Issue August 2010 | presented by jaxenter

www.jaxenter.com

jaxmag

Java, Enterprise Architecture, Agile and Eclipse

Scala is a programming language for the JVM, that integrates both features from object-orientated and functional programming languages. The bytecode of Scala programs is fully compatible with Java, and developers can call Scala from Java, and Java from Scala.

In July, 2010, the Scala community celebrated another milestone with the release of Scala 2.8, which featured a completely redesigned collection library and brand new, specialised type parameters – among a list of other updates and improvements.

#2

Scala

The Joy of Scala

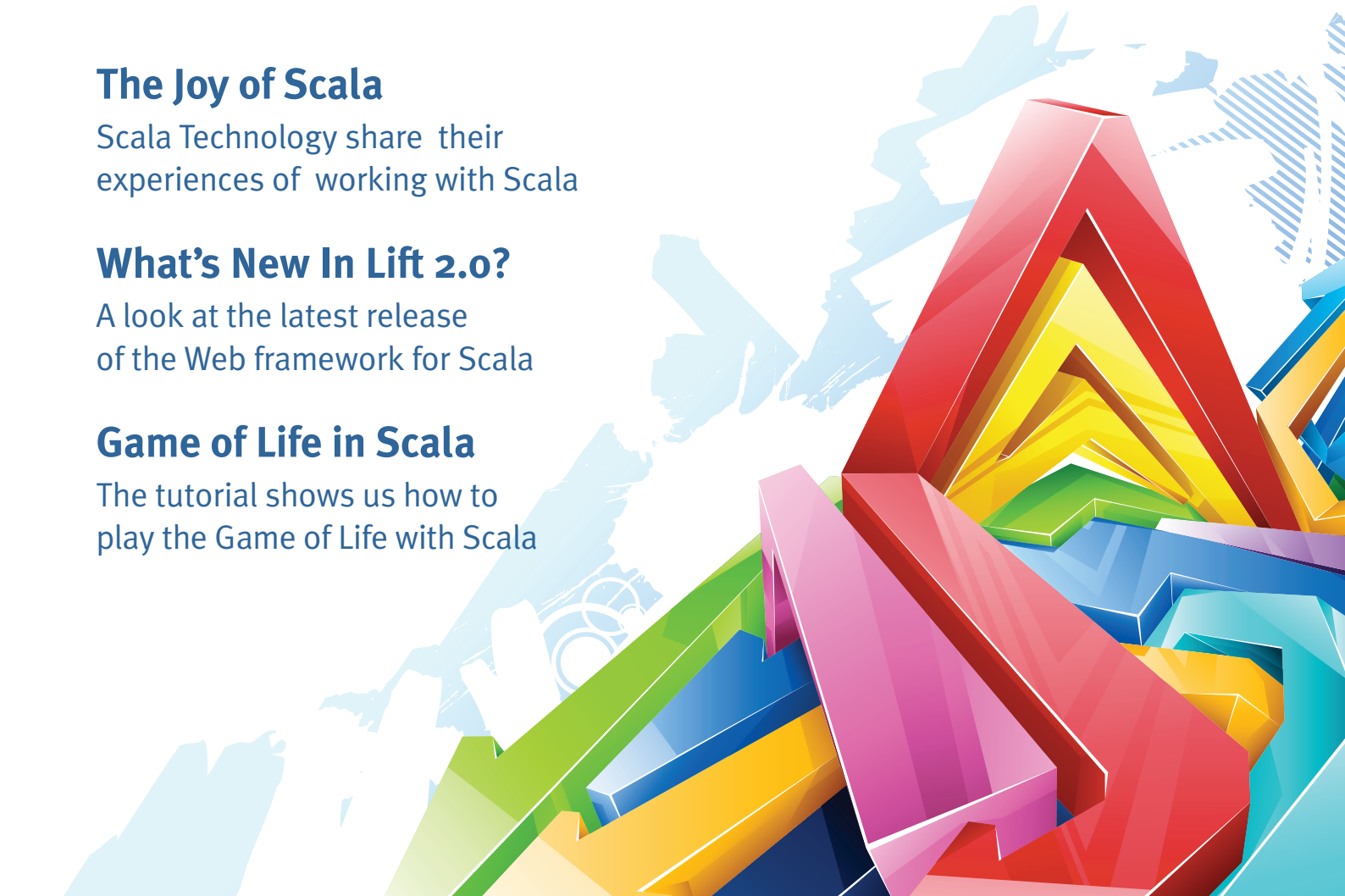
Scala Technology share their experiences of working with Scala

What's New In Lift 2.0?

A look at the latest release of the Web framework for Scala

Game of Life in Scala

The tutorial shows us how to play the Game of Life with Scala



Scala Changes the Java World

When Java set out to conquer the world, many were impressed by its clarity, simplicity and elegance. Derived from C++, Java offered a range of modern and simplified elements, but was much cooler than, say, Delphi or Visual Basic (which at the time were the most popular languages for rapid and easy programming.)

Polyglot Programming

Java was originally considered as a single-language platform or, as most people still view it today, as a programming language. But, it rapidly evolved into a multi language platform, making polyglot programming possible. However, different languages on the Java platform are nothing new: languages such as Beanshell, the expression language for JSF, and many more have been around for more than a decade. But, they were only ever niche languages, and never threatened Java's dominance within the Java ecosystem.

Then, came Ruby on Rails and shortly after that, JRuby, the Ruby implementation for the Java Virtual Machine. Ever since then, developers have been able to use Ruby inside the Java ecosystem. Shortly afterwards, Groovy came along with very similar concepts, but tailored for the Java developers who had no desire to learn something so fundamentally different.

All this development has turned the single-language platform, into a multi language platform. The time is ripe for other languages to appear. Now, Scala – in development since 2003 – is about to conquer the Java world!

Welcome Scala!

Martin Odersky, the creator of Scala, is one of many developers who were fascinated by Java, as it appeared in the mid 90's. What fascinated him, was that it was the first programming language with a sophisticated garbage collection (be sure to check out our video of Martin's Keynote during our conference in Munich, Germany, for more info on this topic!) A few years later, Martin and his team supported Sun Microsystems in developing Generics for Java but, after that, they dived into creating their own language project. Although it was not related to Java the language, Martin and his team remained committed to the JVM: Scala was standards compliant and fully supported the JVM.

Today, we see a robust language that follows many aspects of the functional paradigm, but does not disregard object-orientated principles.

Scala code has captured the community's imagination, with its compact syntax and ability to define internal DSLs, which describe a project or domain-specific task through specific language extensions.

This issue is all about that very programming language!

This issue of JAXmag is dedicated to this exciting programming language:

Scala! Scala has the potential to substantially alter the Java world, while supporting all the standards of the Java platform. Scala requires developers to adopt a new mindset, but in return promises increased productivity.

Read on, for a tutorial taking you through creating your very first serious Scala component, and discover whether or not Scala is right for you. Take a look at the Scala web framework, Lift. You may find yourself telling your boss you're sure you'll have more fun with Scala – and will be more productive, to boot! ;)

Read on, and see if JAXmag's Scala experts can't persuade you to come over to the other side!

Yours,
Sebastian Meyen
Editor in Chief, JAXenter.com & JAXmag

Jessica Thornsby

Sebastian Meyen

Claudia Fröhling

Hartmut Schlosser

Editorial Team JAXmag

PS: If you wish to learn more about Scala, we recommend checking out our full day tutorial with Ted Neward, at JAX London in late September (www.jaxlondon.com for more details!

