14 Parameterized Classes

Last time, we advocated subtyping from multiple types as a way to get more reuse out of client code (the `Colored` example.

We continue on this track here, by showing how we can use parameterized classes in combination with traits to enable even more code reuse.

We've been playing with an ADT for integer lists for a while now. Suppose we wanted to define lists of Booleans, or lists of doubles. It seems silly to define a completely new ADT for those, because aside from the types describing the contents of the lists, the implementation of those lists is exactly the same. (Try it.) So can we somehow reuse our existing ADT and implementation for lists and extend it to work with an arbitrary element type? The answer is yes, and we can do it just like we did for the parameterized trait `Colored[T]` — we just need to define a parameterized List[T] ADT:

```
CREATORS
  empty :      () -> List[T]
  singleton :  T -> List[T]
  merge :      (List[T], List[T]) -> List[T]

ACCESSORS
  isEmpty :    () -> Boolean
  first :      () -> T
  rest :       () -> List[T]
```

with the usual specification – the only difference is in the signature, which is parameterized by a type `T`. (I've kept the operations simple for now. In particular, I've left out `isEqual()` – equality in the presence of parameterization is somewhat subtle, so I'll come back to it later.)

Here is the code obtained from the usual Interpreter Design Pattern, except taking parameterization into account:

```
object List {

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

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

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


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

    override def hashCode ():Int = 41
    override def toString ():String = ""
  }


  private class ListSingleton[U] (i:U) extends List[U] {
    def isEmpty ():Boolean = false
    def first ():U = i
    def rest ():List[U] = List.empty()

    override def hashCode ():Int = 41 + i.hashCode()
    override def toString ():String = " " + i.toString()
  }


  private class ListMerge[V] (L:List[V], M:List[V]) extends List[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)
    override def hashCode ():Int =
      41 * (
        41 + L.hashCode()
      ) + M.hashCode()
    override def toString ():String = L.toString() + M.toString()
  }
}
```

```
abstract class List[T] {
  def isEmpty ():Boolean
  def first ():T
  def rest ():List[T]
}
```

A couple of things to note. First, when defining the abstract class `List`, we specify a *type parameter*. That type parameter stands for a type that we give when we use the `List` abstract type, and that type parameter can occur in the body of the abstract class to refer to that type.

Second, the implementation classes are also parameterized, which makes sense because they need to be subtypes of `List[T]`, which is itself parameterized.

Third, I've made the creators into parameterized functions – that's because modules cannot be parameterized in Scala — think about it, would that even make sense? — and so there has to be a single module `List` containing the creators, and so it is easier to just make it so that there is a single version of each creator that works for all lists, that is, works no matter the element type of the list. Thus, when we invoke `singleton()`, we get to specify the kind of list we are working on: `List.singleton[Int](1)` will create the integer list containing the single element 1, while `List.singleton[Boolean](true)` will create the Boolean list containing the single element `true`.

As I said before, we often do not usually need to explicitly specify a type when calling parameterized functions — the type checker will often figure out the right type for us. Accordingly, from now on, I will sometimes leave out the type argument on calls to parameterized function, with the understanding that I can always toss them in if the type checker gets confused.

Here are some sample lists:

```
scala> val L1 = List.merge[Int](List.singleton[Int](1),List.singleton[Int](2))
L1: List[Int] =  1 2

scala> val L2 = List.merge[Boolean](List.singleton[Boolean](true),
                                    List.singleton[Boolean](true))
L2: List[Boolean] =  true true
```

Once we have `List` as a parameterized class, we can now write function that work on lists irrespectively of the type of the elements of those lists. In fact, the general rule is that once you have parameterized classes (or parameterized traits), one often needs parameterized functions to get the most code reuse out of the parameterization.

