18 Traits and Multiple Inheritance

Recall our code for CPoint from last time. We defined `CPoint` to be a subtype of `Point`, and we used inheritance to have the implementation classes of `CPoint` inherit code form the implementation classes of `Point`. I also advocated using the explicit-delegation implementation of `CPoint` as a way to understand inheritance.

Let's look at the code for `CPoint` again:

```
object CPoint {

  def cartesian(x:Double,y:Double,c:Color):CPoint =
    new CartesianCPoint(x,y,c)

  def polar(r:Double,theta:Double,c:Color):CPoint =
    if (r<0)
      throw new Error("r negative")
    else
      new PolarCPoint(r,theta,c)

  private def reconstructCart (p:Point,c:Color):CPoint =
      new CartesianCPoint(p.xCoord(),p.yCoord(),c)

  private def reconstructPolar (p:Point,c:Color):CPoint =
      new PolarCPoint(p.distanceFromOrigin(),
                      p.angleWithXAxis(),c)


  private class CartesianCPoint (xpos:Double, ypos:Double, c:Color)
           extends CartesianPoint(xpos,ypos) with CPoint {

    def distance (q:CPoint):Double =
      super[CartesianPoint].distance(q)

    override def move (dx:Double,dy:Double):CPoint =
      reconstructCart(super[CartesianPoint].move(dx,dy),c)

    def add (q:CPoint):CPoint =
```

```
    reconstructCart(super[CartesianPoint].add(q),q.color())

  override def rotate (t:Double):CPoint =
    reconstructCart(super[CartesianPoint].rotate(t),c)

  def isEqual (q:CPoint):Boolean =
    super[CartesianPoint].isEqual(q) && (c==q.color())

  def color ():Color = c

  def updateColor (nc:Color):CPoint =
    new CartesianCPoint(xpos,ypos,nc)

  // BRIDGE METHODS

  override def isEqual (q:Point):Boolean = q match {
    case cq:CPoint => isEqual(cq)
    case _ => false
  }

  // CANONICAL METHODS

  override def toString ():String =
    "cartesian(" + xpos + "," + ypos + "," + c + ")"

  override def equals (other : Any):Boolean =
    other match {
      case that : CPoint => this.isEqual(that)
      case _ => false
    }

  override def hashCode ():Int =
    41 * (
      41 * (
        41 + xpos.hashCode()
      ) + ypos.hashCode()
    ) + c.hashCode()
}


private class PolarCPoint (r:Double, theta:Double, c:Color)
          extends PolarPoint(r,theta) with CPoint {
```

```
def distance (q:CPoint):Double =
  super[PolarPoint].distance(q)

override def move (dx:Double,dy:Double):CPoint =
  reconstructCart(super[PolarPoint].move(dx,dy),c)

def add (q:CPoint):CPoint =
  reconstructCart(super[PolarPoint].add(q),q.color())

override def rotate (angle:Double):CPoint =
  reconstructPolar(super[PolarPoint].rotate(angle),c)

def isEqual (q:CPoint):Boolean = {
  super[PolarPoint].isEqual(q) && c==q.color()
}

def color ():Color = c

def updateColor (nc:Color):CPoint =
  new PolarCPoint(r,theta,nc)

// BRIDGE METHODS

override def isEqual (q:Point):Boolean = q match {
  case cq:CPoint => isEqual(cq)
  case _ => false
}

// CANONICAL METHODS

override def toString ():String =
  "polar(" + r + "," + theta + "," + c + ")"

override def equals (other : Any):Boolean =
  other match {
    case that : CPoint => this.isEqual(that)
    case _ => false
  }

override def hashCode ():Int =
  41 * (
```

```
        41 * (
          41 + r.hashCode()
        ) + theta.hashCode()
      ) + c.hashCode()
  }
}


trait CPoint extends Point {

  def xCoord ():Double
  def yCoord ():Double
  def angleWithXAxis ():Double
  def distanceFromOrigin ():Double
  def distance (q:CPoint):Double
  def move (dx:Double,dy:Double):CPoint
  def add (q:CPoint):CPoint
  def rotate (theta:Double):CPoint
  def isEqual (q:CPoint):Boolean
  def isOrigin ():Boolean

  def color ():Color
  def updateColor (nc:Color):CPoint

  // bridge methods
  def distance (q:Point):Double
  def add (q:Point):Point
  def isEqual (q:Point):Boolean
}
```