Experiences of working with Scala

The Joy of Scala

People embark on their Scala journeys in different ways. Maybe it's a personal interest in finding out the best way of achieving something; an interest spawned by a conversation or an article on the Internet; a language that your boss heard some cool stuff about. Our interest started from a feeling that the world of programming languages had much more to give than what we had seen so far.

Jon Pretty, Peter O'Sullivan, Vince Kenealy

Picture a sunny morning in 2004: a light and airy office in a village in rural England where a group of people are seeking the ideal development language. An innate and intuitive distrust of proprietary options; a less visceral - but nonetheless real - dislike of PHP; a positive feeling for some aspects of Java but a sense it could be cleaner, less cumbersome and altogether "better"; an evaluation of the lesser-known and more exotic languages; and an evolving admiration for, and growing recognition that there was one: expressive, fast, safe, maintainable, scalable. It had to be Scala!

Looking back this was an ambitious but risky step. In those days Scala was not the stable, tried and tested solution to our problems it is now. The compiler was buggy. There were few libraries, limited documentation and even less sample code. The community was very small but punched above its weight to support other members when they encountered problems. But we all knew that we had found something good and, despite these early teething problems, it was worth sticking with. With greater knowledge came intellectual satisfaction rather than contempt.

As time goes by ....

Over the last five odd years we have completed a variety of projects, for many clients in many different environments and domains. Each has brought its own unique challenges, but from each assignment we have learned: where to start; how to integrate Scala into a multi-language, multi-paradigm environment; how to build Scala web applications that scale and are easy to replicate; design patterns and "best practice"; about Scala libraries and what makes them more or less ge-

neral and flexible. And also of course how to construct the development environment and team structure that makes us most effective and efficient.

Health warning

What follows in an attempt to pass on some of the things we've learnt. We're not saying all these will be right for you in your environment, that they're the best ways of doing things or that they will work for everyone. This is just what we've picked up along the way.

Why we still use Scala? The benefits.

The benefits we originally sought to obtain from Scala have, largely, been realised but as we've been through full development cycles we've found that some of the delights of using Scala are both more subtle and more profound than we realised.

Expressivity

Scala is hugely capable at giving the programmer more power to express code in the way he wants to think about it. Be it through case classes, custom extractors for pattern matching, user-defined operators or just utilising existing Java libraries in more eloquent code, the language has charisma!

Conciseness

From expressiveness comes conciseness. Or at least, the option to be concise: it's for the programmer to decide whether to directly mirror the verbosity of Java, or to imitate Haskell's terseness. Unsurprisingly, most users will pick a happy medium!

Scala code is generally a boilerplate-free zone. There is often a tendency amongst certain developers to get a misplaced sense of satisfaction from writing boilerplate code: writing hundreds of lines of code without thinking about them seems like productivity, but it doesn't take a genius to work out that it's basically wasted time and completely contrary to the DRY (Don't Repeat Yourself) principle, maybe the single most important consideration in software engineering.

But why? If you find perpetual repetition in your code, it's most likely a deficiency of the language you're using. A few years ago, Jean-Paul Nerrière of IBM identified 1500 of the most useful English words and combined with a simplified English grammar he created a language called "Globish". It's great because it makes it very easy for non-English speakers to communicate. But just imagine how frustrating it is when you want to talk about your cousin but have to refer to her as your father's sister's daughter every time you mention her. Like Globish, Java simply doesn't have the vocabulary or constructs to avoid this repetition.

What might take several lines of code in Java, or the use of a common design pattern, might be expressed in a single line of functional Scala code. This not only makes it quicker to write but also makes it easier to pick up someone else's code and understand it, not to mention old, long-forgotten (and of course undocumented) code of your own...

Let's not overlook the other edge of this sword: that with conciseness can come incomprehensibility: anyone who has dabbled with Code Golf [1] will know the sorts of symbolic soup your code can resemble if conciseness is your only master. Scala is all about striking that balance.

Static typing

Many beginners find static typing a pain: You're trying to get something working quickly, but the compiler is just whinging about the sloppiness of your coding. Sure, the code might have worked, but you wouldn't have known without actually running it. But "oh!," I almost hear you crow, "I wrote a unit test first to check that..."

I'm going to be blunt about this: can you seriously tell me you would rather write a unit test than have the compiler tell you your program is never going to work, so there's no point even trying (and by the way, your errors are here, here, here and here)?

After a while static typing will become the biggest timesaver in your development cycle. It's the crux of what gives Scala developers so much confidence to develop. Instead of developing in tiny iterative steps, Scala mitigates the risk of making tremendous refactorings across multiple crosscutting concerns. If you've been using Scala for long enough, you won't think twice before embarking upon a significant rewrite; knowing that the Scala typechecker is there as an immensely powerful safety-net gives you the confidence to naively shuffle code around, safely assured that – although the compiler might give you a long 'to do' list of errors – once you've fixed them, your code will just work.

Speed and Robustness

Scala's combination of expressivity and static typing means in practice that it takes far less time to develop working applications, from tiny toy code, to heavyweight applications.

From a business perspective, this means you have a choice of doing more with the same resource, to develop applications more quickly, or do the same with less.

The greatest efficiency and effectiveness gains are made with the addition of frameworks, customised libraries to your domain, a well set up development environment, a well-structured and appropriately-skilled development team and robust architecture.

Native Scala libraries have been a while coming. This is possibly because existing Java libraries have been adequate for the purpose, but more likely because building good, reusable and effective frameworks is hard.

It's a common criticism of Scala that it's too complex. And it's true that Scala's feature list is not small. Most Scala developers probably use less than half of the features available to them. But unless you're building libraries or doing academic research, this is all that's necessary: all the productivity benefits stem from Scala's everyday features, not from the more esoteric details. If you don't need them, don't use them (and you won't even know they're there).

The Scala development team, ably lead by Martin Odersky, are very conscious of Scala's position at the intersection between cutting-edge academic research, and commercial pragmatism. Much thought has gone into the design of the language to make it coherent and practical across the length

of the learning curve, by means of appropriate documentation and meaningful error messages.

Java compatibility

In case you had made it this far without realising, Scala and Java are compatible. A fundamental premise of Scala's development has been full compatibility with Java. That is to say, you can use Java libraries in your Scala code, and you can use Scala libraries in your Java code.

But how is this possible, given Scala's vastly superior capabilities? Some ingenious ideas have gone into making Java's many warts evaporate when referencing Java from Scala. There's no need to cloud your mind wondering about special cases for primitives or boxing; Scala makes it an implementation detail.

But what about in the other direction? Well, unfortunately there's no water-into-wine miracle to suddenly turn Java into Scala just because you're using Scala libraries, so if you're calling Scala code from Java, you have to do it in the usual Java way, and you may need to know some implementation details. But let's not forget that compiled Scala is plain old Java bytecode, and the Java compiler can't tell the difference.

Developer Satisfaction

This should not be overlooked. Scala is simply a far more satisfying language to work with. On top of the added confidence you get as a programmer, there's never the sense that you're fighting the language to be productive; the code just flows.

The job of programming is fundamentally that of translating: programmers are given a human description of a problem which needs to be solved, and they convert that into a form the computer understands. Humans aren't getting any better at framing their problems in code, but Scala is making the computer better at understanding human language. This reduces the work involved in the translation; the steps necessary to encapsulate the problem as code, so the programmer can spend less time solving the syntax of the problem and more time solving the semantics.

