Object-Oriented Design	**Lecture 9**
CS 3500 Fall 2010 (Pucella)	**Friday, Oct 15, 2010**

9 Implementing Subtyping

Last time, we saw the kind of code reuse we can obtain when we have a subtyping relation between two types. But we didn't say that much about how to get subtyping. In fact, the only subtyping we've seen is in the Specification Design Pattern, here we defined the concrete classes describing the representation of data corresponding to each of the creators as subtypes of an abstract class defining the signature. How about other kind of subtypings between ADTs? Note that for the time being we will look at subtyping between abstract classes only. It makes a lot of things much simpler.

There are three main reasons for introducing subtyping between types D and C:

(1) D may be a generalization of C with extra operations defined. For instance, a `CPoint` is a `Point` with extra operations to deal with the fact that `CPoint`s have a color.

(2) D may be a restricted form of C with additional properties coming from the restriction, but no additional operations. For instance, a nonempty list `NEList` is a `List` with the additional property that it is never empty.

(3) D may bear some relationship to C that makes it a suitable candidate for subtyping in the context of the application at hand. This is much more *ad hoc*, in the sense that it really depends on the application. For example, lists could be declared to be a subtype of trees, since a list can be seen as a degenerate tree in which every node only has a right subtree (which is the rest of the list, recursively). It rarely is useful to think of lists as trees, but the point is we could, and some applications may benefit from it.

We won't have much to say about (3), but (1) and (2) are common enough that it pays to see how to set-up those kind of subtyping relations.

9.1 Subtyping by Generalization

Consider our standard ADTs Point, simplified somewhat so that it only has one creator to simplify our examples

```
CREATORS
  cartesian : (Double, Double) -> Point
```

```
OPERATIONS
  xCoord :              () -> Double
  yCoord :              () -> Double
  move :                (Double, Double) -> Point
  rotate :              (Double) -> Point

  add :                 (Point) -> Point
  isEqual :             (Point) -> Boolean
```

and CPoint:

```
CREATORS
  cartesian : (Double, Double, Color) -> CPoint

OPERATIONS
  xCoord :              () -> Double
  yCoord :              () -> Double
  color :               () -> Color
  updateColor :         (Color) -> CPoint
  move :                (Double, Double) -> CPoint
  rotate :              (Double) -> CPoint

  add :                 (CPoint) -> CPoint
  isEqual :             (CPoint) -> Boolean
```

with the usual specification.[1]

We want to set things up so that `CPoint` is a subtype of `Point`. Let's get a version going for the ADT *without* methods `add()` and `isEqual()` first. We'll see that they cause a few problems.

Ideally, we would like it to be the case that we could call the following code with either `Points` or `CPoints`. That will be our test to make sure that our subtyping enables us to get reuse.

```
def negatePoint (p:Point):Point =
  p.move(-2*p.xCoord(),-2*p.yCoord())

def addPoint (p:Point, q:Point):Point =
  p.move(q.xCoord(),q.yCoord())
```

[1] For `add()` in `CPoint`, assume that the color of the result of `p.add(q)` is the color of `q`.

```
  def rotateAroundPoint (p:Point, angle:Double, center:Point):Point =
    addPoint(center,addPoint(p,negatePoint(center)).rotate(angle))
```

Let's first define an implementation for ADT POINT using the Specification Design Pattern, as usual — remember, we do not have `add()` and `isEqual()` in the abstract class, so while we do have those methods implemented in the implementation class for cartesian points, those methods are not accessible. (Why?)

```
object Point {

  def cartesian (x:Double,y:Double):Point =
    new CarPoint(x,y)

  private class CarPoint (first:Double, second:Double) extends Point {
    def xCoord ():Double =
      first

    def yCoord ():Double =
      second

    def move (dx:Double,dy:Double):Point =
      cartesian(first+dx,second+dy)

    def rotate (theta:Double):Point =
      new CarPoint(first * math.cos(theta) - second * math.sin(theta),
                   first * math.sin(theta) + second * math.cos(theta))

    def add (p:Point):Point =
      move(p.xCoord(),p.yCoord())

    def isEqual (q:Point):Boolean =
      (first==q.xCoord() && second==q.yCoord())

    // CANONICAL

    override def toString ():String =
      "cart(" + first + "," + second + ")"

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

