Scala on the spotlight
(part 2)

Mount Everest North Face as seen from the path to the base camp, Tibet. Wikimedia Commons. GNU 1.2.

Jacques Noyé
Ecole des Mines de Nantes
Scala as a composition language

• Component = class or trait

• Composition via mixins

• Abstraction:
  • Parameters
  • Abstract members
  • Self types
Class composition with traits

• A unit of code reusable through inheritance

• Traits mix traits of Bracha’s mixins and Schärli et al.’ traits!
A simple trait

- Similar to a class but:
- No parameters
- Can be used for “mixin” composition

```scala
scala> trait AbsIterator[T] {
    def hasNext: Boolean
    def next: T
}
defined trait AbsIterator
```

Example from An Overview of the Scala Programming Language Tech. Report LAMP-REPORT-2006-001
Traits are not mere interfaces

```
trait AbsIterator[T] {
  def hasNext: Boolean
  def next: T
}

trait RichIterator[T] extends AbsIterator[T] {
  def foreach(f: T => Unit): Unit =
    while (hasNext) f(next)
}
```

- Traits may contain concrete methods and fields (and maintain state)
Traits composition

scala> class StringIterator(s: String) extends AbsIterator[Char] {
    private var i = 0
    def hasNext = i < s.length
    def next = { val x = s.charAt i; i = i + 1; x }
}
defined class StringIterator

scala> class Iter(s: String) extends StringIterator(s) with RichIterator[Char]
defined class Iter

scala> new Iter("foo") foreach println
f
o

• Iter has two parents: a superclass StringIterator and a mixin RichIterator
Questions

• What happens if the same parent is inherited via different paths?
• What happens if several parents define the same member?
• How to resolve supercalls?
Linearization

- The inheritance relationship forms a DAG
- *Linearization* rebuilds a total order
• Start by linearizing the “superclass”
• This is the last part of the linearization
Linearization - 2

- Linearize the mixins from left to right, excluding the parents already linearized

• Any
  ▲
  AnyRef
  ▲
  AbsIterator
  ▲
  StringIterator  RichIterator
  ▲
  Iter
• Add the root of the DAG
Stackable modification trait

trait SyncIterator[T] extends AbsIterator[T] {
  abstract override def hasNext: Boolean =
    synchronized(super.hasNext)
  abstract override def next: T =
    synchronized(super.next)
}

• Chain of supercalls (not available with multiple inheritance)
• Binding of super deferred to mixin time
• The existence of a super method has to be checked
Traits as (true) interfaces

trait Queue[T] {
    def head: T
    def tail: Queue[T]
    def append(x: T): Queue[T]
}

object Queue {
    def apply[T](xs: T*): Queue[T] =
        new QueueImpl[T](xs.toList, Nil)
    private class QueueImpl[T](
        private val leading: List[T],
        private val trailing: List[T]
    ) extends Queue[T] {
        def head: T = mirror.leading.head
        ...
    }
}
Traits as (true) interfaces

```scala
scala> val q = Queue(1)
q: Queue[Int] = Queue$QueueImpl@36e2c698

scala> val q1 = q append 2
q1: Queue[Int] = Queue$QueueImpl@56ccaef

scala> import Queue.QueueImpl
import Queue.QueueImpl

scala> new QueueImpl
<console>:11: error: class QueueImpl cannot be accessed in object Queue
   new QueueImpl
   ^
```
trait Abstract {
  type T
  def transform(x: T)
  val initial: T
  var current: T
}

class Concrete extends Abstract {
  type T = String
  def transform(x: String) = x + x
  val initial = "hi"
  var current = initial
}
Type Parameterization