Morale and productivity tend to be higher in Scala teams and the community is stronger because it's simply a more satisfying way to code, and seeing development happen more quickly makes the productivity more tangible to the development team.

Competitive edge

We believe that all of the above virtuously conspire to give us an edge (as Scala application developers and consultants), and an edge to our customers (reliant on speed of delivery, speed to change, quality and cost effectiveness).

How do I get started?

Clearly there are a number of different starting points and each will have its requirements: a startup will have different needs from an established organisation's Java development team. However there are a number of common questions that occur: What do I do first?

Analyse and prioritise

Many organisations will have a backlog of development needs: new features, new applications, new releases, major and minor changes along with a pipeline of projects that are desirable but have not passed business-case hurdles. Whilst it is our core belief that in the long term most of these would be better performed in Scala, the key issue is one of prioritisation: where is the greatest benefit to be attained soonest, and with the least risk?

Sadly there is no easy answer to this question – we have developed a process, enabled by software (developed in Scala, of course) to facilitate a development team (senior technical managers, line-of-business owners, and developers) identify the best projects to start with. The key considerations are:

- Complexity
- Risk
- Business criticality and impact
- Performance requirements
- Scalability
- Scala expertise
- Quality
- Timescales
- Team development

Analysis of these factors allows us to produce a portfolio of candidate projects to migrate to Scala.

Assess and develop your organisation

You might have a great team of developers, but which of them have the motivation and propensity to become great Scala developers? Your prioritisation above will give you a good idea about which projects should use Scala first, and some idea of the resources you will need but what team will be optimal to deliver now and in the future?

Do you already have the Scala resources or are you re-training, hiring or seeking to bring in outside expertise to help? You may find that a lot of your Java developers are secret Scala devotees in their spare time.

Many organisations may choose to bring in Scala experts at the start to work alongside their Java developers, to help set things up, get things going and to act as mentors once the team is running on their own.

Create your development environment and process

Despite being relatively new, there are already a great many open-source frameworks, methodologies and tools on offer from the Scala community to help you develop your first Scala application. Each has its own merits and we don't believe there is a one-size-fits-all answer. You could ask one of your developers to research and recommend an answer, or you could seek advice from a Scala specialist to recommend the optimal toolset, environment and process based around your existing setup, requirements and future aspirations.

Position Scala within your environment

Most organisations have a heterogeneous environment involving a range of technologies which have been well integra-

ted to fulfil their needs. These include databases, application frameworks, business applications, security infrastructure, web interfaces and multiple layers of messaging and communications services and protocols.

You will need to understand where and how Scala needs to integrate; how to achieve the greatest benefit from it; how to insulate your application from the future, and all the uncertainty that brings.

Operational management and support

Having reached the milestone of having initial applications developed and running, they need to be supported and managed. Scala fits comfortably into a Java environment, though there are still specific support considerations. Scala applications can be considerably more robust than other languages, though – like any system – require support and maintenance. This could be developed through an internal operations team, while second and third level support may be provided by the development team or a third party.

Quality assurance and audit

Any new Scala development, especially by an inexperienced team should be quality-assured and audited. Scala's flexibility offers many possible ways of implementing the same project, but each will have different performance characteristics, and some will be easier to manage than others. Scala goes a long way towards facilitating best practice in software development, but there's more than one way to skin a cat. It's not always clear which architectural approach is the most appropriate from a perspective taking into account long-term considerations like maintainability and future releases.

Development support

The success of many projects relies on the expertise and experience of external resources. Bringing in vital skills can be important at critical times on all projects. The Scala market is on the cusp of great things, but it's not yet as mature as the Java marketplace.

More mature Scala organisations

Many organisations have been using Scala for some time. Perhaps not for all their development projects but with some elements of Scala mixed with the major Java applications. Very few have the experience of evaluating how effective their Scala implementations actually are, how they are performing compared to how they could perform, how their teams are working and how efficient their development environment is.

Summary

We believe that developing in Scala offers significant benefits to most organisations, and for organisations already using Java, Scala offers low-hanging fruit as a boost to productivity. These benefits are obtainable almost immediately but will be even more applicable in the long term as well. We believe that maximising the benefits of Scala implementation relies on the skills and experience of the team involved.

Scala Technology [2] exists to help organisations maximise the benefit of their use of Scala. Working closely with Martin Odersky and the Scala team in Switzerland, Scala Technology promotes the use of best practice in implementing Scala in industry. Led by the team which launched the first commercial applications in Scala in 2004, our team has the tools, processes and experience to develop teams and applications that can truly take advantage of everything Scala has to offer.

Jon Pretty is a technologist, developer, systems architect and Scala evangelist with over five years' hardcore Scala experience under his belt.

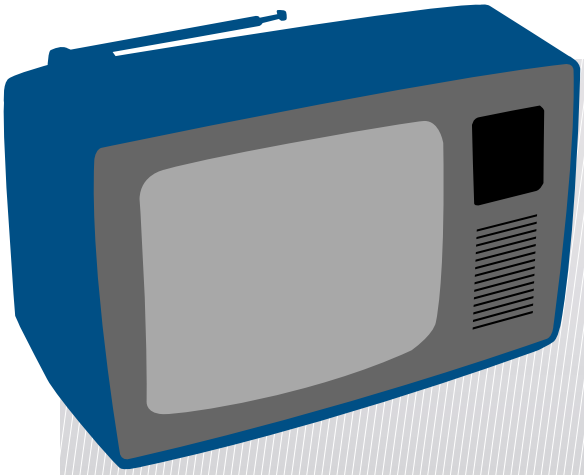
Vince Kenealy has 25 years experience in the IT industry focused on the business value of innovation and service garnered at IBM, TIBCO and Symantec.

Peter O'Sullivan has worked with many organisations in the public and private sector, over the last 20 years, helping them generate maximum value from their use of IT.

Links & Literature

[1] http://codegolf.com

[2] www.scalatechnology.com



More Scala on JAX TV

Are you reading JAXmag while connected to the internet? Then check out JAX TV, with a keynote from Scala creator Martin Odersky, plus an interview with Scala geek Ted Neward!

Play the Game!

Tutorial: Game of Life in Scala

Most Java developers have probably heard about Scala, a promising new language for the Java Virtual Machine. In this tutorial we get our hands dirty in order to get a feel for what Scala is really about. Please join us, as we build a small application step by step.

Heiko Seeberger, Marcus Denison

There is always one question at the beginning of a tutorial: "What kind of example application should we build?" It should be appealing and, if possible come with a graphical user interface. However, making it too complex could end up breaking the mold. Furthermore we want to get along without any dependencies (except for test tools,) because we want to check out Scala itself and not drift into any frameworks like Lift [1] or Akka [2].

Game of Life: Rules

Our playing field is an infinite grid. Every square represents one cell, which has one of two possible states: alive or dead. A generation is made up from all cells at a certain point in time. The following rules describe how we get from one to the next generation:

- Any live cell with fewer than two live neighbours dies because of under-population.
- Any live cell with more than three live neighbours dies because of overcrowding.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any dead cell with exactly three live neighbours becomes a live cell as if by reproduction.

So, we have chosen an idiomatic and simple implementation of the fantastic Game of Life [3]. If you haven't heard about this interesting biological simulation, you really should take a look. We will not focus on a highly performant or distributed solution, but rather attempt to present Scala to you as simply as possible.