18.1 Multiple Inheritance

Note that there is some code in common between `CartesianCPoint` and `PolarCPoint`. Could we use inheritance, in the way we used it a few lectures ago, to avoid duplicating the code in common — for instance, the `equals()` method. (We could do it also for the other methods, but because they use `super`, it's a bit trickier...) As before, we can define a `CommonCP` class that is a supertype of both `CartesianCPoint` and `PolarCPoint` and that is a subtype of `CPoint`, a form of what I called an innocuous use of inheritance before . Here's the code:

```
object CPoint {

```

```
def cartesian(x:Double,y:Double,c:Color):CPoint =
  new CartesianCPoint(x,y,c)

def polar(r:Double,theta:Double,c:Color):CPoint =
  if (r<0)
    throw new Error("r negative")
  else
    new PolarCPoint(r,theta,c)

private def reconstructCart (p:Point,c:Color):CPoint =
    new CartesianCPoint(p.xCoord(),p.yCoord(),c)

private def reconstructPolar (p:Point,c:Color):CPoint =
    new PolarCPoint(p.distanceFromOrigin(),
                    p.angleWithXAxis(),c)


private trait CommonCP extends CPoint {

  override def equals (other : Any):Boolean =
    other match {
      case that : CPoint => this.isEqual(that)
      case _ => false
    }
}

private class CartesianCPoint (xpos:Double, ypos:Double, c:Color)
          extends CartesianPoint(xpos,ypos) with CommonCP {

  def distance (q:CPoint):Double =
    super[CartesianPoint].distance(q)

  override def move (dx:Double,dy:Double):CPoint =
    reconstructCart(super[CartesianPoint].move(dx,dy),c)

  def add (q:CPoint):CPoint =
    reconstructCart(super[CartesianPoint].add(q),q.color())

  override def rotate (t:Double):CPoint =
    reconstructCart(super[CartesianPoint].rotate(t),c)

  def isEqual (q:CPoint):Boolean =
```

```
    super[CartesianPoint].isEqual(q) && (c==q.color())

  def color ():Color = c

  def updateColor (nc:Color):CPoint =
    new CartesianCPoint(xpos,ypos,nc)

  // BRIDGE METHODS

  override def isEqual (q:Point):Boolean = q match {
    case cq:CPoint => isEqual(cq)
    case _ => false
  }

  // CANONICAL METHODS

  override def toString ():String =
    "cartesian(" + xpos + "," + ypos + "," + c + ")"

  override def hashCode ():Int =
    41 * (
      41 * (
        41 + xpos.hashCode()
      ) + ypos.hashCode()
    ) + c.hashCode()
}


private class PolarCPoint (r:Double, theta:Double, c:Color)
          extends PolarPoint(r,theta) with CommonCP {

  def distance (q:CPoint):Double =
    super[PolarPoint].distance(q)

  override def move (dx:Double,dy:Double):CPoint =
    reconstructCart(super[PolarPoint].move(dx,dy),c)

  def add (q:CPoint):CPoint =
    reconstructCart(super[PolarPoint].add(q),q.color())

  override def rotate (angle:Double):CPoint =
    reconstructPolar(super[PolarPoint].rotate(angle),c)
```

```
    def isEqual (q:CPoint):Boolean = {
      super[PolarPoint].isEqual(q) && c==q.color()
    }

    def color ():Color = c

    def updateColor (nc:Color):CPoint =
      new PolarCPoint(r,theta,nc)

    // BRIDGE METHODS

    override def isEqual (q:Point):Boolean = q match {
      case cq:CPoint => isEqual(cq)
      case _ => false
    }

    // CANONICAL METHODS

    override def toString ():String =
      "polar(" + r + "," + theta + "," + c + ")"

    override def hashCode ():Int =
      41 * (
        41 * (
          41 + r.hashCode()
        ) + theta.hashCode()
      ) + c.hashCode()
  }
}


trait CPoint extends Point {

  def xCoord ():Double
  def yCoord ():Double
  def angleWithXAxis ():Double
  def distanceFromOrigin ():Double
  def distance (q:CPoint):Double
  def move (dx:Double,dy:Double):CPoint
  def add (q:CPoint):CPoint
  def rotate (theta:Double):CPoint
```

```
  def isEqual (q:CPoint):Boolean
  def isOrigin ():Boolean

  def color ():Color
  def updateColor (nc:Color):CPoint

  // bridge methods
  def distance (q:Point):Double
  def add (q:Point):Point
  def isEqual (q:Point):Boolean
}
```

Note that we have to make `CommonCP` a trait, because `CartesianCPoint` can only extend one class, and it's `CartesianPoint`, since we really want to delegate (implicitly) most method calls to `CartesianPoint`. Similarly for `PolarCPoint`.

