# CS 2340 Objects and Design - Scala

## Traits and Packages

Christopher Simpkins
chris.simpkins@gatech.edu

# Traits

Traits are like abstract classes except:

- Traits cannot take constructor arguments, and
- `super` calls are dynamically bound (not statically as in classes) - more later

Given this trait definition:

```
trait Philosophical {
  def philosophize() {
    println("I consume memory, therefore I am!")
  }
}
```

You can *mix-in* the trait using the `extends` keyword, just like for classes:

```
class Frog extends Philosophical {
  override def toString = "green"
}
```

Traits define types, and mixing-in a trait creates a subtype relationship

# Mixing-in Multiple Traits

When extending a class and mixing-in a trait, class must come first

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "green"
}
```

Can mix-in mutpile traits with additional `with` keywords and override methods from traits just like methods from superclasses

```
class Animal
trait HasLegs

class Frog extends Animal with Philosophical with HasLegs {
  override def toString = "green"

  override def philosophize() {
    println("It ain’t easy being "+ toString +"!")
  }
}
```

# Thin vs. Rich Interfaces

- Thin interfaces easier for library authors to maintain
- Rich intercaes easier for client code to use

A rich rectangle class without traits

```
class Point(val x: Int, val y: Int)

class Rectangle(val topLeft: Point, val bottomRight: Point) {
  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
}
```

# Extension Traits

A trait can turn a thin interface into a rich interface

- Define the basic thin interface
- Define rich extensions using the basic thin interface

For example:

```
trait Rectangular {
  def topLeft: Point
  def bottomRight: Point
  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
}
```

`topLeft` and `bottomRight` are the basic thin interface on which the rich extension methods `left, right` and `width` are based

# Using an Extension Trait

Given the following complete definition for `Rectangle`

```
class Rectangle(val topLeft: Point, val bottomRight: Point)
  extends Rectangular
```

Users can use the rich interface

```
scala> val rect = new Rectangle(new Point(1, 1), new Point(10, 10))
rect: Rectangle = Rectangle@3536fd

scala> rect.left
res2: Int = 1

scala> rect.right
res3: Int = 10
```

BTW, how does this work? In particular, where are the definitions of `topLeft` and `bottomRight` that are abstract in the `Rectangular` trait?

# The Uniform Access Principle in Action

Recall the definitions of Rectangle and rectangular:

```
trait Rectangular {
  def topLeft: Point
  def bottomRight: Point
  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
}
class Rectangle(val topLeft: Point, val bottomRight: Point)
  extends Rectangular
```

- `topLeft` and `bottomRight` are abstract *parameterless methods* in trait `Rectangular`
- `Rectangle` overrides `topLeft` and `bottomRight` with parametric fields
- Overriding methods with fields is made possible by the uniform access principle and employing the convention for defining parameterless methods

# An Enrichment Example From the Standard Library

Recall the `Rational` class from chapter 6. If you wanted to add convenient comparison operators, their implmentation would look something like this:

```
class Rational(n: Int, d: Int) {
  // ...
  def < (that: Rational) = this.numer * that.denom > that.numer *
    this.denom

  def > (that: Rational) = that < this

  def <= (that: Rational) = (this < that) || (this == that)
  def >= (that: Rational) = (this > that) || (this == that)
}
```

Other classes that supported these comparison operators would look the same. Can we factor out the boilerplate?

# The `Ordered` Trait

Here's the entire definition of the `Ordered` trait from the Scala library (minus comments):

```
trait Ordered[A] extends java.lang.Comparable[A] {

  def compare(that: A): Int

  def <  (that: A): Boolean = (this compare that) <  0
  def >  (that: A): Boolean = (this compare that) >  0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
  def compareTo(that: A): Int = compare(that)
}
```

Classes that mix-in the `Ordered` trait need only define the compare method. All of the other convenient comparisoin operations are defined in terms of `compare`

Note that `Ordered` takes a type parameter.

# The `Rational` Class with `Ordered` Mixed-in

Given the following (details elided) definition of `Rational` (note the type parameter supplied to `Ordered`):

```scala
class Rational(n: Int, d: Int) extends Ordered[Rational] {
  // ...
  def compare(that: Rational) =
    (this.numer * that.denom) - (that.numer * this.denom)
}
```

You get all the comparison operators defined in trait `Ordered`

```
scala> val half = new Rational(1, 2)
half: Rational = 1/2

scala> val third = new Rational(1, 3)
third: Rational = 1/3

scala> half < third
res5: Boolean = false

scala> half > third
res6: Boolean = true
```

# Stackable Modifications

Consider an abstract `IntQue` class:

```
abstract class IntQueue {
  def get(): Int

  def put(x: Int)
}
```

We can make a basic concrete subclass like this:

```
import scala.collection.mutable.ArrayBuffer
class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]

  def get() = buf.remove(0)

  def put(x: Int) { buf += x }
}
```

... and we can add modifications in a modular way using traits.

# Stackable `Doubling` Modification

Say we want to double the contents of the `BasicIntQue` as they are added. We can define this modificaiton to `InQue`'s as a trait:

```
trait Doubling extends IntQueue {
  abstract override def put(x: Int) { super.put(2 * x) }
}
```

- `trait Doubling extends IntQueue`, so you can only mix this trait into classes that extend `IntQue`
- The `abstract override` modifer (which can only be done in traits) on the `put` method together with the `super` call says that `Doubling` must be mixed into a class that has a concrete definition of `put` (which might itself be provided by a trait that is mixed-in *before* `Doubling` is mixed-in)

# The `Doubling` Modification in Action

We can mix-in the `Doubling` trait in a class definition

```
scala> class MyQueue extends BasicIntQueue with Doubling
defined class MyQueue

scala> val queue = new MyQueue
queue: MyQueue = MyQueue@91f017

scala> queue.put(10) scala> queue.get()
res12: Int = 20
```

Since the definition of `MyQueue` defines no new code, we don't need to define a whole new class. We can mix-n traits in calls to `new`

```
scala> val queue = new BasicIntQueue with Doubling
queue: BasicIntQueue with Doubling = $anon$1@5fa12d

scala> queue.put(10)

scala> queue.get()
res14: Int = 20
```

# Stackable Modifications

But wait, there's more! We can define additional traits and stack these modifications. Here are two more modification traits:

```
trait Incrementing extends IntQueue {
  abstract override def put(x: Int) { super.put(x + 1) }
}
trait Filtering extends IntQueue {
  abstract override def put(x: Int) { if (x >= 0) super.put(x) }
}
```

Here's how these *stackable modifications* to `BasicIntQue` work:

```
scala> val queue = (new BasicIntQueue with Incrementing with Filtering)
queue: BasicIntQueue with Incrementing with Filtering...

scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get()
res15: Int = 1

scala> queue.get()
res16: Int = 2
```

# Linearization Enables Stackable Traits

- In a class, methods in traits are executed from right to left
- `super` calls in a trait are bound dynamically based on where the trait is in the linearization - for stacked traits, `super` invokes the trait to the left
- `super` calls in a class are bound statically to the superclass, which is linearized before the `super` call is bound

The upshot: the order of trait mix-ins is important

```
scala> val queue = new BasicIntQueue with Filtering with Incrementing
queue: BasicIntQueue with Filtering with Incrementing...

scala> queue.put(-1); queue.put(0); queue.put(1) scala> queue.get()
res17: Int = 0

scala> queue.get()
res18: Int = 1

scala> queue.get()
res19: Int = 2
```

# Traits in Summary

- Like abstract classes that can't take constructor arguments
- Can contain abstract methods (like Java interfaces) and definitions, including fields and concrete methods
- Can mix-in multiple traits
- Specially defined traits with `abstract override` methods and dynamically-bound `super` calls create *stackable modifications* that can be mixed-in flexibly into classes

Traits are Scala's mechanism for *mix-in composition* that avoids many of the pitfalls of true multiple inheritance. Mixing-in traits is something between inheritance and composition. Traits create types like inheritance, but are also used for type-aware composition, as with stackable modifications.

# Packages

Two ways to define packages:

- with a package statement at the top of a source file,

```
package bobsrockets.navigation

class Navigator
```

- or using *packaging* syntax

```
package bobsrockets.navigation {
  class Navigator
}
```

Both syntaxes create a class named `bobsrockets.navigation.Navigator`
I prefer the package statement syntax, which is like Java's.

# Packages Namespaces/Scopes

You can nest packages and get the expected referencing behavior:

```
package bobsrockets {
 package navigation {
    class Navigator // ...
  }
  class Ship {
    // No need to say bobsrockets.navigation.Navigator
    val nav = new navigation.Navigator
  }
  package fleets {
    class Fleet {
      // No need to say bobsrockets.Ship
      def addShip() { new Ship }
    }
  }
}
```

Note that if you use separate files and package statements (the Java way, and the way most Scala programmers do it), you have to import parent package members - they're not automatically in scope as the are in the single-file packaging syntax example above.

# Imports

Given the following package:

```
package bobsdelights

abstract class Fruit( val name: String, val color: String)

object Fruits {
  object Apple extends Fruit("apple", "red")
  object Orange extends Fruit("orange", "orange")
  object Pear extends Fruit("pear", "yellowish")
  val menu = List(Apple, Orange, Pear)
}
```

- `import bobsdelights._` imports all members of the package into the namespace in which the import occurs (which could be anywhere - including inside a method)
- `import bobsdelights.Fruit` imports only the Fruit class
- `import Fruits.{Apple, Orange}` imports only `Apple` and `Orange` from object Fruits

# More Imports

```
package bobsdelights

abstract class Fruit( val name: String, val color: String)

object Fruits {
  object Apple extends Fruit("apple", "red")
  object Orange extends Fruit("orange", "orange")
  object Pear extends Fruit("pear", "yellowish")
  val menu = List(Apple, Orange, Pear)
}
```

- `import Fruits.{Apple => McIntosh, Orange}` imports `Apple` and `Orange` but renames `Apple` to `McIntosh` in the current namespace
- `import Fruits.{_}` or `import Fruits._` imports all members from object Fruits
- `import Fruits.{Pear => _,_}` imports all members of Fruits except Pear