```
    override def hashCode ():Int =
      41 * (
        41 + first.hashCode()
      ) + second.hashCode()
  }
}


abstract class Point {

  def xCoord ():Double
  def yCoord ():Double

  def move (dx:Double,dy:Double):Point
  def rotate (t:Double):Point
}
```

Now, onto an implementation for ADT CPoint. Again, we implement it using the Specification Design Pattern. The implementation is completely straightforward. And it turns out that we can specify that a `CPoint` is a subtype of `Point` by simply stating that the abstract class `CPoint` *extends* `Point`. Note that subtyping relationship is between the abstract class representing the signatures here. There is no relationship between the implementation of those signatures. (Draw a diagram of the subtyping relationship, if it helps you.)

```
object CPoint {

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

  private class CarCPoint (first:Double, second:Double,
                           col: Color) extends CPoint {

    def color ():Color = col

    def updateColor (c:Color):CPoint =
      new CarCPoint(first,second,c)

    def xCoord ():Double =
      first

    def yCoord ():Double =
      second
```

```
    def move (dx:Double,dy:Double):CPoint =
      new CarCPoint(first+dx, second+dy,col)

    def add (cp:CPoint):CPoint =
        new CarCPoint(first+cp.xCoord(),second+cp.yCoord(),cp.color())

    def rotate (theta:Double):CPoint =
      new CarCPoint(first * math.cos(theta) - second * math.sin(theta),
                    first * math.sin(theta) + second * math.cos(theta),
                    col)

    def isEqual (q:CPoint):Boolean =
      (first==q.xCoord() && second==q.yCoord() && col==q.color())

    // CANONICAL

    override def toString ():String =
      "cart(" + first + "," + second + "," + col + ")"

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

    override def hashCode ():Int =
      41 * (
        41 * (
          41 + first.hashCode()
        ) + second.hashCode()
      ) + col.hashCode()
  }
}


abstract class CPoint extends Point {

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

  def xCoord ():Double
```

```
  def yCoord ():Double

  def move (dx:Double,dy:Double):CPoint
  def rotate (t:Double):CPoint
}
```

And it's that simple. Let's make sure we can call our `rotateAroundPoint()` function with `Point`s:

```
val p : Point = Point.cartesian(1,1)
val q : Point = Point.cartesian(1,2)
val r : Point = rotateAroundPoint(q,math.Pi/2,p)
println("Result = " + r)
```

Running this yields:

```
Result = cart(0.0,1.0)
```

And we can reuse the same `rotateAroundPoint()` function to work with `CPoint`s as well:

```
val p : Point = CPoint.cartesian(1,1,Color.blue())
val q : Point = CPoint.cartesian(1,2,Color.red())
val r : Point = TestCore.rotateAroundPoint(q,math.Pi/2,p)
println("Result = " + r)
```

yielding

```
Result = cart(0.0,1.0,blue)
```

What about those two operations we haven't dealt with yet, `add()` and `isEqual()`? They're already implemented in both implementation classes `CarPoint` and `CarCPoint`. It's just a matter of revealing them through the abstract classes. It's no problem to add them to the `Point` abstract class:

```
object Point {

  def cartesian (x:Double,y:Double):Point =
    new CarPoint(x,y)

  private class CarPoint (first:Double, second:Double) extends Point {
    def xCoord ():Double =
      first
```

```
    def yCoord ():Double =
      second

    def move (dx:Double,dy:Double):Point =
      cartesian(first+dx,second+dy)

    def rotate (theta:Double):Point =
      new CarPoint(first * math.cos(theta) - second * math.sin(theta),
                   first * math.sin(theta) + second * math.cos(theta))

    def add (p:Point):Point =
      move(p.xCoord(),p.yCoord())

    def isEqual (q:Point):Boolean =
      (first==q.xCoord() && second==q.yCoord())

    // CANONICAL

    override def toString ():String =
      "cart(" + first + "," + second + ")"