But the above code shows that we can inherit from more than one class, as long as all but one of the classes we inherit from is a trait. Scala lets us do a form of *multiple inheritance*.

The idea behind multiple inheritance is exactly what the term seems to imply: inheriting methods from more than one supertype. We saw that having more than one supertype is not a problem, as far as subtyping is concerned. For inheritance, though, things are not so clean. Subtyping allows you to reuse code on the client side, while inheritance allows you to reuse code on the implementation side. Inheritance is an implementation technique that lets us reuse implementation code. In both Java and Scala, when we *extend* a class, we not only define a subtype, but also allow inheritance from that class.

One problem with multiple inheritance—inheriting from multiple superclasses—is that it is inherently ambiguous.

Consider the following classes `B`, `C`, `D`, defined in some hypothetical extension of Scala with multiple inheritance via multiple `extends`:

```
class B {
  def foo ():Int = 1
}

class C A {
  def foo ():Int = 2
}

class D extends B,C
```

Both classes `B` and `C` define a method `foo()`, each returning different values. Suppose we have `d` an object of class `D`, and suppose that we invoke `d.foo()`. What do we get as a result? Because `D` does not define `foo()`, we must look for it in its superclasses, from which it inherits. But it inherits one `foo` method returning 1 from `B`, and one `foo()` method returning 2 from `C`. Which one do we pick? There must be a way to choose one or the other. We could either say: the superclasses are "searched" in the order they were defined when `D` was created, or maybe we need a way to say exactly which method to call, such as `foo[A]()` or `foo[C]()`. Different languages that support multiple inheritance have made different choices. The most natural is to simply look in the classes in the order in which they occur in the `extends` declaration. But that's a bit fragile, since a small change (flipping the order of superclasses) can make a big difference, and the small change can be hard to track down. But there is a different way of looking at the problem that makes looking at the order in which classes appear in the declaration of `D` less attractive. Suppose we have the more complex hierarchy with classes `A`, `B`, `C`, `D`:

```
class A {
  def foo ():Int = 1
}

class B extends A

class C extends A {
  def foo ():Int = 2
}

class D extends B,C
```

Now, class `B` inherits a general method `foo()` from `A`, while `C` overwrites `A`'s `foo()` method with its own (presumably more specialized) method `foo()` Again, suppose we have `d` an object of class `D`, and suppose that we invoke `d.foo()`. What do we get as a result? As before, because `D` does not define `foo`, we must look for it in its superclasses from which it inherits. But it inherits one (general) `foo()` method returning 1 from `B` (which got it from `A`), and one more specialized `foo()` method returning 2 from `C`. It may make sense to assume that the more specialized method is more relevant to `D`, since it is available as an alternative to the more general `foo()` method obtained from `A`. So one possibility is to resolve ambiguities by inheriting the most specialized form of the method possible.

What I mean to suggest, here, is that resolving ambiguities in the presence of multiple inheritance is not intuitively clear cut. It gets controversial very fast. C++ is a classic language with multiple inheritance, with a complex set of rules for ambiguity resolution.

There is another problem with multiple inheritance, one slightly more subtle: super resolution—that is, how to resolve the `super` keyword. Here's an illustrative example. Suppose that we have a diamond pattern where `File` is a class for writing to a file (with a `write()`) method, while `EncryptedFile` is an extension of `File` that can do encryption and overrides `write()` so that it encrypts the data before calling `super.write()` to do the actual writing, and LoggedFile is an extension of File that can do logging, overriding write() so that it logs that the file has been written to in some external file before calling super.write() to do the actual writing. If EncryptedLogged multiply-inherits from both EncryptedFile and LoggedFile, then we have to choose, in write(), whether to call the method inherited from EncryptedFile or from LoggedFile. Neither of those will give us an encrypted write that also logs. And we can't call both methods, because then the file will be written twice, certainly not the outcome we expect. Multiple inheritance is tricky.

Because multiple inheritance is tricky, Java and many other languages have taken a different approach: forbid multiple inheritance altogether, so that you cannot inherit from more than one superclass. Then there is no problem with determining where to look for methods if they are not in the current class: look in the (unique) superclass. This is why the extends keyword in Java, which expresses inheritance, can only be used to subclass a single superclass. If you want to subclass other classes as well, those have to be interfaces. Interfaces are not a problem for inheritance, because there is nothing there to inherit: interfaces contain no code. Therefore, when you have a non-tree hierarchy, you need to first identify which subclassing relations between the class you want to rely on inheritance. This choice will force other classes to be interfaces.

