Tutorial on Writing Modular Programs in Scala
Accompanying Documentation

Martin Odersky Gilles Dubochet

13 September 2006 (document revised 27/9/2006)

Abstract

Scala is a JVM-compatible programming language that combines features of both object-oriented and functional languages. It can express common programming patterns in a concise, elegant and type-safe way.

Scala's rich type system supports multiple inheritance (through *mixins*), polymorphic methods and classes (*generics*), implicit type-directed transformations (*views*) and more. Besides, Scala allows anonymous functions and pattern matching on objects.

This tutorial is an introduction to the Scala programming language. In particular, participants will discover how Scala's features can alleviate the shortcomings found in today's object-oriented languages when designing modular systems.

Introduction

This is the accompanying documentation to the Scala tutorial given at the Joint Modular Languages Conference 2006 in Oxford. It describes the "hands-on" part of the tutorial.

At the end of this tutorial, attendants should be able to take home the following:

- some principles and programming abstractions common to modular applications in Scala and
- experience in writing Scala programs by developing a *toy* project that demonstrates these principles.

This tutorial is based on the material covered by the tutorial talk. Slides are available at the same place this document is located. The Scala reference manual [3] is also a useful tool for this tutorial. The Scala API, found at `scala.epfl.ch/docu/files/api/`, can be used without moderation.

1 Meeting Scala

We provide an all-in-one package containing the source code of the tutorial's project, and an automated build script. You can download it from `scala.epfl.ch/docu/related.html`. You will need to have Apache Ant (`ant.apache.org`) available for building.

This tutorial's project is called Skeb and, in the end, should become a simple Excel-like spreadsheet application. After unpacking it on your computer, you should be able to compile it by simply typing `ant` while in its root folder. To execute Skeb's main object `skeb.application`, run

```
bin/scala skeb.application
```

To start this tutorial, have a look at the classes that are provided. Try printing something from Skeb's main object `skeb.application` by simply adding a `Console.print(...)` call to the object's constructor. Try creating a simple class yourself and create an instance from the main object. Add some members to the class and see how you can define methods inside other method's bodies. If you feel playful, confirm the body of the class is indeed its constructor by printing something in there.

Once you have come to believe that Scala is indeed nothing more than Java in disguise, we will be able to proceed to later phases of this tutorial to show you how wrong this is.

2 Pattern matching

In this section, you will write a simple interpreter for Skeb expressions using pattern matching. Expressions are defined by the case class hierarchy in the `skeb.expression` module.

Since case classes are used, it is possible to define the interpreter as an external module, without modifying the expression hierarchy at all. Contrast that with the *encapsulated* approach traditional OO languages would require.

The interpreter will be implemented as class `skeb.Evaluator` based on the following template.

```
class Evaluator {
  /** Reduces an expression to its simplest
    * constant value. */
  def eval (exp: Expression): Constant = ...
}
```

You will use a match expression to differentiate between the various kinds of expressions that must be evaluated. Take advantage of all types of patterns: de-constructor patterns, type patterns and, if necessary, guards. For `Apply` nodes, define the `+` and `-` operations only; we will change that later anyway.

Once the evaluator is working, you can try it on some expressions:

```
(new Evaluator).eval(Number(34))
(new Evaluator).eval(Apply("+",
  List(Number(11), Number(6))))
```

3 Functions

In the previous section, the interpreter dealt with operation application by explicitly matching on the string name of the operation. Unfortunately, this requires to change the interpreter method every time a new operations is added. In this section, we will take the responsibility of applying operations out of the interpreter.

Instead, class `Evaluator` will be extended by a map that, for every known operation name provides a function to evaluate it.

```
type Evaluator = List[Constant] => Constant
val operations: Map[String, Evaluator] =
  new HashMap[String, Evaluator]()
```

Once this is done do the following.

- Modify the `eval` method to look-up operations in the map, and apply the corresponding function to the parameters.
- Make sure you handle the case where the operation is not defined.
- Define some operations and add them to the map. Make sure the functions implementing them test for the number of arguments they receive, for example by using pattern matching.
- Test your system.

Since functions are objects, and since the `Evaluator` class does provide function-like behaviours ("evaluate" would be a rather natural function), we will try transforming the Evaluator object into a first class function.

To do that, simply define Evaluator to extend the correct `Function` class and define an `apply` method.

Spreadsheet cells will need to evaluate the expressions they contain. Now that we have an evaluate function, this should be easy.

- Declare an instance of an `Evaluator` function in the `Cell` class,
- and use it to calculate the value of the cell.

As of now, with cells capable of evaluating their content, the spreadsheet should start working. Go ahead and try it: run the `skeb.application` object.

4 Mixins

We will have a look again at the way operations are handled in the `Evaluator`. The current solution is more elegant than the first try — where absolutely everything was merged in the evaluation method —

but still lacks true modularity. Adding a method requires updating the Evaluator class.

In this section we will use mixins to

- separate the declaration of operations and the evaluator itself in different classes
- and show how the actual configuration can be selected at instance-creation time.

You will need to do the following.

