Getting started with EScala

Jurgen Van Ham

November 5, 2010

Contents

1 Target

The aim of this text is just to collect some useful info to get started with EScala . Not to explain all about the normal scala Also it can help when trying to find problems that might be encountered before. The small snippets of code are intended as practical examples.

It will just start as a collection of useful tips, I felt to be helpful to write some simple applications with EScala.

There is also some information from the slides of Lucas, and other sources (like remarks, etc) which helped in understanding or getting an overview of the possibilities and the future features.

2 Examples

Since examples can be very useful to understand thennddyo language and its mechanisms, there are at this moment 2 different implementations of an example of a world with plants and animals (EScala and scala with observers) and also a very simple simulation of atoms, this last indicates a need for some pattern to avoid 'loops'.

3 Declaring events

3.1 Different types

The real types are not really what a programmers use, but it is a good start to understand what the real picture is.

There are primitive and composite types. In the group of primitive types there are two diffenent types which are not so different after all. On the one hand there are the *imperative* events (declared with the `imperative` keyword) that are controlled explicit by the code. On the other hand there are the *implicit* events that decribe events that happen when executing the program.

primitive	imperative primitive
composite	declarative

3.1.1 Primitive

Althought the imperative and implicit don't seem very alike, when definining a method with a empty body and `Unit` as return type and the desired parameters (for the event) this method could be called in the code. Then a `beforeExec` of that method will result to the same as an `imperative` event.

3.1.2 Declarative

The only way to use the implicit events is via the declarative events.

3.1.3 Programmers Views

For the programmer there are after considering the ideas above, there are two different groups of events, which are declared and used in a different way. The *imperative* events are the most obvious since they are in a way similar to what the Observer pattern offers. The *declarative* events are what makes EScala a different language from the C# approach. In the set of declarative events there are also the *implicit* events that act like a bridge to AOP.

imperative triggered from the code and look similar to a method call (via the apply from Scala)

declarative they are specified in function of other events. The other events can be also not defined by the programmer eg `afterExec` or `beforeExec`

3.2 Imperative events

They are declared with the keyword `imperative` then the keyword `evt` followed by a name with an optional argument.

To trigger the events the name of the event is used similar to a method call. Since handlers in EScala have `Unit` as a return type, this 'method call' will not have a return value.

The leads to declarations like shown below:

```
1 class MyClass {
2   imperative evt somehingHappend[Unit]
3   imperative evt dropped[MyClass]
4
5   def something = { somethingHappened() }
6   def drop() : Unit = { dropped(this) }
7 }
```

3.3 Declarative Events

These are constructed from other events. In the most simple form these other events can be *imperative* events described above or a (combination of) expressions of other events.

```
1 evt somethingReallyHappened=somethingHappened
2 evt disjunctionOfEvents = movedUp || movedDown
3 evt conjunctionOfEvents = moved && changedColor
4 evt filteredEvent = someEvent && boolFunction
```

3.4 Event expressions

The expressions for a declarative can use the constructions shown here. The disjunctiona and the filtering are probably the most obvious. Transforming

an events also gets very useful with using the implicit[1] events. If e_1 and e_2 are event expressions or just events new events can be constructed in the way shown below.

$e_1 \&\& e_2$ Conjunction of events

$e_1 || e_2$ Disjunction of events

$e_1 \backslash e_2$ Difference of events

$e \&\& f$ Events filtered by a boolean function

e.map(f) Transforming the parameters of an event with a function 'f'

3.5 Implicit Events

It is also possible to use *implicit* events in an AOP style, for this there are the keywords `beforeExec` `afterExec` that can be used as an event before or after executing a method. In the future there will be an `afterSet` keyword added that refers the an event that occurs after setting a field of an object. For now there exists a `Variable[T]` class that helps to get a similar effect. This idea could probably also lead to other events when a variable is accessed in another way.

The arguments for these events are the arguments of the method in the case for the `beforeExec`, for the `afterExec` there can be and extra parameter for the return value.

This looks nice in the first place, but most of the time these arguments will need to be transformed by using `map` with a function to extract the required information.

```
1 class MyClass {
2   def aMethod(a : Int) : Float = { .. }
3
4   evt willDo[Int] =
5     beforeExec(aMethod)
6
7   evt hasDone[Float] =
8     afterExec(aMethod) map ((_:Int,r:Float) => r)
9 }
```

3.6 Observable methods

The *implicit* events are only available inside the scope of an object, the reason for this is that knowledge to use these implicit events is only available for the people who designed this class. This encapsulation also allows to change the implementation of a class without affecting 'external' uses of its implicit events.

[1] beforeExec and afterExec

It is possible to expose these implicit events by marking method with the keyword `observable`. This allows the implicit events for this methods to be used in other objects. This way the programmer has control about which methods are exposed for this.

