

CSE 130: Spring 2012

Objects, Classes & Inheritance

Ranjit Jhala, UC San Diego

What is an Object?

What is an object?

Here's a really simple object

```
object countUp {  
  var count = 0           // Data  
  
  def next() = {          // Function  
    count += 1  
    count  
  }  
  
  def hasNext = true      // Function  
}
```

- Object = Map from Names To Properties
- Names = `count`, `next`, `hasNext`
- Properties = Data + Functions

What is an object

Just a fancy *value*

- **Object** = Map from **Names** To **Properties**
- **Properties** = Data + Functions

So, how can we USE an Object?

So, how can we USE an Object?

Here's a client function:

```
scala> def tickN(it: Any, n: Int) =  
        for (i <- 1 to n){ println(it.next()) }  
  
scala> tickN(countUp, 5)
```

What happens?

So, how can we USE an Object?

Here's a client function:

```
def tickN(it: Any, n: Int) =  
  for (i <- 1 to n){println(it.next())}
```

Type checker is unhappy.

```
<console>:9: error: value next is not a member of Any  
  println(it.next())  
           ^
```

How to DESCRIBE the Required Properties ?

How to DESCRIBE the Required Properties ?

```
<console>:9: error: value next is not a member of Any  
  println(it.next())  
           ^
```

- Duh. Cannot call tickN with Any value...
- Which properties **required** by tickN

Describing Objects with “Structural” Types

The mystery type ???

```
def tickN(it: ???, n: Int) =  
  for (i <- 1 to n){println(it.next())}
```

it must map `next` to a function

```
{ def next(): Any }
```

- Called a **Structural** Type
- Describes requirements on structure ...

Need to DESCRIBE the TYPE of an object?

```
def tickN(it: { def next(): Any}, n: Int) =  
  for (i <- 1 to n){println(it.next())}
```

Thats it!

Lets take it for a spin.

```
scala> tickN(countUp, 5)  
1  
2  
3  
4  
5
```

Recap

- Object = Names -> Properties
- Type = Names -> Types

What is an object?

Here's another really simple object

```
object countFib {  
  var a = 0           // Data  
  var b = 1           // Data  
  
  def next() = {      // Function  
    val (a_, b_) = (b, a + b)  
    a = a_  
    b = b_  
    a  
  }  
  
  def hasNext = true  // Function  
}
```

- Object = Map from Names To Properties
- Names = a, b, next, hasNext
- Properties = Data + Functions

So, how can we USE an Object?

We can use it with tickN too!

```
scala> tickN(countFib, 10)  
1  
2  
3  
5  
8  
13  
21  
34  
55  
89
```

tickN doesn't care about **extra** names and properties

So, how can we USE an Object?

tickN can be called with **any** object with suitable **next**

- A **structural** subtype of {def next():Any}

```
scala> tickN("yellowmattercustard".iterator, 5)
y
e
l
l
o
```

So, how can we USE an Object?

tickN can be called with *any* object with suitable **next**

```
scala> tickN(List("cat", "dog").iterator, 5)

cat
dog

java.util.NoSuchElementException: next on empty iterator

Whoops.
```

So, how can we USE an Object?

We can add more **required** methods to tickN

```
def tickN[A](it: {def next(): A; def hasNext: Boolean}, n: Int) =
  for (i <- 1 to n){
    if (it.hasNext) println(it.next())
  }
```

So, how can we USE an Object?

We can add more **required** methods to tickN

Better to give the new required type a name

```
type ticker = { def next(): Any; def hasNext: Boolean }
```

and then ...

```
def tickN(it: ticker, n: Int) =  
  for (i <- 1 to n){  
    if (it.hasNext) {  
      println(it.next())  
    } else {  
      println("Out of Stuff!")  
    }  
  }  
}
```

So, how can we USE an Object?

Which works quite nicely...

```
scala> tickN(List("cat", "dog").iterator, 5)  
cat  
dog  
Out of Stuff!  
Out of Stuff!  
Out of Stuff!
```

Recap

Objects

- Maps from **names** to **properties**
- Properties = Data + Functions (+ Objects)

Types

- Short descriptions of Objects
- Maps from **names** to **types**
- **Structural** Types

So thats OOP! (?)

