15 Multiple Inheritance and Traits

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

```
object List {

  def empty ():List = new ListEmpty()
  def cons (v:Int,l:List):List = new ListCons(v,l)
}

abstract class List {

  def isEmpty ():Boolean
  def first ():Int
  def rest ():List
  def find (f:Int):Boolean
  def append (M:List):List
}

class ListEmpty () extends List {

  def isEmpty ():Boolean = true
  def first ():Int = throw new RuntimeException("Empty.first()")
  def rest ():List = throw new RuntimeException("Empty.rest()")
  def find (f:Int):Boolean = false
  def append (M:List):List = M

  override def toString ():String = ""
}

class ListCons (v:Int,r:List) extends List {

  def isEmpty ():Boolean = false
  def first ():Int = v
```

```
  def rest ():List = r
  def find (f:Int):Boolean =
    (f==v) || r.find(f)
  def append (M:List):List = List.cons(v,r.append(M))

  override def toString ():String = v + " " + r
}
```

```
object MList {

  def empty ():MList = new MListEmpty()
  def cons (v:Int,l:MList):MList = new MListCons(v,l)
}

trait MList extends List {

  def isEmpty ():Boolean
  def first ():Int
  def rest ():MList
  def find (f:Int):Boolean
  def append (M:MList):MList

  def length ():Int
  def sum ():Int
}

class MListEmpty () extends ListEmpty with MList {

  def downcast (l:List):MList = l match {
    case l2:MList => l2
    case _ => throw new RuntimeException("Illegal downcast to MList")
  }

  override def rest ():MList = downcast(super.rest())

  override def append (M:MList):MList = M

  val accLength:Int = 0
  val accSum:Int = 0

  def length ():Int = accLength
  def sum ():Int = accSum
```

```
}

class MListCons (v:Int,r:MList) extends ListCons(v,r) with MList {

  def downcast (l:List):MList = l match {
    case l2:MList => l2
    case _ => throw new RuntimeException("Illegal downcast to MList")
  }
  override def rest ():MList = downcast(super.rest())

  override def append (M:MList):MList = MList.cons(v,r.append(M))

  val accLength:Int = 1+r.length()
  val accSum:Int = v+r.sum()

  def length ():Int = accLength
  def sum ():Int = accSum
}
```

15.1 Multiple Inheritance

Note that there is some code in common between `MListEmpty` and `MListCons`. Could we use inheritance, in the way we used it a few lectures ago, to avoid duplicating the code in common — for instance, the `downcast()` method, as well as the `length()` and `sum()` methods. As before, we can define a `Common` class that is a supertype of both `MListEmpty` and `MListCons` and that is a subtype of `MList`, a form of what I called an "innocuous" use of inheritance. Here's the code:

```
object MList {

  def empty ():MList = new MListEmpty()
  def cons (v:Int,l:MList):MList = new MListCons(v,l)
}

trait MList extends List {

  def isEmpty ():Boolean
  def first ():Int
  def rest ():MList
  def find (f:Int):Boolean
  def append (M:MList):MList
```

```
  def length ():Int
  def sum ():Int
}

trait Common extends MList {

  // needed because length() and sum() refer to these
  // field -- they're just declared here, not defined
  // it's just a promise that subtypes of Common will
  // define them

  protected val accLength:Int
  protected val accSum:Int

  protected def downcast (l:List):MList = l match {
    case l2:MList => l2
    case _ => throw new RuntimeException("Illegal downcast to MList")
  }

  def length ():Int = accLength
  def sum ():Int = accSum
}

class MListEmpty () extends ListEmpty with Common {

  override def rest ():MList = downcast(super.rest())

  override def append (M:MList):MList = M

  val accLength:Int = 0
  val accSum:Int = 0
}

class MListCons (v:Int,r:MList) extends ListCons(v,r) with Common {

  override def rest ():MList = downcast(super.rest())

  override def append (M:MList):MList = MList.cons(v,r.append(M))

  val accLength:Int = 1+r.length()
  val accSum:Int = v+r.sum()
}
```

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

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.

Why is this the problem? Because multiple inheritance—inheriting from multiple superclasses—is inherently ambiguous. Consider the following classes `A`, `B`, `C`, `D`, defined in some hypothetical extension of Scala with multiple inheritance.

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

class B extends A

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

class D extends B,C
```

Class `B` inherits method `foo` from `A`, while `C` overwrites `A`'s `foo` method with its own. Now, 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. There is also the problem of whether we look up in the hierarchy before looking right in the hierarchy. (We did not find `foo` in `B`; do we look for it in `A` before looking for it in `C`, or the other way around?) The point is, it becomes complicated very fast.

It turns out that many approaches are not particularly useful in some situations: 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 the superclass's `write()` method 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 its superclass's `write()` method 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. This example is an example of the *diamond problem* for multiple inheritance.

Since 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 they do not allow inheritance: interfaces contain no code, so there is no code to inherit. 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 and falling prey to the diamond problem. 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. Now, 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.

15.2 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 12.

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 tbe 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 ():List = new ListEmpty()

  def cons (v:Int,l:List):List = new ListCons(v,l)
}

abstract class List extends Ordered[List] {

  def isEmpty ():Boolean
  def first ():Int
  def rest ():List
  def find (f:Int):Boolean

  def append (M:List):List
}

class ListEmpty () extends List {

  def isEmpty ():Boolean = true
  def first ():Int = throw new RuntimeException("Empty.first()")
  def rest ():List = throw new RuntimeException("Empty.rest()")
  def find (f:Int):Boolean = false
  def append (M:List):List = M

  def <= (M:List):Boolean = true

  override def toString ():String = ""
}

class ListCons (v:Int,r:List) extends List {

  def isEmpty ():Boolean = false
  def first ():Int = v
  def rest ():List = r
  def find (f:Int):Boolean =
```

```
    (f==v) || r.find(f)
  def append (M:List):List = List.cons(v,r.append(M))

  def <= (M:List):Boolean =
    if (M.isEmpty())
      false
    else if (v==M.first())
      r <= M.rest()
    else
      false

  override def toString ():String = v + " " + r
}
```