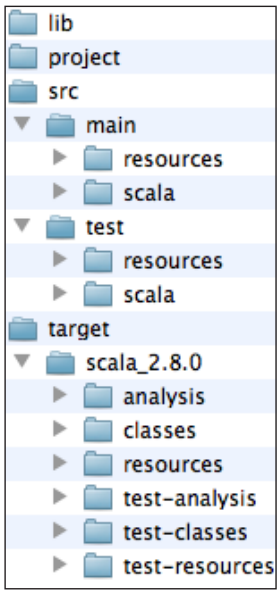
Join us in building this example step by step. You can take a look at the code provided on http://www.github.com/weiglewilczek/gameoflife.Our sample project is open source and is published under the Eclipse Public License [4]. Using Git, you can take a look at the commit history to fully understand each step we take.

Development Environment

Another important question we have to answer: How should our development environment look? Java developers are used to the comfort offered by IDEs like Eclipse, IntelliJ IDEA or NetBeans and the release of Scala 2.8 has finally brought us a few good plugins for the "big" IDEs.

Nevertheless, we will not require an IDE at this point! Developers tend to be picky about IDEs, which means that some might stop reading if we selected the "wrong" one. Moreover our example has very few lines of code, which is typical for Scala. Thus, we will get along with a plain text editor. But of course you may use any IDE to follow this example.

Compiling? Testing? Running the application? For this matter we are bringing in a nice build tool called SBT [5] (Sim-



III. 1: SBT project structure

ple Build Tool). If you know Ant or Maven, you will love SBT. If you're new to SBT, you'll quickly come to understand why it has "Simple" in its name. SBT is written in Scala itself and takes into account some of its specialities. Actually building works pretty fast, starting the interactive console (REPL) with the entire class path of the project just works, and we are able to cross-build against various Scala versions - these are just a few features we wish to highlight. Projects like Lift or Akka are already using SBT, or planning to use it.

Of course SBT is not the only possibility out there for compiling and running Scala code. So if you have your preferred IDE or want to use the command line tools that ship with the Scala distribution, you may skip the next part.

Starting a new SBT Project

Before starting our project, we need to "install" SBT. It's really easy since SBT is only a JAR file: Download the latest version sbt-launch-0.7.4.jar from the SBT website, you'll find this in the Downloads [6] section. After that, just follow the simple steps provided in the Setup [7] section: "how to create a startup script" Within MacOS it could look like this:

```
java -Xmx1024M -jar sbt-launch-0.7.4.jar "$@"
```

One more step before starting a project: make yourself a project-folder, e.g. gameoflife and go into that directory using the shell. Now we are good to go; run the SBT startup script to create a new project and answer the following questions:

```
gameoflife$ sbt
Project does not exist, create new project? (y/N/s) y
Name: gameoflife
Organization: com.weiglewilczek.gameoflife
Version [1.0]:
Scala version [2.7.7]: 2.8.0
```

Following the output, notice that SBT is downloading Scala 2.7.7 and Scala 2.8.0, which makes our lives easier as we don't have to worry about that anymore. You may wonder why SBT is also downloading Scala 2.7.7? That's because SBT itself runs on 2.7.7.

Starting SBT without any arguments will call the interactive SBT console, which provides commands such as exit, compile, test or run. Running exit or compile should be self-explanatory, but what happens when we execute run? It will look for and execute a singleton object with a main method. Now, there is a nifty little feature: when putting the "tilde" in front of the command, SBT will go into "triggered" mode where it will wait for changes in the filesystem and with any change, the command will be executed again. Using ~compile as an example, SBT will compile every time a file in the project is changed and saved. For further commands and details, please take a look at the excellent documentation on the SBT website.

Running test for the first time will create some additional folders. When taking a look at illustration 1 you will see how your project should look. If you have ever worked with Maven, the src and target folders should look familiar, whereby the various Scala versions we are building our project against are held in the target folder. Looking at the project folder we can find configuration and downloaded files. We can drop libraries into the lib folder which are automatically added to the class path. Another feature compliant with Maven are "Managed Dependencies," but we don't actually need them here.

Before we get down to business, let's code up and run the most legendary piece of code. Switch into the folder src/main/scala and create Hello.scala:

```
object Hello {
  def main(args: Array[String]) {
    println("Hello World!")
  }
}
```

Calling run in SBT will give us the well known "Hello World!" message.

Classes and Packages

Being a Java developer, we know the principles of object-orientated programming. Due to this, we will take an approach at Scala from the side we already know. That's indeed possible, because Scala is a hybrid language combining the best features of object-orientated and functional [8] programming. We won't get too theoretical in this tutorial, but have outlined some of the basics of object-orientation in Scala in the adjacent box.

The Game of Life is all about dead and live cells. Therefore we will start with creating a file called Cell.scala in the source directory src/main/scala. There we define the class Cell using the keyword class, just like in Java:

```
class Cell
```

Starting with an empty class without fields or methods, we do not need curly braces or a semicolon thanks to the semicolon inference. Unlike Java, there is no access modifier public, because that is the default in Scala.

Packages are similar yet more powerful compared to Java packages. One important difference is that there is no need to mirror the package structure in the file system. Therefore, it is possible to omit the "root package" of a project, but of course underlying packages should be represented by the directory structure. In our example we will have only the com.weiglewilczek.gameoflife package and therefore there is no need for subfolders. According to this our Cell should look like:

```
package com.weiglewilczek.gameoflife
class cell
```

The REPL

Although our cells are not too useful yet, we now want to introduce an important Scala tool: the interactive Scala console, also called Read Evaluate Print Loop (REPL). Running console in SBT will give us the REPL using the complete class path of our project. In the REPL we can write code and have it interpreted and executed immediately. With :help we can list various commands which we might need.

First, let's get our package imported. We'll use the keyword import, followed by a whitespace and use the tab key to find our package with the autocompletion.

```
scala> import com.weiglewilczek.gameoflife._
import com.weiglewilczek.gameoflife._
```

Similar to Java's "*" is the underline in Scala, which means that everything gets imported from our package.

Now, let's create a new instance of our cell using the keyword new just as we know it from Java. Since we are not using any arguments, parentheses are not needed yet.

```
scala> new Cell
res0: com.weiglewilczek.gameoflife.Cell = com...
```

Watching the REPL output, it will automatically create a new variable called res0 (later we will point out how we can provide our own variable names), printing its type and the result of the toString method.

Let's quit the REPL entering :quit (or :q) and get back to the code where we can give our cells some additional features.

Class Parameters, Fields and Methods

As cells are placed on a grid, we want to add their coordinates. Adding so called class parameters enclosed in parentheses after the class name will make this possible:

```
class Cell(x: Int, y: Int)
```

Being purely object-orientated

Everything being an object in Scala makes it a little stricter tha Java. Forget about static and primitive types. Except for the allocation there aren't any operators either, only methods.

As an example, expecting two primitives and a plus operator when adding two numbers would be wrong here. Instead we have got two Int objects and one of them calls the + method, which is defined on Int. Since + is a method, we could write 1.+(2), but Scala allows us to write 1 + 2. That is the so called infix operator notation and can be used whenever a method has only one argument.

Instead of static, Scala has got singleton objects, which represent a class and an instance at the same time. These are first class objects and may be used as arguments when calling methods.

That looks different from that what we know from Java, right? First, there are no class parameters in Java and second, we annotate any parameters just like we do in UML, starting with the name followed by a colon and then the type.