What else do we need?

What IS a class Anyway?

What IS a class Anyway?

Seem to be doing just fine without them!

Many OO languages are **Class**-less

- Google's **Go**
- JavaScript (also type-less ...)
- Java- (only interfaces)

Why do we need Classes?

You tell me!

- ???
- ???
- ???
- ???

What is a Class?

Unified mechanism for two tasks

Specification

A **name** for **describing** contents

Implementation

A **template** for **creating** new objects

What is a Class?

Spectrum from Specification to Implementation

1. Interfaces

Only Specification of Types

2. Abstract Class

Specification + **Some** Concrete Implementations

3. Class

Specification + **All** Concrete Implementations

The Class Spectrum in Scala: Interfaces

The Class Spectrum in Scala: Interfaces

Interfaces: Only Specify Type

```
type ticker = { def next(): Any; def hasNext: Boolean }
```

Some other ways too (later.)

The Class Spectrum in Scala: Abstract Classes

The Class Spectrum in Scala: Abstract Classes

Lets make a **ScreenSaver!**

Here's the basic loop for the graphics:

```
def run(box: JPanel, points: List[Point]) {  
  ...  
  while (true) {  
    for (p <- points) {  
      p.draw(g)  
      p.move(d, delta)  
      p.collide(d)  
    }  
  }  
  ...  
}
```

Need a type `Point` describing objects that we can:

- draw, move, collide

The Class Spectrum in Scala: Abstract Classes

Need a type describing draw, move, collide

First, some essential fields...

```
abstract class Point {  
  
    // Must be filled in by (concrete) sub-classes  
    val name : String      // immutable  
    var x     : Int        // mutable  
    var y     : Int        // mutable  
  
    // Defined, but may be "overridden" by sub-classes  
    var xVel = Rand(1, 5)  
    var yVel = Rand(1, 5)  
    ...  
}
```

The Class Spectrum in Scala: Abstract Classes

Need a type describing draw, move, collide

Next, the methods ...

```
abstract class Point {  
    // Must be filled in by (concrete) sub-classes  
    def xDim : Int  
    def yDim : Int  
    def render(g: Graphics): Unit  
    def move(d: Dimension, delta: Int)  
    ...  
}
```

... these are pure specification

must be filled in by any actual `Point` object

The Class Spectrum in Scala: Abstract Classes

Need a type describing draw, move, collide

Next, we **specify** the methods ...

```
abstract class Point {  
  // Must be filled in by (concrete) sub-classes  
  def xDim : Int  
  def yDim : Int  
  def render(g: Graphics): Unit  
  def move(d: Dimension, delta: Int)  
  ...  
}
```

... which **must** be filled in an actual Point object

The Class Spectrum in Scala: Abstract Classes

Need a type describing draw, move, collide

Finally, we **implement** a few common methods ...

```
abstract class Point {  
  
  // Can be overridden by concrete Points  
  def collide(d: Dimension) {  
    hitWall(x, 0, d.width - xDim) map {(v:Int) =>  
      x      = v  
      xVel = -xVel  
    } ...  
  }  
  
  // Cannot be overridden by sub-classes  
  final def draw(g: Graphics) {  
    val c = g.getColor()    // save old color  
    render(g)               // render object  
    g.setColor(c)           // restore old color  
  }  
}
```

... which **must** be filled in an actual Point object

So, how do we create a real Point ?!

The Class Spectrum in Scala: (Concrete) Classes

The Class Spectrum in Scala: Classes

Like an `abstract` class but with **everything** defined.

```
class Dot(val name: String, x0: Int, y0: Int) extends Point {  
    ...  
}
```

`extends Point` says `Dot` is a concrete implementation of `Point`.