3.7 Filtering events with a boolean function

When defining declarative events it is possible to add conditions to filter on the occurences of the event. This can be even on the arguments of a beforeExec or afterExec. This allow to limit events to those that are really interesting.

```
1 def aMethod(a :Int) : Unit = { .. }
2 evt callEven =
3     beforeExec(aMethod) &&
4     ((arg : Int) => arg%2==0)
```

3.8 Transforming arguments

For declarative events it is possible to transform the arguments via the `map` operator. This allow to have only useful information in the arguments of an event. Probably in many cases only the sender of the event is interesting. This can be done like this.

```
1 evt preHandler[MyClass]=
2    beforeExec(handle) map ((_:Any) => this)
```

3.9 Varlist

The `scala.events.VarList[T]` can be used to combine many objects of a type and react to an event sent by one of those objects. For this there is an operator `any` that allows a function to select an event from those elements.

The result is a disjunction on the selected event from a set of objects.

```
1 class OthClass {
2   imperative evt myEvent
3 }
4 class myClass {
5   val lst=new scala.events.VarList[OthClass]
6   for (i <- 1 to 10) {lst += new OthClass }
7   evt anEvent=lst.any(oc => oc.myEvent)
8 }
```

In the example above the `anEvent` will occur when the myEvent occurs in an element of the `lst`.

3.10 Variable[T]

This class helps to observe a variable until there are implicit events for this like the `afterSet` mentioned before. This variable sends a `changed` event with 2 parametes, the first is the old value and the second the new value.

Assignign a value to such variable can be done with the `:=`-operator. Since the apply method read the value just using the variable as a normal one will return the value of it.

```
1  class MyClass {
2     def changedTo(old:Int, new:Int) : Unit = {
3       println("T changed "+old+" -> "+new);
4     }
5
6     val t=new Variable[Int](5)
7     evt aChange=t.changed
8     aChange += changedTo _
9
10    t := 6
11    println("T is now "+t)
12 }
```

3.11 Inheritance and Overriding

When a subclass overrides an event of its superclass, the subclass could refer to the event of the superclass with the keyword `super` this allows to add a filter to such event.

```
1  class SuperClass {
2    var nmbr : Int =5
3    evt someEvent
4    ...
5  }
6  class SubClass extends SuperClass{
7    override evt someEvent = super.someEvent && super.nmbr%2==0
8  }
```

When not specifying an expression for a declarative event (no imperative keyword) it will be abstract similar to a method without a body. It is possible to have handler for such an event that will be refined in a subclass.

4 Handlers

A handler must have always an argument list between parenthesis, even when this is an empty list. Since when there are no parenthesis the compiler handles this method in a different way.

The signature of such a handler method must match the one from the event it will handle. This allows to read the parameters of the event via the parameters of the method that handles the event.

A handler also must have the return type `Unit`

5 Registering a handler

And event can 'register' many handlers, to add a handler to the set of handlers of an event, use the `+=` operator. In a similar way the `-=` operator can remove a registered handler again.

The handlers are scala functions, in most cases after the name of the handler an underscore is needed to make it a `partially applied function` for the underlying Scala. Althought there are other constructions possible.

This means for pratical purposes that such a partially applied function can be called with only its arguments and it already contains the object where the methods will be called from.

```
1 evt myEvent
2
3 def handler() : Unit = { ..}
4
5 myEvent += handler _
```

6 Known causes of problems

This is just a list of the problems that took me some time to get solved. It is probably the most growing part of the text. But I hope it will save time when using EScala .

6.1 Type of Handler

A handler should have parenthesis even without arguments and a return type of Unit. Failing to do so can result in a situation where the number of handlers grows, because unregistering[2] doesn't work

6.2 Not unregistering handlers

When handlers are not unregistered the number can grow which leads to a decrease in speed. Especially when an application does register and unregister a lot. An annoying side effect when handlers are not unregistered is that the objects that contain the handlers will always stay reachable via the event. This leads to a 'memory leak' that the garbage collector can't solve. Since via the

[2] event -= handler

registered handlers in the event objects could still be reachable event they is no other way anymore.

6.2.1 An easy solution, or not yet

A solution could be to unregister in some automatic way. The problem is that at this moment no possible way is known for this. A weak reference to a partial applied function would not protect this partial function, when it is a normal (strong) reference, the object to which this partial applied function belows will be protected from the garbage collection process during the time it is registered.

6.2.2 What is possible at this moment

A practical approach could be to count the number of reactions by modifying the `EventsLib.scala` before compiling it by adding a method to `EventNode[T]` that returns the size of the variable `_reactions`. This will need some interpretation, but when this numbers always increases, it is worth to look into the unregistering of events.