OK, but what do these class parameters mean? And don't we have a constructor? The compiler will take the class parameters and create the so called primary constructor for this class. This means that now we create our cells like this:

```
scala> new Cell(1, 2)
res0: com.weiglewilczek.gameoflife.Cell = com...
```

Class parameters are like private fields which means that we cannot access the coordinates and we cannot change the state of our cells. Later we will change the visibility of the coordinates, but immutability is one of the most important principles of functional programming. Even in Java, immutable objects are well known and going by Effective Java [9] it is always better to use immutable, rather than mutable, objects.

Using the keyword val defines an immutable field or local variable, just like using the final modifier in Java. There is also var to define mutable fields or local variables. Now we will create a private immutable field which will return the position of a cell as a String:

```
private lazy val position = "(%s, %s)".format(x, y)
```

But how does the compiler know that this field is a String? The Scala compiler is pretty smart: based on the value that is assigned, it recognizes that position shall be a String. This feature - called type inference - is also applicable for methods and generic types, making Scala code very lightweight even though it is statically typed.

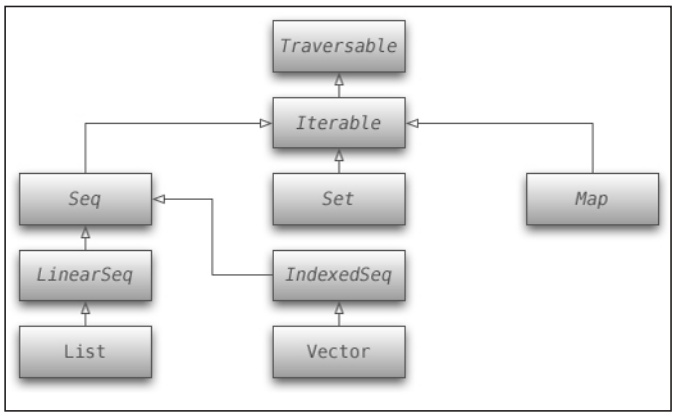
But, what is the keyword lazy doing there? When using lazy, the initialization and therefore the evaluation of the "right side" only happens when the field is being approached for the first time. Until now that was not happening, since position is a private field and it is not used internally in Cell. But taking a look at the REPL output suggests that we could use position to overwrite the toString method, because our cells are immutable:

```
override def toString = position
```

Using the keyword def will create a method. override must be used when overwriting concrete methods or fields and may optionally be used when implementing abstract ones. As before, type inference allows us to omit the return type. In this case that's fine, but when implementing non-trivial methods it is always good style to give the type. In our case it would look like this:

```
override def toString: String = position
```

Even if we do annotate the type, the code is still more lightweight than it is in Java: first, we don't need curly braces when having a one-liner and second we don't need a return statement, since a method returns the last line of a block by default. Moving ahead in this tutorial, we'll see more of this



III. 2: Important collections

in more complex methods. Let's see how our changes will affect things on the REPL:

```
scala> new Cell(1, 2)
res0: com.weiglewilczek.gameoflife.Cell = (1, 2)
```

That looks better, right? However, we still can't access the coordinates, because the class parameters are still private. Putting the keyword val in front will make them public:

```
class Cell(val x: Int, val y: Int) { ...
```

Let's take a quick look into the REPL in order to see how this works. Now we are also using the keyword val to define a local variable with a meaningful name which we can use to further access our cell:

```
scala> val cell = new Cell(1, 2)
cell: com.weiglewilczek.gameoflife.Cell = (1, 2)
scala> cell.x
res0: Int = 1
```

Case Classes

Now let's get even easier. Using the keyword case in front of the class definition will convert our cell to a case class. This makes the compiler provide us with the following features, amongst others:

- Class parameters are vals by default, so there is no need to write val in front of our parameters anymore.
- Instances may be created without the keyword new, but explaining this would be to much for our tutorial.

Listing 1: Cell.scala

```
package com.weiglewilczek.gameoflife
case class Cell(x: Int, y: Int) {
  override def toString = position
  private lazy val position = "(%s, %s)".format(x, y)
}
```

- Reasonable implementations are provided for equals, hashCode and toString; in this case we have overwritten toString ourselves, of course.

In listing 1, the complete code for our cells is shown after all these changes were made. Let's take a quick look at the REPL, just to make sure that things work as expected:

```
scala> val cell = Cell(1, 2)
cell: com.weiglewilczek.gameoflife.Cell = (1, 2)
scala> Cell(1, 2) equals Cell(1, 2)
res0: Boolean = true
scala> Cell(1, 2) equals Cell(1, 1)
res1: Boolean = false
```

For Comprehensions

This might be a tick too early, but referring to our example it is necessary that we now look at one very interesting language feature called "for comprehensions." In Scala like in other functional programming languages, it is common that everything has a return value. For example, there is an if-else, but that's not only for control flow like in Java but also has a result value, which can be assigned to a variable or can be used as a return value for a method. Therefore the if-else in Scala matches the conditional operator (x ? y : z) in Java.

Now let's talk about for comprehensions. In their primitive form, they are like loops:

```
for (i <- 1 to 10) { ... }
```

Exceptionally having no return value, they will only wield some sort of side effect. By the way, to is not a keyword, but a method which, because of one advanced feature called implicit conversions, may be used on Int values and returns a collection of the type Range.

Having for comprehensions in their functional form, they end with the keyword yield and have a result value. A quick little example:

```
scala> for (i <- 1 to 10) yield i + 1
res0: ...IndexedSeq[Int] = Vector(2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
```

We can observe that a collection (Range 1 to 10) is mapped onto another one (Range 2 to 11). Now we want to use for comprehensions in order to calculate the neighbours of a cell:

```
def neighbours: Traversable[Cell] =
  for {
    i <- (x - 1 to x + 1)
    j <- (y - 1 to y + 1) if (i != x) || (j != y)
  } yield Cell(i, j)
```

OK, let's explain this from the top to the bottom:

- Since we do not have a trivial one-liner here, we are giving the return type; as one can observe, type parameters are defined in brackets.

- Using curly braces instead of round ones; thanks to this, we can write more lines without using a semicolon.
- Using two so called generators in an effort to run trough all x- and y-coordinates around our cell.
- To lock out our own cell we are using a filter.
- Employing yield creates a collection of neighbour cells.

Taking a look at listing 2, we see the complete code for our cell after applying these changes. Let's take a quick look at the REPL again, where we find the eight neighbours for the given cell as expected:

```
scala> cell.neighbours
res1: Traversable[...Cell] = Vector((0, 1), (0, 2), (0, 3), ...)
```

Unit Testing with specs

Running code in the REPL will give us a good feel for our code, but of course the need for systematic testing also applies to Scala. We could use JUnit, but there are some Scala test tools which are very powerful in creating self-explanatory tests.

We are going to take a look at specs [10]: Please download the current version 1.6.5 for Scala 2.8.0 (specs_2.8.0-1.6.5.jar) and put the JAR file into the lib directory of the project.

Switch to the src/test/scala folder, which is our test folder, and create a class called CellSpec. We will use the same package as we have used for Cell and extend the class Specification form specs. Put the following lines into the body of the class:

```
"A Cell" should {
  "have eight neighbours" in {
    Cell(0, 0).neighbours must haveSize(8)
  }
}
```

That looks more like prose, but is valid Scala code. Again we have implicit conversions working for us, so that the methods should and can be called on the Strings or the method must on the result from neighbours. As a result, we have got a pretty self-explanatory test.

Concluding, let's see how tests are executed in SBT. Executing test will result in all tests being compiled and executed.