- Create a trait `skeb.operation.Arithmetic` and move all operation definitions you create previously there.
- Notice how the `Arithmetic` trait must depend on the `Evaluator` class to compile successfully. Either defining the trait as a subclass of `Evaluator` or "require" it.
- Mixin the `Arithmetic` trait into the Evaluator class when you create an instance (that is in the `Cell` class).

Once this is done, define some more traits that add operations, such as

- a module providing additional mathematical functions (financial, algebraic, trigonometric,...),
- an encryption module for ROT-13 encoding and decoding string data,
- or whatever else you like.

Mix them into your cells: adding functionality in a purely modular fashion is easy and straightforward.

5 Higher-order Functions

In this last section, we will use all techniques seen before to improve the performances of Skeb. The naive implementation for calculating the value of a cell means that whenever its value is requested, all expressions of all cells it depends on will be re-evaluated. In order to improve that, we will buffer each cell's value so as not to recalculate it each time.

The problem is that if the value of a cell referenced (even indirectly) by the current cell changes, the buffered value will be wrong. We therefore need to inform a cell to update its value (clear its buffer) whenever any cell it depends on is updated.

To do that, we will use a publish/subscribe event-forwarding mechanism provided by the Scala GUI package. This system allows elements to receive events published by other elements they subscribed to. The `scala.gui.Publisher` class implements the necessary mechanism. The relevant part of its public and inherited interface follows.

```
abstract class Publisher extends Actor[Any] {
  /** An event requesting a subscription to this
    * publisher. */
  protected case class Subscribe(
    subscriber: Publisher) extends Event
  /** An event requesting a un-subscription to this
    * publisher. */
  protected case class Unsubscribe(
    subscriber: Publisher) extends Event
  /** The type of an event handler. */
  type Handler = PartialFunction[Any, Unit]
  /** The top level of the subscriber's event loop.
    * This is the default event loop for this
    * element. */
  object toplevel {
    def eventloop(handler: Handler): Unit
  }
  /** Installs a new handler for incomming
    * events. */
  final def eventloop(handler: Handler): Nothing
  /** Subscribes onself to the list of publishers,
    * so as to receive further events from them. */
  def subscribe(publishers: Publisher*): Unit
  /** Un-subscribes onself from the list of
    * publishers, so as not to receive further
    * events from them. */
  def unsubscribe(publishers: Publisher*): Unit
  /** Publishes the event to all subscribers. */
  def publish(event: Event): Unit
}
```

The important concept to understand in this class is that of its event loop. An event loop is a partial function — a function which domain does not cover all possible values of its input type — that will be

applied to any incoming event. If the event is in the domain of the function, it will be executed, otherwise, nothing will happen. In all cases, the partial function will be "re-loaded" and execute again for the next incoming event: the event loop is in effect an infinitely recursive function.

The `eventloop` function of the subscriber class installs the provided partial function (of type `Handle`) as the current event loop for the subscriber. Another call to `eventloop` will replace it with the new one. For implementation reason, the first event loop of the subscriber must be installed using the `toplevel` version of `eventloop`. Don't forget to subscribe to any publisher you wish to receive events from, they won't arrive otherwise.

```
val top = toplevel eventloop {
  case Scared => turnWhite
  case Sick =>
    turnGreen
    eventloop {
      case SawDoctor =>
        turnPink
        eventloop(top)
      case Scared => turnYellow
    }
}
```

The transformation you need to do in Skeb is as follows.

- Define `Cell` as an event publisher (which also makes it a subscriber).
- Modify the `value` member of the `Cell` class to lazily calculate itself. You can use a value of type `Option` to differentiate between an existing and a not-yet-calculated value.
- When the user sets a new expression for the cell, subscribe to all cells that are referenced by the expression, in order to be informed of their updates. Don't forget to unsubscribe from cells that are no longer relevant with the new expressions.
- Publish the fact that the cell's value changed to all other cells that depend on it.
- Define an event loop to handle value change events by resetting the cell's value buffer.

Conclusion

This tutorial did by no means cover all modular-friendly features that Scala offers. The following papers might be relevant. Odersky's and Zenger's [5] describes how Scala's *type members* allow removing hard links between program parts and in effect modularise them. Odersky's [2] covers much of what this tutorial discusses in two short pages. Burak's and al [1] describe how Scala's XML library can be used for easily defining XML-defined components. But in the end, you will only appreciate how module-friendly Scala is by trying it out for your own projects.

For more on general Scala programming, try [4].

References

[1] B. Emir, S. Maneth, and M. Odersky. Scalable programming abstractions for XML services. In *Dependable Systems: Software, Computing, Networks*, pages 103–126, 2006.

[2] M. Odersky. The Scala experiment – can we provide better language support for component systems? In *Proceedings of the 33rd Symposium on Principles of Programming Languages*, pages 166–167, 2006.

[3] M. Odersky. *The Scala Language Specification 2.0.* École polytechnique fédérale de Lausanne, Switzerland, March 2006.

[4] M. Odersky et al. An overview of the Scala programming language, second edition. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne, 2006.

[5] M. Odersky and M. Zenger. Scalable component abstractions. In *Proceedings of the 20th Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2005.