- Backtrack
The need for nonvariance (imperative features)

class cell[+T](init: T) {
    private var current = init
    def get = current
    def set(x: T) { current = x }
} // won’t compile

val c1 = new Cell[String]("abc")
val c2: Cell[Any] = c1
        c2.set(1)
val s: String = c1.get
The need for abstract types

class Food
class Grass extends Food

abstract class Animal {
def eat(foo: Food)
}
class Cow extends Animal {
  override def eat(food: Grass) {}
}
A case for abstract types

scala> class Food
class Grass extends Food

abstract class Animal {
  def eat(foo: Food)
}
class Cow extends Animal {
  override def eat(food: Grass) {} 
}

<console>:7: error: class Cow needs to be abstract, since method eat in class Animal of type (Food)Unit is not defined
  class Cow extends Animal {
     ^

<console>:8: error: method eat overrides nothing
     override def eat(food: Grass) {} 
        ^
What about allowing covariant parameters?

class Food
class Grass extends Food
class Fish extends Food

abstract class Animal {
    def eat(food: Food)
}
class Cow extends Animal {
    override def eat(foo: Grass) {} // assume it compiles
}
val bessy: Animal = new Cow
bessy eat (new Fish) // ok as bessy is an Animal!!!
Using abstract types

class Food
class Grass extends Food
abstract class Animal {
    type SuitableFood <: Food // abstract type with upper bound
    def eat(food: SuitableFood)
}
class Cow extends Animal {
    type SuitableFood = Grass // concrete type
    override def eat(food: Grass) {}
}
Using abstract types

scala> class Fish extends Food
defined class Fish

scala> val bessy: Animal = new Cow
bessy: Animal = Cow@2f2379f2

scala> bessy eat (new Fish)
<console>:11: error: type mismatch;
found   : Fish
required: bessy.SuitableFood
  bessy eat (new Fish)
Using abstract types

```scala
scala> class Fish extends Food
defined class Fish

scala> val bessy: Animal = new Cow
bessy: Animal = Cow@2f2379f2

scala> bessy eat (new Fish)
<console>:11: error: type mismatch;
found   : Fish
required: bessy.SuitableFood
  bessy eat (new Fish)
  ^
```

path-dependant type
Path-dependent types

```scala
package animals

object Main {
  def main(args: Array[String]) {
  val bessy = new Cow
  bessy eat (new Grass)
  val marguerite = new Cow
  marguerite eat (new bessy.SuitableFood)
  }
}
```
Path-dependent types and inner classes

scala> class Outer {  
   class Inner  
}  
}  
defined class Outer

scala> val o1 = new Outer  
o1: Outer = Outer@2029a303

scala> val i1 = new o1.Inner  
i1: o1.Inner = Outer$Inner@200069ed

scala> val o2 = new Outer  
o2: Outer = Outer@badfba

scala> val i2 = new o2.Inner  
i2: o2.Inner = Outer$Inner@6ec4786e

scala> val l = List(i1, i2)  
l: List[Outer#Inner] = List(Outer$Inner@200069ed, Outer$Inner@6ec4786e)
class Outer {
    outer =>
        class Inner
            println(Outer.this eq outer)
    }
}
Traits and trait layers

Collaboration C1
- a role of C1
- base code for A

Collaboration C2
- a role of C2
- another role of C1

Class A1
- another role of C2

Class A2
- base code for B

Class A3
The observer pattern

```scala
abstract class SubjectObserver {
  trait Subject {
    private var observers: List[Observer] = List()
    def subscribe(obs: Observer) =
      observers = obs :: observers
    def publish =
      for (val obs <- observers) obs.notify(this)
  }
  trait Observer {
    def notify(sub: Subject): Unit
  }
}
```