For instance, the length of a list is independent of the type of its elements. While we could write a length function that works with integer lists, with signature:

```
def length (L:List[Int]):Int
```

and one that works with Boolean lists, with signature:

```
def length (L:List[Boolean]):Int
```

It is much easier to simply write a parameterized function that takes the type of the list to work on as a parameter:

```
scala> def length[T] (L:List[T]):Int =
         if (L.isEmpty())
           0
         else
           1 + length[T](L.rest())
length: [T](L: List[T])Int

scala> length[Int](L1)
res0: Int = 2

scala> length[Boolean](L2)
res1: Int = 2

scala> length(L1)
res2: Int = 2
```

We see, in the last line, that indeed we do not need to specify the type parameter when it can be deduced from the context. (Here, `L1` is a `List[Int]`.)

Not every function on lists can be made into a parameterized list though. For instance, the function `sum` to add the elements of a list is naturally restricted to lists whose elements can be added:

```
scala> def sum (L:List[Int]):Int =
         if (L.isEmpty())
           0
         else
           L.first()+sum(L.rest())
sum: (L: List[Int])Int

scala> sum(L1)
res3: Int = 3

scala> sum(L2)
<console>:8: error: type mismatch;
 found   : List[Boolean]
 required: List[Int]
```

```
sum(L2)
    ^
```

Finally, sometimes we need to restrict the types to which a parameterized function can be applied. Recall the `Point/CPoint/Rect/CRect` example from last lecture.

Suppose we wanted to define a function that takes a list of `CPoint`s and returns the list of colors of those points:

```
scala> val p1 = CPoint.cartesian(1.0,2.0,Color.red())
p1: CPoint = cpoint(1.0,2.0,red)

scala> val p2 = CPoint.cartesian(3.0,4.0,Color.blue())
p2: CPoint = cpoint(3.0,4.0,blue)

scala> val L1 = List.merge(List.singleton(p1),List.singleton(p2))
L1: List[CPoint] =  cpoint(1.0,2.0,red) cpoint(3.0,4.0,blue)

scala> def colorsP (L:List[CPoint]):List[Color] =
         if (L.isEmpty())
           List.empty[Color]()
         else
           List.merge(List.singleton(L.first().color()),colorsP(L.rest()))
colorsP: (L: List[CPoint])List[Color]

scala> colorsP(L1)
res0: List[Color] =  red blue
```

We can define a version of that function to work `CRect`s:

```
scala> val r1 = CRect.create(p1,p2,Color.red())
r1: CRect = crect(cpoint(1.0,2.0,red),cpoint(3.0,4.0,blue),red)

scala> val r2 = CRect.create(p1,p2,Color.blue())
r2: CRect = crect(cpoint(1.0,2.0,red),cpoint(3.0,4.0,blue),blue)

scala> val L2 = List.merge(List.singleton(r1),List.singleton(r2))
L2: List[CRect] =  crect(cpoint(1.0,2.0,red),cpoint(3.0,4.0,blue),red)
                     crect(cpoint(1.0,2.0,red),cpoint(3.0,4.0,blue),blue)

scala> def colorsR (L:List[CRect]):List[Color] =
```

```
         if (L.isEmpty())
           List.empty[Color]()
         else
           List.merge(List.singleton(L.first().color()),colorsR(L.rest()))
colorsR: (L: List[CRect])List[Color]

scala> colorsR(L2)
res1: List[Color] =  red blue
```

Clearly, `colorsP()` and `colorsR()` have the same definition. So the question arises, can we define a single parameterized function that works for both lists of `CPoint`s and lists of `CRect`s? This doesn't work:

```
scala> def colors[A] (L:List[A]):List[Color] =
         if (L.isEmpty())
           List.empty[Color]()
         else
           List.merge(List.singleton(L.first().color()),colors(L.rest()))
<console>:6: error: value color is not a member of type parameter A
```

The reason is subtle. If you look at the static types, the static type of `L.first()` is `A` (that is, whatever type we instantiate the type parameter as when we call the function), and if we have a value of type `A`, we don't know whether there is a method `color()` to call on it, since we know nothing about `A` at this point. So the type checking fails.

