

Observable class implementation in EScala

Documentation of the practical lab made during the winter semester 2010/2011 at the TU Darmstadt.

Git branch : `ipl-ws1011-observ-class`

Authors :

- Alexandre Chataignon
- David Wagner

Introduction

The goal of this practical lab was to implement the use of observable class in scala.

The purpose of observable class is to define an event that will be declared for every instance of this observable class.

Example

For example, a bank need to make transactions. Before any credit on every Transaction, they have to call a method of a TransactionManager object (for example, writing the transaction in a log, verifying the transaction, ...).

With observable class, code would look like that :

```
observable class Transaction {  
  
    observable def credit() {  
        System.out.println("Credit")  
    }  
  
}  
  
class TransactionManager {  
  
    evt e_bcredit[Unit] = beforeExec(anyInstance[Transaction].credit)  
  
    e_bcredit += beforeCredit _  
  
    def beforeCredit() {  
        System.out.println("Before credit")  
    }  
  
}
```

User guide

The observable keyword

To declare a class as observable, just add the observable keyword in its declaration

```
observable class Pipo {}
```

The allInstances VarList

Once a class is observable, all its instances will be elements of a list `allInstances[ClassType]`. To use `allInstances`, you will have to import it first from the `scala.events` library.

For example :

```
import scala.events.allInstances

observable class Pipo {}
object Test {
  def main(args: Array[String]) {
    val p1 = new Pipo
    val p2 = new Pipo

    println(allInstances[Pipo])
  }
}
```

Will have for a result : `VarList(Pipo@d42d08, Pipo@1d86fd3)`

The anyInstance shortcut

As observable classes' first purpose is to use events on all the instances of the class, you can use `anyInstances[ClassType]` to apply an event to any instance of this class. Like `allInstances`, it has to be imported from the `scala.events` library.

For example :

```
import scala.events.anyInstances

observable class Pipo {
  evt e1 = ...
}

object Test {
  evt e_any[Unit] = anyInstance[Pipo].e1
  e_any += SomeMethod _

  def main(args: Array[String]) {
    val p1 = new Pipo
    val p2 = new Pipo
  }

  def SomeMethod() { ... }
}
```

With this code, every time e1 will occur on p1 or p2, `SomeMethod()` will be called.

Observable method can be used as well like :

```
import scala.events.anyInstances

observable class Pipo {
  observable def obsmethod() { ... }
}

object Test {
  evt e_any[Unit] = beforeExec(anyInstance[Pipo].obsmethod)
  e_any += SomeMethod _

  def main(args: Array[String]) {
    val p1 = new Pipo
    val p2 = new Pipo
  }

  def SomeMethod() { ... }
}
```

You notice that you have to ensure that `[Unit]` must be precised when declaring event using `anyInstance`.

Technical documentation

For the implementation of observables classes, we have defined two new phases in the compiler : `observableclass` and `allinstances`. They are placed in this order just after the `obsrefs` phase.

The `observableclass` phase

In this phase, the compiler will look for observable classes (classes with the `observable` keyword).

For each observable class found, the compiler will create an object (with `ModuleDef`) named `<nameoftheclass>$all`, placed in the same level of the class. This object herits from `scala.events.AllObject[Class]`.

Here is the code of `scala.events.AllObject` :

```
class AllObject[T] {
  var all = new VarList[T]

  def register(instance: T) {
    all += instance
  }
  def unRegister(instance: T) {
    all -= instance
  }
}
```

As you can see, `AllObject` contains basically a `VarList` named `all` and two methods to add or remove instance.

Then, once the `$all` object is created and corectly typed, the compiler will modify the constructor of the class to call the `register` method of `AllObject` on himself, so every time the class is instancied, the instance will be register in the `all` attribute of the `AllObject`.

The `allinstances` phase

In this phase, the compiler will look for all the `AllInstances` and `AnyInstances` calls to transform them.

`AllInstances` and `AnyInstances` are in fact defined in the library so the compiler passes the namer and typer phases. The code in the library is like that :

```
def allInstances[C]: VarList[C] = throw new NoSuchMethodException("this code has to be compiled with EScala")
def anyInstance[C]: C = throw new NoSuchMethodException("this code has to be compiled with EScala")
```

As you can see, these are dummy method just to pass the firsts phases but if they are not transformed, the code will throw an exception.

So this phase will transform them by looking for the `$all` object associated to the `T` type, and then :

- `allInstances[C]` is in fact `C$all.all`
- `anyInstance[C]` is in fact `C$all.all.any(...)`