Scala Traits
From Java Interfaces to Mix-Ins.

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: `cazzola@di.unimi.it`
twitter: `@w_cazzola`

Traits
Introduction

In Java a class can implement an arbitrary number of interfaces

- useful to declare that it exposes multiple abstractions and
- to implement a fictitious multiple inheritance

But . . .

- the same interface is implemented with the same code with little or none adaptation,
- part of that code could be unrelated to the main class and
- there isn't a easy mechanism to reuse it

The terms mixin or concern are often used for such focused and potentially reusable parts of an instance.

Traits
Scala Traits as Mixins!

Scala provides a complete mixin solution called trait

- classes can "mix in" traits in scala as can implement interfaces in java
- traits can be mixed in as well as the instances are created.

Traits preserve separation of concerns while allowing to compose behaviors on demand.

As a java programmer you can see traits as

- interfaces with optional implementations or
- a "constrained" form of multiple inheritance.

Traits
Observer Pattern: an Example!

```
class ButtonWithCallbacks(val label: String, val clickedCallbacks: List[() => Unit]) {
  require(clickedCallbacks != null, "Callback list can't be null!")

  def this(label: String, clickedCallback: () => Unit) =
    this(label, List(clickedCallback))

  def this(label: String) = {
    this(label, Nil)
    println("Warning: button has no click callbacks!")
  }

  def click() = {
    // logic to give the appearance of clicking a physical button ...
    clickedCallbacks.foreach(f => f())
  }
}
```

Traits
Observer Pattern: an Example! (Cont'd)

```
class Button(val label: String) {
  def click() = { /* Logic to give the appearance of clicking a button... */ }
}
```

```
trait Subject {
  type Observer = { def receiveUpdate(subject: Any) }
  private var observers = List[Observer]()
  def addObserver(observer:Observer) = observers ::= observer
  def notifyObservers = observers foreach (_.receiveUpdate(this))
}
```

```
class ButtonCountObserver {
  var count = 0
  def receiveUpdate(subject: Any) = count += 1
}
```

```
class ObservableButton(name: String) extends Button(name) with Subject {
  override def click() = {
    super.click()
    notifyObservers
  }
}
```

```
object ButtonObserverTest {
  def main(args: Array[String]) = {
    val observableButton = new ObservableButton("Okay")
    val buttonObserver = new ButtonCountObserver
    observableButton.addObserver(buttonObserver)
    for (i <- 1 to 3) observableButton.click()
    printf("The button has been clicked %d times\n", buttonObserver.count)
  }
}
```

Observer Pattern: an Example! (Cont'd)

When the mixed class is necessary just once

- the **ObservableButton** class can be omitted
- the trait can be directly mixed into the instance

```
object ButtonObserverTest {
  def main(args: Array[String]) = {
    val observableButton = new Button("Okay") with Subject {
      override def click() = {
        super.click()
        notifyObservers
      }
    }
    val buttonObserver = new ButtonCountObserver
    observableButton.addObserver(buttonObserver)
    for (i <- 1 to 3) observableButton.click()
    printf("The button has been clicked %d times\n", buttonObserver.count)
  }
}
```

```
[18:59]cazzola@surtur:~/lp/scala>scala ButtonObserverTest
The button has been clicked 3 times
```

Several traits can be stacked on the same class.

```scala
trait Clickable { def click() }
```

```scala
class Button(val label: String) extends Clickable {
  def click() = { /* Logic to give the appearance of clicking a button... */ }
}
```

```scala
trait ObservableClicks extends Clickable with Subject {
  abstract override def click() = {
    super.click()
    notifyObservers
  }
}
```

- Note the use of super! What does it refer to?
  - Does it refer to **Clickable** or **Subject**? Neither of them!
  - **Clickable** declares but doesn't define click();
    Subject doesn't have it at all.
  - It will be bound when the trait is bound.

```scala
object ButtonClickableObserverTest {
  def main(args: Array[String]) = {
    val observableButton = new Button("Okay") with ObservableClicks
    val buttonClickCountObserver = new ButtonCountObserver
    observableButton.addObserver(buttonClickCountObserver)
    for (i <- 1 to 3) observableButton.click()
    printf("The button has been clicked %d times\n", buttonClickCountObserver.count)
  }
}
```

The new trait will add

- the possibility of putting a veto on a change (a click).

```scala
trait VetoableClicks extends Clickable {
  val maxAllowed = 1 // default
  private var count = 0

  abstract override def click() = {
    if (count < maxAllowed) { count += 1; super.click() }
  }
}
```

- `super` and **`abstract`** again
- it only calls the `super.click()` method when `count` $\leq$ `maxAllowed`.

```scala
object ButtonClickableObserverVetoableTest {
  def main(args: Array[String]) = {
    val observableButton = new Button("Okay") with ObservableClicks with VetoableClicks
    val buttonClickCountObserver = new ButtonCountObserver
    observableButton.addObserver(buttonClickCountObserver)
    for (i <- 1 to 3) observableButton.click()
    printf("The button has been clicked %d times\n", buttonClickCountObserver.count)
  }
}
```

```
[18:11]cazzola@surtur:~/lp/scala>scala ButtonObserverTest
The button has been clicked 1 times
```

- method lookup proceed right to left
- what happens if we use the traits in the reverse order?

Traits

- don't support auxiliary constructors nor do they accept an argument list for the primary constructor;
- can extend classes or other traits but they can't pass arguments to them (so they can extend only classes/traits with a no argument constructor)
- are executed every time an instance is created that uses the trait.

```scala
trait T1 { println(" in T1: x = " + x); val x=1; println(" in T1: x = " + x) }
trait T2 { println(" in T2: y = " + y); val y="T2"; println(" in T2: y = " + y) }
class Base12 {
  println(" in Base12: b = " + b); val b="Base12"; println(" in Base12: b = "+b)
}
class C12 extends Base12 with T1 with T2 {
  println( " in C12: c = "+c); val c="C12"; println(" in C12: c = "+c)
}
println( "Creating C12:" ); new C12; println( "After Creating C12" )
```

```
[18:24]cazzola@surtur:~/lp/scala>scala TT.scala
Creating C12:
 in Base12: b = null
 in Base12: b = Base12
 in T1: x = 0
 in T1: x = 1
 in T2: y = null
 in T2: y = T2
 in C12: c = null
 in C12: c = C12
After Creating C12
```

References

- Martin Odersky and Matthias Zenger.
  Scalable Component Abstractions.
  In Proceedings of OOPSLA'05, pages 41–57, San Diego, CA, USA, October 2005. ACM Press.
- Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black.
  Traits: Composable Units of Behaviour.
  In Proceedings of the ECOOP'03, LNCS 2743, pages 248–274, Darmstadt, Germany, July 2003. Springer.
- Venkat Subramaniam.
  Programming Scala.
  The Pragmatic Bookshelf, June 2009.
- Dean Wampler and Alex Payne.
  Programming Scala.
  O'Reilly, September 2009.