Scala weakens this restriction a little bit, without providing full multiple inheritance. The idea is that traits can also contain code, but all traits are "linearized", so that Scala finds a linear order over all traits in an inheritance hierarchy and resolves super calls using that linear order. This provides a nice solution to the super resolution problem above. Of course, there is a trade-off, and it's that the algorithm for linearization is fairly complex, and I will point you to the Scala documentation for more details.

To simplify using traits, I recommend that you only use traits to inherit methods that are not already inherited from another class, and to avoid using super calls in traits. Doing so will free you from having to think about linearization.

18.2 Traits for Augmenting Interfaces

There are a few ways in which traits are used in Scala that satisfy the recommendations I make in the last section.

The idea is to use traits to "augment" an interface, that is, to turn a thin interface—an interface without a lot of functions—into a rich interface—one with a lot of function.

Consider the streams interface from a few lectures back:

```
trait Stream[A] {

  def hasElement ():Boolean
  def head ():A
  def tail ():Stream[A]
```

```
}
```

That's a thin interface, as it provides only three functions. Now, we can enrich this interface by adding several functions to the traits that are all expressible in terms of those three functions. A class that implements the `Stream` trait only need to provide the three functions above to automatically inherit all these other functions derivable from them. These derived operations include printing the stream, or zipping the stream with another stream, and generally include most of the stream gadgets we saw in Lecture 15.

Here is the augmented `Stream` trait:

```
trait Stream[A] {

  def hasElement ():Boolean
  def head ():A
  def tail ():Stream[A]

  // Derived operations

  def print ():Unit = {
    if (hasElement()) {
      println("  " + head());
      tail().print()
    }
  }

  def printN (n:Int):Unit =
    if (hasElement())
      if (n > 0) {
        println("  " + head())
        tail().printN(n-1)
      }
      else
        println("  ...")

  def sequence (st:Stream[A]):Stream[A] =
    new Sequence(this,st)

  private class Sequence (st1:Stream[A], st2:Stream[A]) extends Stream[A]
    {
    def hasElement ():Boolean = {
      st1.hasElement() || st2.hasElement()
    }
```

```
  def head ():A =
    if (st1.hasElement())
      st1.head()
    else
      st2.head()
  def tail ():Stream[A] =
    if (st1.hasElement())
      new Sequence(st1.tail(),st2)
    else
      st2.tail()
}

def zip[B] (st2:Stream[B]):Stream[Pair[A,B]] =
  new Zip[B](this,st2)

private
class Zip[B] (st1:Stream[A],st2:Stream[B]) extends Stream[Pair[A,B]] {
  def hasElement ():Boolean = {
    st1.hasElement() && st2.hasElement()
  }
  def head ():Pair[A,B] = Pair.create(st1.head(),st2.head())
  def tail ():Stream[Pair[A,B]] = st1.tail().zip(st2.tail())
}

def map[B] (f:(A)=>B):Stream[B] =
  new Map[B](this,f)

private
class Map[B] (st:Stream[A], f:(A)=>B) extends Stream[B] {
  def hasElement ():Boolean = st.hasElement()
  def head ():B = f(st.head())
  def tail ():Stream[B] = st.tail().map(f)
}

def filter (p:(A)=>Boolean):Stream[A] =
  new Filter(this,p)

private
class Filter (st:Stream[A], p:(A)=>Boolean) extends Stream[A] {
  def findNext (s:Stream[A]):Stream[A] =
    if (s.hasElement()) {
      if (p(s.head()))
```

```
          s
        else
          findNext(s.tail())
      } else
        s
    def hasElement ():Boolean = findNext(st).hasElement()
    def head ():A = findNext(st).head()
    def tail ():Stream[A] = new Filter(findNext(st).tail(),p)
  }
}
```

To pick another example, here is a trait that can be used to extend any class implementing a `<=` operation defined as a partial order, and that provides derived operations `<`, `>`, `>=`, `<>`, and `isEqual()`:

```
trait Ordered[T <: Ordered[T]] { this:T =>     // self type!

  // needs to be defined

  def <= (v:T):Boolean

  // all of these definitions are derived from the one above

  def >= (v:T):Boolean = v <= this
  def === (v:T):Boolean = (v <= this) && (v >= this)
  def isEqual (v:T):Boolean = this===v
  def <> (v:T):Boolean = !(v===this)
  def < (v:T):Boolean = (this <= v) && (this <> v)
  def > (v:T):Boolean = (v < this)
}
```