The Class Spectrum in Scala: Classes

Parameters define several things “for free”

```
class Dot(val name: String, x0: Int, y0: Int) extends Point {  
    // Filled in by "constructor"  
    var x      = x0  
    var y      = y0  
    def xDim   = 20  
    def yDim   = 20  
    ...  
}
```

Generated Constructor

- Initializes fields from parameters `x0`, `y0`
- Named parameters automatically become fields

The Class Spectrum in Scala: Classes

We define required methods

```
class Dot(val name: String, x0: Int, y0: Int) extends Point {  
    def render(g: Graphics) {  
        g.fillOval(x, y, xDim, yDim)  
    }  
}
```

```

    override def move(d: Dimension, delta: Int) {
      x += (delta * xVel)
      y += (delta * yVel)
    }
  }
}

```

- Remainder **inherited** from Point

The Class Spectrum in Scala: Classes

We have **also** automatically defined new **type** ...

```

class Dot(val name: String, x0: Int, y0: Int) extends Point {
  def render(g: Graphics) {
    g.fillOval(x, y, xDim, yDim)
  }

  override def move(d: Dimension, delta: Int) {
    x += (delta * xVel)
    y += (delta * yVel)
  }
}

```

... with **name** Dot

- and specified it is a **subtype** of Point
- Called **Nominal** (by-name) subtyping
- Can use Dot where Point is **expected**

The Class Spectrum in Scala: Classes

Note the auxiliary constructor **this**

- Feeling lazy about supplying x0, y0?

```

// Auxiliary constructor, invokes "real" constructor
def this(n: String) =
  this(n, Rand(0, 100), Rand(0, 100))

```

The Class Spectrum in Scala: Classes

Lets create an instance

With the default constructor ...

```
new Dot("p1", 50, 50)
```

... and with the auxiliary one

```
new Dot("p2") //Uses default constructor
```

Recap: What is a Class?

Spectrum from Specification to Implementation

1. Interfaces

Only Specification of Types

e.g. `ticker`

2. Abstract Class

Specification + **Some** Concrete Implementations

e.g. `Point`

3. Class

Specification + **All** Concrete Implementations

e.g. `Dot`

OK! Lets Run The Screensaver!

So far: Objects + Classes + (Some) Inheritance

How to get colored dots?

Inheritance

Extend Dot with a notion of color

```
class ColorDotV1(name: String, color: Color)
  extends Dot(name) {
    override def render(g: Graphics) {
      g.setColor(color) // "color" in scope from params
      super.render(g)
    }
  }
```

Note: `extends Dot(name)`

- Pithy way of calling the (old) super-class constructor
- Passing it the parameter(s) of the (new) sub-class

Inheritance Enables Reuse

Get lots of **properties** from without any work!

- Only need to override or extend bits that are different...

Lets make another shape

How about a Box ?

```
class Box(name: String, h: Int, w: Int) extends Dot(name) {
  override def yDim = w
  override def xDim = h

  override def render(g: Graphics) {
    g.fillRect(x, y, h, w)
  }
}
```

```
}  
}
```

Get lots of stuff without any work!

- Get `move` (from `Dot`) for free!
- Get `collide` (from `Point`) for free!

Lets make another shape

Lets make another shape

How about a `Box` ?

Lets check it out ...

```
val points = List( ...  
                  , new Box("p4", 20, 40)  
                  )
```

But wait a minute...

Wait a minute...

... must be wary of things you get **for free**

- Good Angel: Is a `Box` really a `Dot` ?
- Evil Angel: But we're getting so much **reuse** !!
- Good Angel: ...

Lets make (yet) another shape

How about we **color** them `Boxes`?

- How?
- Inheritance?
- Lets see...

Lets make (yet) another shape: ColorBox

Lets try extending Box ...

```
class ColorBoxV1(name: String, color: Color, h: Int, w: Int)
extends Box(name, h: Int, w: Int) {
  override def render(g: Graphics) {
    g.setColor(color)
    super.render(g)
  }
}
```

... Yikes! We're repeating ourselves!

- render is **just** like ColorDot.render
- How can we reuse the above render ?

Lets make (yet) another shape: ColorBox

Ok, lets try extending ColorDot ...