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

    override def hashCode ():Int =
      41 * (
        41 + first.hashCode()
      ) + second.hashCode()
  }
}


abstract class Point {

  def xCoord ():Double
  def yCoord ():Double

  def move (dx:Double,dy:Double):Point
  def rotate (t:Double):Point
```

```
  def add (p:Point):Point
  def isEqual (p:Point):Boolean
}
```

The problems occur when we try to do the same thing with the `CPoint` abstract class. If we simply update it to be:

```
abstract class CPoint extends Point {

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

  def xCoord ():Double
  def yCoord ():Double

  def move (dx:Double,dy:Double):CPoint
  def rotate (t:Double):CPoint

  def add (p:CPoint):CPoint
  def isEqual (p:CPoint):Boolean
}
```

then the Scala type checker complains horribly:

```
CPoint.scala:8: error: class CarCPoint needs to be abstract, since:
method isEqual in class CPoint of type (p: Point)Boolean is not defined
method add in class CPoint of type (p: Point)Point is not defined
  private class CarCPoint (first:Double, second:Double,
                ^
one error found
```

Basically, the type checker is complaining that we're missing methods `add()` and `isEqual()` that can work on `Points` in our `CPoint` class. The type checker complaining means that there's a possibility of our code being unsafe. So what's the problem?

The problem is subtle. Can we come up with code that, were the code above to be accepted by the type checker, would be unsafe — that is, cause a method-not-found error?

Here's the offending code:

```
val p:Point = CPoint.cartesian(2,4,Color.yellow())
val q:Point = Point.cartesian(0,100)
val r:Point = p.add(q)
```

What's going on here? Note that *if we assume that the type checker accepted our `CPoint` implementation and made `CPoint` a subtype of `Point`*, then the static types all agree (after the type checker adds the upcast in the first binding to treat the `CPoint` as a `Point`). But what happens during execution? `p` is bound to a `CPoint` (so that the dynamic type of `p` is `CPoint`) and `q` is bound to a `Point`. The call `p.add(q)` requires the system to look at the appropriate `add()` method in the *dynamic type* of `p`, so in `CPoint`. The `add()` method in `CPoint` expects a value of static type `CPoint`, and we're giving it `q`, a value of static type `Point`. We would need a downcast for this to work, and the type checker never introduces downcasts for us, so this fails to type check. And indeed, were we to execute this, the `add()` method in `CPoint` would try to access the `color()` method on `q`, and since `q` is a `Point`, the method `color()` would not be found. So the type checker prevented this error from happening.

Great. But then how do we get `CPoint` to be a subtype of `Point`, since we still intuitively believe that they should be subtypes? The solution is in the example above. We need to provide methods `add()` (and `isEqual()`) in `CPoint` that can work with `Point`s, on top of those we already have that can work with `CPoint`s.

The easiest way to do this is simply to add a new method declaration to the `CPoint` abstract class, telling Scala that there is an additional `add()` method (and `isEqual()` method) of the appropriate type, and implement those two new methods in `CarCPoint`:

```
object CPoint {

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

  private class CarCPoint (first:Double, second:Double,
                           col: Color) extends CPoint {

    def color ():Color = col

    def updateColor (c:Color):CPoint =
      new CarCPoint(first,second,c)

    def xCoord ():Double =
      first

    def yCoord ():Double =
      second

    def move (dx:Double,dy:Double):CPoint =
      new CarCPoint(first+dx, second+dy,col)

```

```
    def add (cp:CPoint):CPoint =
      new CarCPoint(first+cp.xCoord(),second+cp.yCoord(),cp.color())

    def add (p:Point):Point =
      p match {
        case cp:CPoint => add(cp)
        case _ => Point.cartesian(first+p.xCoord(),second+p.yCoord())
      }

    def rotate (theta:Double):CPoint =
      new CarCPoint(first * math.cos(theta) - second * math.sin(theta),
                    first * math.sin(theta) + second * math.cos(theta),
                    col)

    def isEqual (q:CPoint):Boolean =
      (first==q.xCoord() && second==q.yCoord() && col==q.color())

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

    // CANONICAL

    override def toString ():String =
      "cart(" + first + "," + second + "," + col + ")"