The trait is parameterized on the type of values that the current value can be compared against. As we shall see below, when we extend a class `Foo` with `Ordered`, we usually have `Foo` implement `Ordered[Foo]`. That parameter `T` is a subtype of `Ordered[T]` is required for us to call the various derived operations on values of type `T` in the interface. Moreover, the trait uses something called a *self type*: an annotation `this:T =>` inside the trait to indicate that the instance on which the methods are invoked (i.e., the `this` instance) should be considered to have type `T`. (Otherwise the system uses that `this` has type `Ordered`.) This is important because otherwise something like `def >= (v:T):Boolean = v <= this` cannot type check: `v` has type `T`, but `this` has type `Ordered`, and `<=` expects an argument of type `T`, not `Ordered`. Of course, the Scala type system checks that whenever you a class implements `Ordered[T]`, then the class is a subtype of `T` to ensure that the self type is consistent with the use of the trait. *Exercise: figure out how that checks guarantees that no unsafe programs* *are accepted.*

As an example, here is a simple class that implements rational numbers:

```
class Rational (n:Int,d:Int) extends Ordered[Rational] {

  private def sgn (a:Int):Int = if (a<0) -1 else if (a>0) 1 else 0
  private def abs (a:Int):Int = if (a<0) -a else a
  private def gcd (a:Int,b:Int):Int = if (b==0) a else gcd(b,a % b)

  private val g:Int = gcd(abs(n),abs(d))
  private val numer:Int = (sgn(n*d)) * (abs(n)/g)
  private val denom:Int = (abs(d)/g)

  def unary_- ():Rational = new Rational(-n,d)

  def + (r:Rational):Rational =
    new Rational(numer*r.denom + r.numer*denom,denom*r.denom)

  def * (r:Rational):Rational =
    new Rational(numer*r.numer,denom*r.denom)

  def <= (r:Rational):Boolean = (numer*r.denom <= r.numer*denom)


  // CANONICAL METHODS

  override def equals (other:Any):Boolean = other match {
    case that:Rational => (this<=that && that<=this)
    case _ => false
  }

  override def hashCode ():Int =
    41 * (
      41 + numer.hashCode()
    ) + denom.hashCode()

  override def toString ():String = numer + "/" + denom
}
```

The class implements `<=`, and the derived operations inherited from the trait give the rest of the comparison operations.

Here is a different example, the implementation of the List ADT can be extended by

`Ordered`, defining `<=` to mean that the list is a prefix of another list—yielding that two lists are equal when they are a prefix of each other.

```
object List {

  def empty[A <: Ordered[A]] ():List[A] = new ListEmpty[A]()

  def singleton[B <: Ordered[B]] (i:B):List[B] = new ListSingleton[B](i)

  def merge[C <: Ordered[C]] (L:List[C], M:List[C]):List[C] =
     new ListMerge[C](L,M)


  private abstract class Common[T <: Ordered[T]] extends List[T] {

    def <= (M:List[T]):Boolean =
      if (M.isEmpty())
        false
      else if (first()==M.first())
        rest() <= M.rest()
      else
        false
  }


  private class ListEmpty[T <: Ordered[T]] () extends Common[T] {

    def isEmpty ():Boolean = true
    def first ():T = throw new RuntimeException("empty().first()")
    def rest ():List[T] = throw new RuntimeException("empty().rest()")

    def length ():Int = 0

    override def hashCode ():Int = 41

    override def toString ():String = ""
  }



  private class ListSingleton[U <: Ordered[U]] (i:U) extends Common[U] {

    def isEmpty ():Boolean = false
```

```
    def first ():U = i
    def rest ():List[U] = List.empty()

    def length ():Int = 1

    override def hashCode ():Int = 41 + i.hashCode()

    override def toString ():String = " " + i.toString()
  }



  private class ListMerge[V <: Ordered[V]] (L:List[V], M:List[V])
         extends Common[V] {

    def isEmpty ():Boolean =
      (L.isEmpty() && M.isEmpty())

    def first ():V =
      if (L.isEmpty())
        M.first()
      else
        L.first()

    def rest ():List[V] =
      if (L.isEmpty())
        M.rest()
      else
        List.merge(L.rest(),M)

    def length ():Int = L.length() + M.length()

    override def hashCode ():Int =
      41 * (
        41 + L.hashCode()
      ) + M.hashCode()

    override def toString ():String = L.toString() + M.toString()
  }
}
```

```
abstract class List[T <: Ordered[T]] extends Ordered[List[T]] {

  def isEmpty ():Boolean
  def first ():T
  def rest ():List[T]
  def length ():Int
  def <= (M:List[T]):Boolean
}
```