Listing 2: Cell.scala

```
package com.weiglewilczek.gameoflife
case class Cell(x: Int, y: Int) {
  def neighbours: Traversable[Cell] =
    for {
      i <- (x - 1 to x + 1)
      j <- (y - 1 to y + 1) if (i != x) || (j != y)
    } yield Cell(i, j)
  override def toString = position
  private lazy val position = "(%s, %s)".format(x, y)
}
```

As we can see, the strings which describe the test in our Scala code are used in the test report. Failed tests in particular, are outlined very nicely. To see how a failed test looks, we just apply a little change in the for comprehension like this: i <- (x – 1 to x – 1):

```
[info] A Cell should
[info]   x have eight neighbours
[info]    'Vector((-1, -1))' doesn't have size 8. It has size 1 ...
```

Collections

After finishing the cells, we will move to the generations, which represent the collective of all cells at a specific time. This leads us to collections, which are perfectly made for demonstrating the functional character of Scala.

Taking a look at illustration 2 we will see important representatives of the collection hierarchy. Traversable with about 100(!) methods is found on top. That amount makes it clear that Scala collections are very powerful, which is typical for a functional programming language.

We want to describe a generation based on its living cells. Let's create a new class Generation with the class parameter / field aliveCells of the type Set[Cell]:

```
package com.weiglewilczek.gameoflife
class Generation(val aliveCells: Set[Cell] = Set.empty)
```

We would like to point out two things here: for one thing, collections are typed in Scala. No surprise, because Java generics also came from Martin Odersky, Scala's "godfather." Another thing, we are defining an empty Set as default value, which is a new feature of Scala 2.8. With this, we can create a generation without the aliveCells argument.

```
scala> val generation = new Generation
generation: com.weiglewilczek.gameoflife.Generation = ...
scala> generation.aliveCells
res0: Set[com.weiglewilczek.gameoflife.Cell] = Set()
```

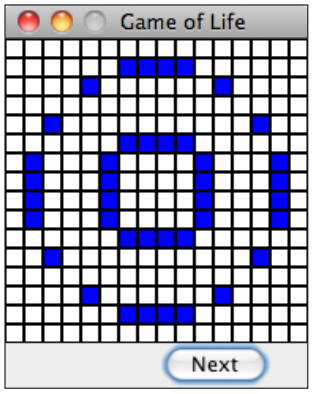
Of course, defining preconditions is also a part of a good style in Scala. Let's define a precondition where null is not valid for the aliveCells parameter. For this matter we use a method called require, which comes from the singleton object Predef whose members are automatically imported by the Scala compiler.

```
require(aliveCells != null, "aliveCells must not be null!")
```

Filtering Collections

According to the game rules, when moving from one given generation to the next we have to determine the (number of) alive neighbours for a cell. For this we can make use of our already implemented method Cell.neighbours, although we still have to determine which of the resulting neighbours are alive.

For this purpose we will employ the power of functional programming, as we use the filter method that is a member of every



III. 3: Swing surface

collection. This method does not take a "normal" argument, but a function. Functions are similar to methods, since they may be called with arguments and have a return value. However, functions are not members of a class, but objects themselves, just in terms of "everything is an object." As for numbers and Strings, so called literals are available for functions, too, which gives us the opportunity to simply write down a function.

The filter method expects a function that takes a parameter of the same type as the one the collection is parameterized, and returns a Boolean. In our example we have to provide a function that has a Cell parameter:

```
private def aliveNeighbours(cell: Cell) =
  cell.neighbours filter { neighbour => aliveCells contains neighbour }
```

We notice that function literals are written with the arrow symbol. On the left side we have our arguments and on the right side the body of the function. Once again, thanks to the type inference, we don't have to give any types.

Also, a shorter notation is possible. filter requires a function Cell => Boolean and looking at the contains method from the Set aliveCells we see the exact same signature. Luckily the Scala compiler is working for us, converting this method into a function. All we have to write now is:

```
private def aliveNeighbours(cell: Cell) =
  cell.neighbours filter aliveCells.contains
```

Besides the alive neighbours, we also need the dead ones. The code is pretty similar to aliveNeighbours. Although, since we are not only working with the contains method but also using some extra logic with the negation, we have to write the function the "wordy" way.

```
private def deadNeighbours(cell: Cell) =
  cell.neighbours filter { neighbour => !(aliveCells contains neighbour) }
```

Our generation should at the moment look like listing 3.

Listing 3: Generation.scala

```
package com.weiglewilczek.gameoflife
class Generation(val aliveCells: Set[Cell] = Set.empty) {
  require(aliveCells != null, "aliveCells must not be null!")
  private def aliveNeighbours(cell: Cell) =
    cell.neighbours filter aliveCells.contains
  private def deadNeighbours(cell: Cell) =
    cell.neighbours filter { neighbour => !(aliveCells contains neighbour) }
}
```

Mapping Collections

Now we want to determine the next generation, which is built from cells that stay alive and those that wake up from the dead.

```
def next: Generation = {
  val stayingAlive = ...
  val wakingFromDead = ...
  new Generation(stayingAlive ++ wakingFromDead)
}
```

Once again we can see that there is no return necessary, as the last line in Scala is always our return value. Both variables staylingAlive and wakingFromDead are supposed to be Sets, which allows us to use the ++ method to create a new Set[Cell] for the next generation.

From what we already know, we are able to implement staylingAlive: Let's filter out all alive cells which have only two or three neighbours:

```
val stayingAlive =
  aliveCells filter { 2 to 3 contains aliveNeighbours(_).size }
```

This is another short form to write function literals: we abandon the parameter list and arrow symbol, directly writing the function body, where the underline is a placeholder for the missing parameter. Having more parameters would force us to supply more underlines. Of course we could also have provided a more explicit notation:

```
aliveCells filter { cell => 2 to 3 contains aliveNeighbours(cell).size }
```

The next step is to implement wakingFromDead. Luckily we don't have to look at all the dead cells on the playing field, but only at those that are in the neighbourhood of alive cells, because only a cell with three neighbours will wake up. Therefore we have to provide a way to get from the alive ones to their dead neighbours. We already have the method deadNeighbours for a single cell, but how do we retrieve the dead neighbours for all alive cells?

Getting our result in the imperative way often used in Java, we would probably provide some sort of loop and write very detailed code in order to define how things must happen. Using functional programming our life is a tad bit easier. We just write what we want, but not how it shall be done!

OK, back to the topic! What we are trying to do here here in two steps: we know the set of alive cells (aliveCells) and want to transform this into the set of dead neighbour cells. Thereafter, we have to filter if the dead neighbour cells have got exactly three alive neighbours.