... whats the problem?

Problem

We bundled Colored-ness into ColorDot ...

... but color is **totally independent** of Shape!

How can we **free up** colored-ness from its chains?

Traits

Traits

Describe object **behaviors** independent particular classes ...

... Can be **mixed** into any class to add that behavior to the class!

Traits describe Object Behaviors Independently

```
trait Colored extends Point {  
  
    val color: Color // Will be defined by mixin-target  
  
    abstract override def render(g: Graphics) {  
        g.setColor(color)  
        super.render(g) // Call render of mixin-target  
    }  
}
```

This exactly **specifies** and **implements** colored-ness ...
... without bundling it inside a particular class!

Traits describe Object Behaviors Independently

```
trait Colored extends Point {  
  
    val color: Color // Will be defined by mixin-target  
  
    abstract override def render(g: Graphics) {  
        g.setColor(color)  
        super.render(g) // Call render of mixin-target  
    }  
}
```

Wait! Aren't traits just like Java interfaces?

Traits Can Be Mixed-Into Appropriate (Host) Classes

To use a trait, just **mixin** to an appropriate host class.
(e.g. subtype of Point)

```
class ColorDot(name: String, val color: Color)  
    extends Dot(name) with Colored  
  
class ColorBox(name: String, val color: Color, h: Int, w: Int)  
    extends Box(name, h, w) with Colored
```

Lets take the new classes for a spin!

Traits: Recap

Describe object **behaviors** independent particular classes ...

... Can be **mixed** into any class to add that behavior to the class!

Are traits different from Multiple Inheritance

Are traits different from Multiple Inheritance

Why can't I do something like this:

```
abstract class Point
class Dot extends Point {...}
class Colored extends Point {...}
```

and then

```
class ColoredDot extends Dot, Colored {...}
```

Wouldn't this solve the problem of reusing Color and Dot ?

- Problem 1: Which parent do I pick method from?
- Problem 2: Need to compose **render** from **both**

Are traits different from Multiple Inheritance

The cunning thing about traits:

```
trait Colored extends Point {

  val color: Color // Will be defined by mixin-target

  abstract override def render(g: Graphics) {
    g.setColor(color)
    super.render(g) // Call render of mixin-target
  }
}
```

- **stacks** the new behavior
- **on top of** existing behavior! (`super.render`)

- independent of the host class!
- So, color-a-box, color-a-dot, color-anything!

Another Trait: Bouncing

Another Trait: Bouncing

Lets add a notion of **gravity** to the objects...

```
trait Bouncing extends Point {
  ...
  abstract override def move(d: Dimension, delta: Int) {
    yVel += delta
    shootUp()           // if ROTFL
    super.move(d, delta) // preserve
  }
}
```

Another Trait: Bouncing

Lets add a **gravity** behavior to the objects...

- independent of shape, color, etc.

```
trait Bouncing extends Point {
  ...
  abstract override def move(d: Dimension, delta: Int) {
    yVel += delta
    shootUp()           // if ROTFL
    super.move(d, delta) // preserve
  }
}
```

Note how new behavior stacked on top of old!

- Reuse old (horizontal) motion via `super.move`

Another Trait: Bouncing

We can add bouncing behavior to **any** Point object

```
class BouncingColorDot(name: String, val color: Color)
  extends Dot(name) with Colored with Bouncing

class BouncingColorBox(name: String, val color: Color, h: Int, w: Int)
  extends Box(name, h, w) with Colored with Bouncing
```

Or even directly **retro-actively** to the instance object

```
new Dot("p2") with Bouncing
```

Moral of the story?

Moral of the story

Inheritance enables reuse

BUT

Class(ical) inheritance **locks** reusable behaviors into unrelated classes (e.g. ColorDotV1)

In fact ...

Our code can be cleaned up more...

Moral of the story

Inheritance enables reuse

BUT

Class(ical) inheritance **locks** reusable behaviors into unrelated classes (e.g. ColorDotV1)

SO

Keep a sharp eye out for independent behaviors,
and **set them free** using **traits**.