inherit it.

//page 218 in the book

# Deploying a Scala application to Heroku

# Actor paradigm

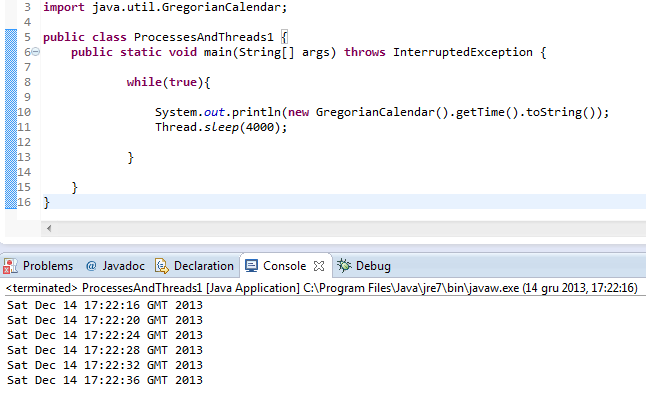
5.1. Multithreaded applications in Java

When we work on our computer we take for granted that our computer runs a few programs at the same time: we can type in a document while our computer is downloading a movie from the Internet or we watch a clip on youtube and at the same time our computer has to make sure that the time on the clock on the taskbar changes accordingly every minute. We talk about concurrency when multiple threads or processes are being executed independently, in parallel.

Java provides a support for multithreaded applications.

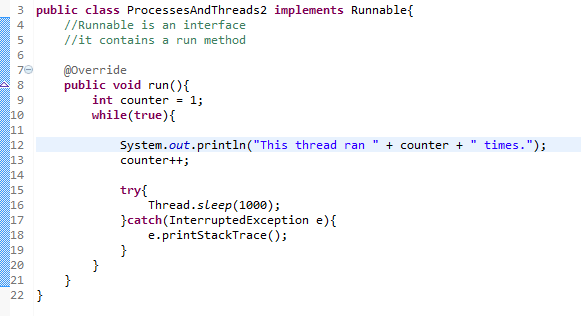
A **process** is “an instance of a computer program that is being executed”. (Wikipedia) A process starts when the program is initiated. For example, when you click on the Eclipse icon, a process that runs Eclipse starts. A program, such as Eclipse, contains several instructions, so the process executes those instructions, but when a single instruction is being executed we refer to it as a **thread**. So a thread is “an execution of the smallest sequence of programmed instructions”. One process can contain multiple threads. The execution of those threads is managed by an operating system scheduler. When multiple threads are contained in one process they share resources, processes do not share resources (resources like memory). Threads also share the code of the process they are in. They also share the process’s values that the process’s variables reference. **Multithreading** is a model in computer science that allows for multiple threads within one process to be executed independently at the same time. The concept is crucial in modern applications. Imagine an application that has to wait to finish one calculation before performing another one, it would appear to the user as frozen. Gamers, be thankful!

Single-threaded Java example:

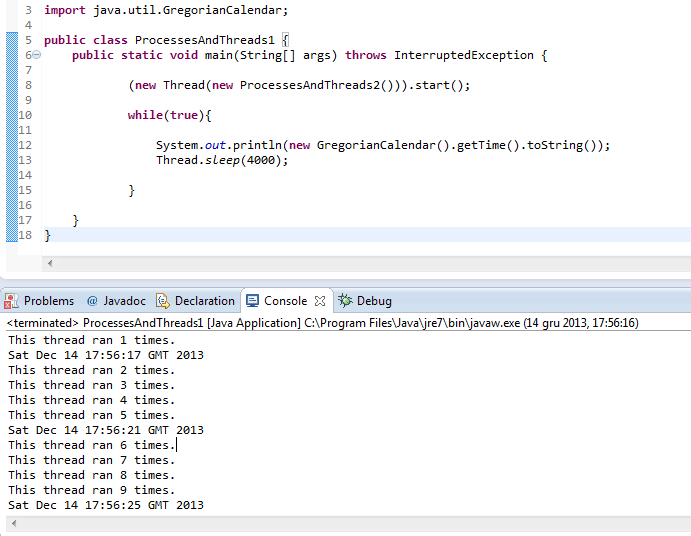


The above application executes an infinite while loop. The body of the loop consists of a print statement that prints date and time, but since the thread is stalled, it prints the date and time every 4 seconds. It is a single threaded application as only one instruction is being executed at a time.

Two-threaded Java application example:



We have created another class called ProcessesAndThreads2 and we created another thread within that class. In order to do that we had to implement an interface called Runnable, which contains a method called run. That method is called whenever we run that thread.



In the above screenshot we can see that we added a call to class ProcessesAndThreads2 in order to be able to access the thread in that class and then we start the thread with start() method. The thread from the ProcessesAndThreads2 class is called a background thread and the thread in the main class is called the foreground thread. Note that the background thread sleeps every 1000 milliseconds so 1 second. Thanks to the while loop created in class ProcessesAndThreads2 we can see with what frequency the threads are being executed.

//book page 691

//shared data and locks

5.2. Multithreaded applications in Scala

Scala has gone a step further. Scala uses actors

# Akka

# Spray

# Application

# References:

* <http://www.drdobbs.com/architecture-and-design/interview-with-scalas-martin-odersky/231001802>
* Wikipedia