For our first task, we'll try the collection method map, which transforms a collection into a new one. Let's look at a quick example in the REPL:

```
scala> List(1, 2, 3) map { x => x + 1 }
res15: List[Int] = List(2, 3, 4)
When porting this to our example, it could look like this:
aliveCells map deadNeighbours
```

Once again, the Scala compiler makes our lives easier, making a function out of the deadNeighbours method, so that this code can compile. But that is not what we wanted, because we are transforming a Set[Cell] into a Set[Traversable[Cell]], since deadNeighbours returns a Traversable[Cell] for every Cell. For such a case we have a collection method called flatMap, which takes nested collections and literally pounds them flat. Once again a little example in the REPL:

```
scala> List(1, 2, 3) map { x => List(x - 1, x, x + 1) }
res18: List[List[Int]] = List(List(0, 1, 2), List(1, 2, 3), List(2, 3, 4))
scala> List(1, 2, 3) flatMap { x => List(x - 1, x, x + 1) }
res19: List[Int] = List(0, 1, 2, 1, 2, 3, 2, 3, 4)
```

With this knowledge we can implement wakingFromDead in the following way:

Listing 4: Generation.scala

```
package com.weiglewilczek.gameoflife
class Generation(val aliveCells: Set[Cell] = Set.empty) {
  require(aliveCells != null, "aliveCells must not be null!")
  def next: Generation = {
    val stayingAlive =
      aliveCells filter { 2 to 3 contains aliveNeighbours(_).size }
    val wakingFromDead =
      aliveCells flatMap deadNeighbours filter { aliveNeighbours(_).size == 3 }
    new Generation(stayingAlive ++ wakingFromDead)
  }
  private def aliveNeighbours(cell: Cell) =
    cell.neighbours filter aliveCells.contains
  private def deadNeighbours(cell: Cell) =
    cell.neighbours filter { neighbour => !(aliveCells contains neighbour) }
}
```

Listing 5: GenerationSpec.scala

```
package com.weiglewilczek.gameoflife
import org.specs.Specification
class GenerationSpec extends Specification {
  "Generation.next" should {
    "return an empty Generation for the empty Generation" in {
      new Generation(Set.empty).next.aliveCells mustBe Set.empty
    }
    "yield three horizontal cells for three vertical cells" in {
      val vertical3 = Set(Cell(0, -1), Cell(0, 0), Cell(0, 1))
      val horizontal3 = Set(Cell(-1, 0), Cell(0, 0), Cell(1, 0))
      new Generation(vertical3).next.aliveCells mustEqual horizontal3
    }
    "stay constant for a 2x2 block of cells" in {
      val block = Set(Cell(0, 0), Cell(0, 1), Cell(1, 0), Cell(1, 1))
      new Generation(block).next.aliveCells mustEqual block
    }
  }
}
```

```
val wakingFromDead =
  aliveCells flatMap deadNeighbours filter { aliveNeighbours(_).size == 3 }
```

Taking a look at this, you can see that we are chaining two calls: First calling flatMap on top of aliveCells giving deadNeighbours as an argument. Until now we have had a return value of Set[Cell] that will be called using filter with a function literal, which gives us the dead cells with three alive neighbours.

Listing 4 provides the complete code for the class Generation and Listing 5 provides a few tests, which feature two interesting source generations. On the one hand we have the so called "blinker:" three horizontally aligned cells that are transformed into three vertically aligned ones, and back again. On the other hand we have a generation called "block" which does not change at all.

Swing GUI

We're almost there! It is fascinating to watch the process of how generations evolve. In order to see this, we are going to build a little graphical user interface.

The goal is to have a nice little tutorial, and not how to build a powerful visualization of the Game of Life. So we will create a basic user interface, shown in illustration 3, which has the following features:

- Only views a finite section of the infinite playing field.
- The opportunity to toggle the status of the displayed cells via the mouse at any time.
- A "next" button which take us to the next generation.

As we want to get along with Scala we are only using the Scala Swing library [11], which makes working with Swing easier. Unfortunately explaining in detail what we have done in listing 6 would break the mold of this tutorial. Even though it is not too much code, there are a few concepts used that are way ahead of the basics. In the following, we try to pick out a few nice little highlights.

First, we see how easy it is to write a Swing application: We are extending SimpleSwingApplication and overriding the top method. Components are built by adding more components to the contents field. Overwriting reactions in a component and adding so called Reactions is necessary in order to handle events. Actions are associated directly with a button, as calling the singleton object Button and providing the code for the action as an argument.

Next we return to a few concepts that we have talked about and used earlier, e.g. building the playing field with a for comprehension or passing a function determining whether a cell is alive or dead as an argument.

Even though the class SwingUI is a bit more complicated than we have learned up until now, we will get a good idea of the possibilities offered by Scala and the Scala Swing library.

Conclusion

So far we have covered the most important Scala basics. Not only have we learned about the language itself, but also dis-

Listing 6: SwingUI.scala

```
package com.weiglewilczek.gameoflife
import java.awt.Color
import javax.swing.border.LineBorder
import scala.swing._
import scala.swing.event.MouseClicked
object SwingUI extends SimpleSwingApplication {
  override def startup(args: Array[String]) {
    dimension = args.headOption map { arg =>
      try { arg.toInt } catch { case _ => DefaultDimension }
    } getOrElse DefaultDimension
    super.startup(args)
  }
  override def top = new MainFrame {
    title = "Game of Life"
    resizable = false
    contents = new BoxPanel(Orientation.Vertical) {
      contents += new GenerationPanel(new Generation, update(contents(0) = _))
      contents += Button("Next") {
        val next = contents(0).asInstanceOf[GenerationPanel].generation.next
        update(contents(0) = _)(next)
      }
    }
    private def update(swap: GenerationPanel => Unit)(generation: Generation) {
      swap(new GenerationPanel(generation, update(swap)))
      pack()
    }
  }
  private class GenerationPanel(
      val generation: Generation,
      update: Generation => Unit)
      extends GridPanel(dimension, dimension) {
    contents appendAll cells
    private lazy val cells = {
      def cell(cell: Cell, alive: Cell => Boolean) = {
        new FlowPanel {
          border = LineBorder.createBlackLineBorder
          background = if (alive(cell)) Color.BLUE else Color.WHITE
          contents += new Label("")
          listenTo(mouse.clicks)
          reactions += {
            case e: MouseClicked => {
              val aliveCells =
                if (alive(cell)) generation.aliveCells - cell
                else generation.aliveCells + cell
              update(new Generation(aliveCells))
            }
          }
        }
      }
      for {
        y <- 0 until dimension
        x <- 0 until dimension
      } yield cell(Cell(x, y), generation.aliveCells.contains)
    }
  }
  private lazy val DefaultDimension = 25
  private var dimension: Int = _
}
```

cussed tools like the REPL and SBT. Hopefully with this practical example we have not only shown that Scala is concise (e.g. Cell) and understandable (e.g. CellSpec), but is also capable of offering straightforward solutions to many problems through its functional character (e.g. Generation.aliveNeighbours).

We are certainly hoping to get some feedback, questions, bug-reports (please use the issue tracker [12]) or ideas on how to improve our tutorial! Have fun with Scala!

Heiko Seeberger is managing director of WeigleWilczek and is responsible for the technological strategy with a strong focus on Java, Scala, OSGi, Eclipse RCP, Lift and Akka. Additionally he is an active open source committer and shares his expertise in articles and conference talks.

Marcus Denison is studying Computer Science at the University of Applied Sciences Schmalkalden, focusing on software development. Presently working on projects for the Weigle Wilczek GmbH expanding his remarkable Scala and Lift knowledge.