What we need is a way to restrict the parameterization to ensure that whatever type we instantiate `A` as, that type has a method `color()` on it. We can achieve this by saying that the types we allows as instantiations of type parameter `A` are types that must be a subtype of `Colored[A]`, since this guarantees they have a method `color()` defined:

```
scala> def colors[A <: Colored[A]] (L:List[A]):List[Color] =
         if (L.isEmpty())
           List.empty[Color]()
         else
           List.merge(List.singleton(L.first().color()),colors(L.rest()))
colors: [A <: Colored[A]](L: List[A])List[Color]

scala> colors(L1)
res2: List[Color] =  red blue

scala> colors(L2)
res3: List[Color] =  red blue
```

(Note: we restricted type parameter `A` to be a subtype of `Colored[A]`, because we needed to choose some instantiation of the parameter of the parameterized trait `Colored`. But really, we don't care what the parameter to `Colored` is here – since the parameter to `Colored` controls the return type of `updateColor()`, a method we do not use in `colors`. So really, we want `A` to be a subtype of `Colored` for *any* instantiation of the parameter of `Colored`. How could we express that? Using another parameter! Here is the most general type we can give `colors`:

```
scala> def colors[A,B <: Colored[A]] (L:List[B]):List[Color] =
         if (L.isEmpty())
           List.empty[Color]()
         else
           List.merge(List.singleton(L.first().color()),
                      colors[A,B](L.rest()))
colors: [A,B <: Colored[A]](L: List[B])List[Color]
```

Chew on that.)

Here's another example of a parameterized ADT, binary trees.

Here is the BinTree[T] ADT:

```
  CREATORS
    empty :         () -> BinTree[T]
    node :          (T,BinTree[T],BinTree[T]) -> BinTree[T]

  OPERATIONS
    isEmpty :       () -> Boolean
    root :          () -> T
    left :          () -> BinTree[T]
    right :         () -> BinTree[T]
    size :          () -> Int
```

with specification

$$\texttt{empty().isEmpty()} = \mathit{true}$$
$$\texttt{node(}v\texttt{,}l\texttt{,}r\texttt{).isEmpty()} = \mathit{false}$$
$$\texttt{node(}v\texttt{,}l\texttt{,}r\texttt{).root()} = v$$
$$\texttt{node(}v\texttt{,}l\texttt{,}r\texttt{).left()} = l$$
$$\texttt{node(}v\texttt{,}l\texttt{,}r\texttt{).right()} = r$$
$$\texttt{empty().size()} = 0$$
$$\texttt{node(}v\texttt{,}l\texttt{,}r\texttt{).size()} = 1 + l\texttt{.size()} + r\texttt{.size()}$$

Here is the code obtained from the Interpreter Design Pattern:

```
object BinTree {

  def empty[T] ():BinTree[T] = new Empty[T]()

  def node[T] (n:T, l:BinTree[T], r:BinTree[T]):BinTree[T] =
    new Node[T](n,l,r)


  private class Empty[T] extends BinTree[T] {

    def isEmpty ():Boolean = true
    def root ():T =
      throw new RuntimeException("BinTree.empty().root()")
    def left ():BinTree[T] =
      throw new RuntimeException("BinTree.empty().left()")
    def right ():BinTree[T] =
      throw new RuntimeException("BinTree.empty().right()")
    def size ():Int = 0

    // canonical methods?
    override def toString ():String = "-"
  }


  private class Node[T] (n:T, l:BinTree[T], r:BinTree[T])
                                                  extends BinTree[T] {

    def isEmpty ():Boolean = false
    def root ():T = n
    def left ():BinTree[T] = l
    def right ():BinTree[T] = r
    def size ():Int = 1 + l.size() + r.size()

    // canonical methods?
    override def toString ():String = n + "[" + l + "," + r + "]"
  }
}

abstract class BinTree[T] {

  def isEmpty ():Boolean
```

```
  def root ():T
  def left ():BinTree[T]
  def right ():BinTree[T]
  def size ():Int
}
```

Just as in the `List[T]` case, the parameterization carries over to the implementation classes, and the creators are parameterized functions.