Example slightly modified from An Overview of the Scala Programming Language Tech. Report LAMP-REPORT-2006-001
The observer pattern

```scala
object SensorReader extends SubjectObserver {
  abstract class Sensor extends Subject {
    val label: String
    var value: Double = 0.0
    def changeValue(v: Double) = {
      value = v
      publish
    }
  }

  class Display extends Observer {
    def notify(sub: Sensor) =
      println(sub.label + " has value " + sub.value)
  }
}
```
What does Scala say?

```
abstract class SubjectObserver {
    trait Subject {
        private var observers: List[Observer] = List()
        def subscribe(obs: Observer) =
            observers = obs :: observers
        def publish =
            for (val obs <- observers) obs.notify(this)
    }
    trait Observer {
        def notify(sub: Subject): Unit
    }
}

object SensorReader extends SubjectObserver {
    abstract class Sensor extends Subject {
        val label: String
        var value: Double = 0.0
        def changeValue(v: Double) = {
            value = v
            publish
        }
    }
    class Display extends Observer {
        def notify(sub: Sensor) =
            println(sub.label + " has value " + sub.value)
    }
}

<console>:26: error: class Display needs to be abstract, since method notify in trait Observer of type (SensorReader.Subject)Unit is not defined
    class Display extends Observer {
        ^
```
What does Scala say?

```scala
abstract class SubjectObserver {
  trait Subject {
    private var observers: List[Observer] = List()
    def subscribe(obs: Observer) =
      observers = obs :: observers
    def publish =
      for (val obs <- observers) obs.notify(this)
  }
  trait Observer {
    def notify(sub: Subject): Unit
  }
}

object SensorReader extends SubjectObserver {
  abstract class Sensor extends Subject {
    val label: String
    var value: Double = 0.0
    def changeValue(v: Double) = {
      value = v
      publish
    }
  }
  class Display extends Observer {
    def notify(sub: Sensor) =
      println(sub.label + " has value " + sub.value)
  }
}

<console>:26: error: class Display needs to be abstract,
  since method notify in trait Observer of type
  (SensorReader.Subject)Unit is not defined
  class Display extends Observer {
^
abstract class SubjectObserver {

  type S <: Subject
  type O <: Observer

  trait Subject {
    private var observers: List[O] = List()
    def subscribe(obs: O) =
      observers = obs :: observers
    def publish =
      for (val obs <- observers) obs.notify(this)
  }

  trait Observer {
    def notify(sub: S): Unit
  }
}
Fixing the problem

```scala
object SensorReader extends SubjectObserver {
  type S = Sensor
  type O = Display

  abstract class Sensor extends Subject {
    val label: String
    var value: Double = 0.0
    def changeValue(v: Double) = {
      value = v
      publish
    }
  }

  class Display extends Observer {
    def notify(sub: Sensor) = {
      println(sub.label + " has value " + sub.value)
    }
  }
}
```
What does Scala say?

```scala
abstract class SubjectObserver {
  type S <: Subject
  type O <: Observer
  trait Subject {
    private var observers: List[O] = List()
    def subscribe(obs: O) =
      observers = obs :: observers
    def publish =
      for (val obs <- observers) obs.notify(this)
  }
  trait Observer {
    def notify(sub: S): Unit
  }
}

<console>:12: error: type mismatch;
  found   : SubjectObserver.this.Subject
  required: SubjectObserver.this.S
    for (val obs <- observers) obs.notify(this)
```
Fixing the problem with a self type

```scala
abstract class SubjectObserver {
  type S <: Subject
  type O <: Observer
  trait Subject {
    this: S =>
    private var observers: List[O] = List()
    def subscribe(obs: O) =
      observers = obs :: observers
    def publish =
      for (val obs <- observers) obs.notify(this)
  }
  trait Observer {
    def notify(sub: S): Unit
  }
}
```
Fixing the problem

scala> import SensorReader._
val s1 = new Sensor { val label = "sensor1" }
val s2 = new Sensor { val label = "sensor2" }
val d1 = new Display; val d2 = new Display
s1.subscribe(d1); s1.subscribe(d2)
s2.subscribe(d1)
s1.changeValue(2); s2.changeValue(3)
sensor1 has value 2.0
sensor1 has value 2.0
sensor2 has value 3.0
Try it!

- On-line documentation
- Books
- Tools: emacs support, Eclipse plugin...