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

    override def hashCode ():Int =
      41 * (
        41 * (
          41 + first.hashCode()
        ) + second.hashCode()
      ) + col.hashCode()
  }
}
```

```
abstract class CPoint extends Point {

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

  def xCoord ():Double
  def yCoord ():Double

  def move (dx:Double,dy:Double):CPoint
  def rotate (t:Double):CPoint

  def add (p:CPoint):CPoint
  def isEqual (p:CPoint):Boolean

  // needed for subtyping
  def add (p:Point):Point
  def isEqual (p:Point):Boolean
}
```

A couple of things to notice: first, there are now multiple `add()` and `isEqual()` methods available in `CPoint` (and in `CarCPoint`). This is called *overloading*, and it is allowed, *as long as the methods take arguments of different types*.

How does Scala resolve the overloading? That is, given something like `cp.add(q)`, where `cp` is a value with dynamic type `CPoint`, how does it decide which of the two `add()` methods available in `CPoint` to call? It decides based on the *static type of the argument*. So if `q` is of static type `Point`, then Scala will call the `add(p:Point)` method in `CPoint`. If `q` is of static type `CPoint`, then it will call the `add(cp:CPoint)` method in `CPoint`. This may be counterintuitive (I certainly find it counterintuitive myself) but that's the way it is.

The second thing to notice is that in the implementation of `add()` that takes a `Point` as an argument, we do a dynamic check to see if the value we took as an argument really has dynamic type `CPoint` — if so, we may as well call the `add()` method that can deal with `CPoints`. If not, then we add the points as though they were `Points`. A similar deal occurs with `isEqual()` — if the argument really has dynamic type `CPoint`, we call the `isEqual()` method that can deal with `CPoints`, and if not, then we return false, because points and colored points should not be equal.

This kind of problem will occur every time you try to define a subtype D for a type C whose signature contains methods that expect arguments of type C. In D, those methods will generally take values of type D, and you will need to define "bridge methods" in D for those methods that can also take values of type C. Unfortunately, there is in general no principled way to devise those bridge methods. You'll need to think them through from scratch, depending on the ADT and what it might mean to perform the operation at hand on values of different types.

With the (corrected) code above, we can try a different version of our test, since now we have an `add()` we can call:

```
def negatePoint (p:Point):Point =
  p.move(-2*p.xCoord(),-2*p.yCoord())

def rotateAroundPoint (p:Point, angle:Double, center:Point):Point =
  p.add(negatePoint(center)).rotate(angle).add(center)
```

Running the examples above with this version of `rotateAroundPoint()` gives exactly the same results, as expected.

9.2 Subtyping by Restriction

Subtyping by restricting an ADT is less frequent, but still sometimes useful. Let's illustrate it with non-empty lists as a subtype of lists.

First, recall the List ADT:

```
CREATORS
  empty : () -> List
  cons :  (Int, List) -> List

OPERATIONS
  isEmpty :   () -> Boolean
  first :     () -> Int
  rest :      () -> List
  length :    () -> Int
  append :    (List) -> List
  find :      (Int) -> Boolean
  isEqual :   (List) -> Boolean
```

with the usual specification:

$$\texttt{empty().isEmpty()} = \textit{true}$$

$$\texttt{cons(}n\texttt{,}L\texttt{).isEmpty()} = \textit{false}$$

$$\texttt{cons(}n\texttt{,}L\texttt{).first()} = n$$

$$\texttt{cons(}n\texttt{,}L\texttt{).rest()} = L$$

$$\texttt{empty().length()} = 0$$

$\texttt{cons}(n,L)\texttt{.length()} = 1 + \texttt{L.length()}$

$\texttt{empty().append}(M) = M$

$\texttt{cons}(n,L)\texttt{.append}(M) = \texttt{cons}(n,L\texttt{.append}(M))$

$\texttt{empty().find}(f) = \texttt{false}$

$$\texttt{cons}(n,L)\texttt{.find}(f) = \begin{cases} \textit{true} & \text{if } n = f \\ \textit{true} & \text{if } L\texttt{.find}(f) = \textit{true} \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\texttt{empty().isEqual}(M) = \begin{cases} \textit{true} & \text{if } M\texttt{.isEmpty()} = \textit{true} \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\texttt{cons}(n,L)\texttt{.isEqual}(M) = \begin{cases} \textit{false} & \text{if } M\texttt{.isEmpty()} = \textit{true} \\ \textit{true} & \text{if } n = M\texttt{.first()} \\ & \quad \text{and } L\texttt{.isEqual}(M\texttt{.rest()}) = \textit{true} \end{cases}$$

The implementation, you'll recall, is completely straightforward using the Specification Design Pattern:

```
object List {

  def empty ():List = new ListEmpty()

  def cons (n:Int, L:List):List = new ListCons(n,L)


  // EMPTY LIST REPRESENTATION
  //
  private 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 length ():Int = 0

    def append (M:List):List = M

    def find (f:Int):Boolean = false
```

```
  def isEqual (M:List):Boolean = M.isEmpty()

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

  override def hashCode ():Int = 41

  override def toString ():String = ""
}


// CONS LIST REPRESENTATION
//
private class ListCons (n:Int, L:List) extends List {

  def isEmpty ():Boolean = false

  def first ():Int = n

  def rest ():List = L

  def length ():Int = 1 + L.length()

  def append (M:List):List = List.cons(n,L.append(M))

  def find (f:Int):Boolean = { (f == n) || L.find(f) }

  def isEqual (M:List):Boolean =
    (!(M.isEmpty()) && n==M.first() && L.isEqual(M.rest()))
    override def equals (other : Any) : Boolean =
      other match {
        case that:List => this.isEqual(that)
        case _ => false
      }

  override def hashCode ():Int =
    41 * (
      41 + n.hashCode()
```

```
    ) + L.hashCode()

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


abstract class List {

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

Here are a few test functions to compute the average of the elements of a list:

```
def sum (x:List):Int =
  if (x.isEmpty())
    0
  else
    x.first() + sum(x.rest())

def average (x:List):Int = {
  if (x.isEmpty())
    throw new IllegalArgumentException("Average of empty list")
  else
    (sum(x) / x.length())
}
```

We can try this out on a sample list:

```
val L1:List = List.cons(33,List.cons(66,List.cons(99,List.empty())))
println("Sum = " + sum(L1))
println("Average = " + average(L1))
```

which yields:

```
Sum = 198
Average = 66
```

Now, consider the following variant defining non-empty lists, the NEList ADT:

```
CREATORS
  singleton : (Int) -> NEList
  cons :      (Int, List) -> NEList

OPERATIONS
  isEmpty :     () -> Boolean
  isSingleton : () -> Boolean
  first :       () -> Int
  rest :        () -> List
  length :      () -> Int
  append :      (List) -> NEList
  find :        (Int) -> Boolean
  isEqual :     (List) -> Boolean
```

with the expected specification:

$$\texttt{singleton(}n\texttt{).isEmpty()} = \mathit{false}$$
$$\texttt{cons(}n\texttt{,}L\texttt{).isEmpty()} = \mathit{false}$$
$$\texttt{singleton(}n\texttt{).isSingleton()} = \mathit{true}$$
$$\texttt{cons(}n\texttt{,}L\texttt{).isSingleton()} = \begin{cases} \mathit{true} & \text{if } L\texttt{.isEmpty()} = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}$$
$$\texttt{singleton(}n\texttt{).first()} = n$$
$$\texttt{cons(}n\texttt{,}L\texttt{).first()} = n$$
$$\texttt{singleton(}n\texttt{).rest()} = L$$
$$\texttt{cons(}n\texttt{,}L\texttt{).rest()} = L$$
$$\texttt{singleton(}n\texttt{).length()} = 1$$
$$\texttt{cons(}n\texttt{,}L\texttt{).length()} = 1 + \texttt{L.length()}$$
$$\texttt{singleton(}n\texttt{).append(}M\texttt{)} = \texttt{cons(}n\texttt{,}M\texttt{)}$$
$$\texttt{cons(}n\texttt{,}L\texttt{).append(}M\texttt{)} = \texttt{cons(}n\texttt{,}L\texttt{.append(}M\texttt{))}$$
$$\texttt{singleton(}n\texttt{).find(}f\texttt{)} = \begin{cases} \mathit{true} & \text{if } n = f \\ \mathit{false} & \text{otherwise} \end{cases}$$
$$\texttt{cons(}n\texttt{,}L\texttt{).find(}f\texttt{)} = \begin{cases} \mathit{true} & \text{if } n = f \\ \mathit{true} & \text{if } L\texttt{.find(}f\texttt{)} = \mathit{true} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\texttt{singleton}(n)\texttt{.isEqual}(M) = \begin{cases} \textit{false} & \text{if } M\texttt{.isEmpty()} = \textit{true} \\ \textit{true} & \text{if } M\texttt{.first()} = n \\ & \text{and } M\texttt{.rest().isEmpty()} = \textit{true} \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\texttt{cons}(n,L)\texttt{.isEqual}(M) = \begin{cases} \textit{false} & \text{if } M\texttt{.isEmpty()} = \textit{true} \\ \textit{true} & \text{if } n = M\texttt{.first()} \\ & \text{and } L\texttt{.isEqual}(M\texttt{.rest()}) = \textit{true} \end{cases}$$