Links & Literature

[1] Lift Web-Framework: http://liftweb.net

[2] Akka: http://akkasource.org

[3] Game of Life: http://en.wikipedia.org/wiki/Conways_Game_of_Life

[4] Eclipse Public License: http://www.eclipse.org/legal/epl-v10.html

[5] Simple Build Tool: http://code.google.com/p/simple-build-tool

[6] Download SBT-Launcher: http://code.google.com/p/simple-build-tool/downloads

[7] SBT start script: http://code.google.com/p/simple-build-tool/wiki/Setup

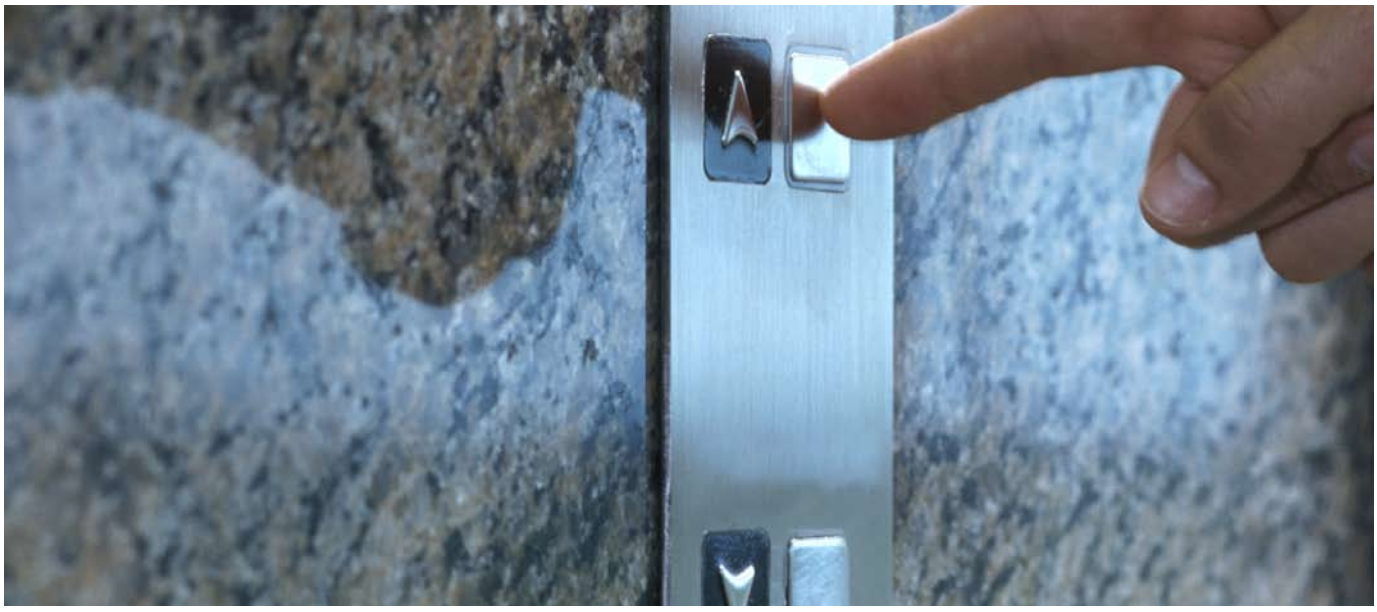
[8] Functional programming: http://en.wikipedia.org/wiki/Functional_programming

[9] Joshua Bloch, Effective Java: http://java.sun.com/docs/books/effective

[10] specs: http://code.google.com/p/specs

[11] Scala Swing: http://www.scala-lang.org/sid/8

[12] Issue Tracker: http://github.com/weiglewilczek/gameoflife/issues



Web Framework for Scala

New in Lift 2.0

Lift [1] is one of the few web frameworks for Scala and is, most famously, the framework that the location-based game Foursquare is built on. In this article, co-author of "The Definitive Guide to *Lift*" and Lift team member Marius Danciu, gives us his run-through of the best new features in Lift 2.0.

Marius Danciu

Lift 2.0 was released in June 30 2010. New in Lift 2.0, is NoSQL support including built-in support for MongoDB and CouchDB. Using the Record framework, Lift abstracts the underlying persistence store allowing users to build support for a large variety of data back-ends. Lift 2.0 comes with support for MongoDB and CouchDB.

Also new in Lift 2.0, is high performance JSON support including an elegant JSON DSL and bidirectional JSON <-> Class conversion. Lift allows working with JSON constructs in a type-safe manner via bi-directional conversions, query support for JSON sub-construct etc. Listing 1 shows an example.

There is powerful, concise REST support, as Lift facilitates the implementation of REST APIs taking leverage of Scala pattern matching. Thus, we can pattern match by the request path, HTTP method type, Accept header etc. Example:

```
serve {
case "api" :: "static" :: _ XmlGet _=> <b>Static</b>
case "api" :: "static" :: _ JsonGet _ => JString("Static")
}
```

Here, first case matches /api/static path for GET method and second case matches /api/static path for GET method if Accept header points to a JSON mime type.

There is support for running Lift apps outside of a J/EE Servlet container. Lift fully abstract the HTTP API stack in order to allow using Lift outside of JEE web containers. Using different implementations of Lift's HTTP provider API, one can run Lift on top of other containers such as Netty or even on top of proprietary HTTP servers.

Lift 2.0 features declarative systems for single Screen input and validation [2], as well as multiple screen Wizards. Much of the web is creating input forms for users to submit, validating those input forms and if the forms pass validation, an action is performed. If the forms don't pass validation, the user is told which fields caused the validation problems and is given an opportunity to fix the problems. Lift provides a single-screen input/validation mechanism called LiftScreen and a multi-page input/validation mechanism (with stateful next/previous buttons) called Wizard. This post will discuss LiftScreen and the next post will discuss Wizard.

Both Wizard and Screen share the following attributes:

- All logic can be tested without involving HTTP
- All logic is declarative
- All state is managed by Lift
- The back-button works as the user would expect it to work
- The form elements are strongly typed
- The rendering logic and templates is divorced from the form logic

Also:

1. Radically improved development experience, including much better error messages and support for dynamically changing system configuration.
2. Support for enterprise infrastructure including JTA and LDAP.
3. Improved Comet support including modern browser detection and better connection starvation detection.
4. Improved support for testing including super-concise dependency injection and run-mode detection.
5. Support for Simple Build Tool [3].
6. Performance improvements.

This is an important step for Lift as it becomes a more and more robust web framework with features helping address-real business needs. Lift will continue evolving in multiple dimensions such as:

- features set
- improved programming model – i.e. evolving more towards functional programming paradigms
- robustness
- documentation
- community support

All these aspects are strongly driven by the Lift community and committer's vision.

Marius Danciu has been working with various Java technologies for about 9 years mostly architecting designing and implementing server side applications based on Java platform. For a relatively small period of time he has designed and developed Windows mobile applications using C/C++. For about two and a half years he has been a committer for Lift framework contributing with various features, bug became and trying to help out Lift community folks. It's safe to say that Scala quickly become his favorite programming language due to numerous reasons, such as expressiveness, conciseness, powerful functional programming support, powerful type system etc.

Links & Literature

[1] Lift: http://liftweb.net/

[2] Lift's Screen: https://liftweb.assembla.com/wiki/show/liftweb/Lift%27s_Screen

[3] Simple Build Tool: http://code.google.com/p/simple-build-tool/

Imprint

Publisher
Software & Support Media GmbH

Editorial Office Address
Geleitsstraße 14
60599 Frankfurt am Main
Germany
www.jaxenter.com

Editor in Chief: Sebastian Meyen
Editors: Jessica Thornsby, Claudia Fröhling, Hartmut Schlosser
Authors: Jon Pretty, Peter O'Sullivan, Vince Kenealy, Heiko Seeberger, Marcus Denison, Marius Danciu
Copy Editor: Nicole Bechtel
Creative Director: Jens Mainz
Layout: Melanie Hahn, Patricia Schwesinger

Sales Clerk:
Mark Hazell
+44 (0)20 7401 4845
markh@jaxlondon.com