Again, this is straightforward to implement using the Specification Design Pattern, without knowing anything about the implementation of lists. Note that we do not need "bridge methods" because we cleverly (!) defined the `isEqual()` and `append()` methods in `NEList` to expect a `List` as argument, and not an `NEList` – in both cases because the operation makes perfect sense when given a non-empty list.

```
object NEList {

  def singleton (n:Int):NEList = new Singleton(n)

  def cons (n:Int, L:List):NEList = new Cons(n,L)


  // SINGLETON LIST REPRESENTATION
  //
  private class Singleton (n:Int) extends NEList {

    def isEmpty ():Boolean = false

    def isSingleton ():Boolean = true

    def first ():Int = n

    def rest ():List = List.empty()

    def length ():Int = 1

    def append (M:List):NEList = cons(n,M)

    def find (f:Int):Boolean = (f==n)

    def isEqual (M:List):Boolean = {
      M.length()==1 && M.first()==n
    }
```

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

  override def hashCode ():Int = 41

  override def toString ():String = n.toString()
}


// CONS LIST REPRESENTATION
//
private class Cons (n:Int, L:List) extends NEList {

  def isEmpty ():Boolean = false

  def isSingleton ():Boolean = L.isEmpty()

  def first ():Int = n

  def rest ():List = L

  def length ():Int = 1 + L.length()

  def append (M:List):NEList = cons(n,L.append(M))

  def find (f:Int):Boolean = { (f == n) || L.find(f) }

  def isEqual (M:List):Boolean =
    { !(M.isEmpty()) && n==M.first() && L.isEqual(M.rest()) }

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

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

```
      41 + n.hashCode()
    ) + L.hashCode()

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


abstract class NEList extends List {

  def isEmpty ():Boolean
  def isSingleton ():Boolean
  def first ():Int
  def rest ():List
  def length ():Int
  def find (f:Int):Boolean
  def append (M:List):NEList
  def isEqual (M:List):Boolean
}
```

We can certainly check that the previously defined `sum()` and `average()` functions work with `NEList`, but we can do a bit better here. the `average()` function above needed to check that the list was non-empty before computing the average. With an `NEList`, we do not need to do this check. So we can define a variant of `average()` that works on `NEList`s and that does not perform that check. (We're using overloading to define a method with the same name as before, but working on different types — recall that Scala will disambiguate based on the *static type* of the argument.)

```
def average (x:NEList):Int = {
  (sum(x) / x.length())
}
```

Trying it out on the previous list, but now reconstructed as a `NEList`:

```
val L2:NEList = NEList.cons(33,NEList.cons(66,NEList.singleton(99)))
println("Sum = " + sum(L2))
println("Average (no check) = " + average(L2))
```

which yields, as before

```
Sum = 198
Average (no check) = 66
```

The result is of course the same, but the average function now does not need to do the emptiness check.

This may not look like a big win, but that's because the example is rather simple. More involved examples can be devised, for instance, a subtype of `Atlas` from your last homework that guarantees that all the exits lead to room that actually occur in the atlas. I'll leave that as an exercise.