8 Hiding Implementation Details

Last time we saw the *Interpreter Design Pattern* as a rather mechanical way to get an implementation that almost out-of-the-box implements a specification for an ADT, using a natural representation via concrete subclasses capturing what each of the creators of the ADT is doing.

That design pattern however reveals a lot of implementation details. The problem with reveal implementation details is that (1) it is information that is not part of the ADT signature, so it shouldn't be available, and (2) if it's there, someone will use it, and if someone uses it, that someone will be in trouble if later on you come back and reimplement the ADT using another implementatin that still satisfies the specification but does not reveal quite the same information as your original implementation.

Let's illustrate this with another example, which will come in handy later.

8.1 Another Example: Lists of Integers

Here is an ADT for lists of integers, using different creators than the usual `empty`/`cons` creators that you know and love.

```
CREATORS
  empty :     () -> List
  singleton:  Int -> List
  merge :     (List,List) -> List

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

and the obvious specification:

`empty().isEmpty()` = `true`

$$\texttt{singleton}(i).\texttt{isEmpty}() = \texttt{false}$$

$$\texttt{merge}(L,M).\texttt{isEmpty}() = \begin{cases} \texttt{true} & \text{if } L.\texttt{isEmpty}() = \texttt{true} \text{ and } M.\texttt{isEmpty}() = \texttt{true} \\ \texttt{false} & \text{otherwise} \end{cases}$$

$$\texttt{singleton}(i).\texttt{first}() = i$$

$$\texttt{merge}(L,M).\texttt{first}() = \begin{cases} M.\texttt{first}() & \text{if } L.\texttt{isEmpty}() = \texttt{true} \\ L.\texttt{first}() & \text{otherwise} \end{cases}$$

$$\texttt{singleton}(i).\texttt{rest}() = \mathit{empty}()$$

$$\texttt{merge}(L,M).\texttt{rest}() = \begin{cases} M.\texttt{rest}() & \text{if } L.\texttt{isEmpty}() = \texttt{true} \\ \texttt{merge}(L.\texttt{rest}(),M) & \text{otherwise} \end{cases}$$

$$\texttt{empty}().\texttt{length}() = 0$$
$$\texttt{singleton}(i).\texttt{length}() = 1$$
$$\texttt{merge}(L,M).\texttt{length}() = L.\texttt{length}() + \texttt{M}.\texttt{length}()$$

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

$$\texttt{singleton}(i).\texttt{find}(f) = \begin{cases} \texttt{true} & \text{if } i = f \\ \texttt{false} & \text{otherwise} \end{cases}$$

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

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

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

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

Applying the Interpreter Design Pattern, and providing a reasonable implementation for the canonical methods, we get something like:

```
/*
 * The default result of the InterpreterDesign Pattern
 *
 */


object List {

  def empty ():List = new ListEmpty()
  def singleton (i:Int):List = new ListSingleton(i)
  def merge (L:List, M:List):List = new ListMerge(L,M)
}


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 isEqual (L:List):Boolean = L.isEmpty()

  def length ():Int = 0

  def find (f:Int):Boolean = false

  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 = ""
}
```

```
class ListSingleton (i:Int) extends List {

  def isEmpty ():Boolean = false

  def first ():Int = i

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

  def isEqual (L:List):Boolean =
    (!L.isEmpty() && L.first()==i && L.rest().isEmpty())

  def length ():Int = 1

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

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

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

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


class ListMerge (L:List, M:List) extends List {

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

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

  def rest ():List =
    if (L.isEmpty())
      M.rest()
    else
```

```
      List.merge(L.rest(),M)

  def isEqual (N:List):Boolean =
    ((L.isEmpty() && M.isEmpty() && N.isEmpty()) ||
     (!N.isEmpty() && !this.isEmpty() &&
      N.first()==this.first() &&
      N.rest().isEqual(this.rest())))

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

  def find (f:Int):Boolean = (L.find(f) || M.find(f))

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

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

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



abstract class List {
  // declarations (so, no code)

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

Note the structure: a module `List`, an abstract class `List` that *declares* but does not *define* the operations of the ADT,[12] and two concrete classes that are subtypes of the list `List` and that actually define the operations declared in the abstract class (as well as the canonical methods).

Here's a simple interactive session illustrating the code above:

```
scala> val l = List.merge(List.singleton(66),List.singleton(99))
l: List =  66 99

scala> l.first()
res0: Int = 66

scala> l.rest()
res1: List =  99

scala> l.rest().first()
res2: Int = 99

scala> l.length()
res3: Int = 2
```

Now, the way I chose to print lists (in the `toString()` canonical method is ugly. Let's do something better that uses brackets at the beginning and the end of the list. That's easy to do if we have a helper method, which I'll call `toStringLst()`, and change `toString()` to use that helper method – in `ListEmpty`:

```
  def toStringLst () : String = ""

  override def toString () : String = "[ ]"
```

in `ListSingleton`:

```
  def toStringLst () : String = " " + n

  override def toString () : String = "[" + toStringLst() + " ]"
```

and in `ListMerge`:

[12] A *declaration* is a promise that you will implement the method when you subclass the abstract class. In other words, it's a promise that any subtype of the abstract will supply an implementation for those methods. You do not need to declare the canonical methods, because their canonical: they're always declarted by default in every class, abstract or otherwise. A *definition* is a declaration that also supplies some code defining the method.

```
  def toStringLst () : String = L.toStringLst() + M.toStringLst()

  override def toString () : String = "[" + toStringLst() + " ]"
```

Of course, if we make this change, then the code does not type-check anymore. Why? Look at the code. In the `toString()` implementation of `ListMerge`, we have a call to `L.toStringLst()`, where `L` is a `List`. When the type checker looks at `List` to see if a `toStringLst()` method is declared there, there isn't one. So it gives an error.[13] Thus, in order to satisfy the type checker, we need to make sure that every subtype of `List` defines `toStringLst()`, which is done by declaring `toStringLst()` in `List`. Here is the resulting implementation of the List ADT in full

```
/*
 * The default result of the Interpreter Design Pattern
 *   with a nicer toString() method
 *
 */

object List {

  def empty ():List = new ListEmpty()
  def singleton (i:Int):List = new ListSingleton(i)
  def merge (L:List, M:List):List = new ListMerge(L,M)
}


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 isEqual (L:List):Boolean = L.isEmpty()

  def length ():Int = 0
```

[13]Recall, the type checker is in charge of ensuring safety — that you never try to call methods that don't exist. Because `List` does not declare `toStringLst()`, someone could presumably add another subtype to `List` that does not define `toStringLst()` (since it's not declared in the abstract class `List`, subtypes of `List` are under no obligation to define it), and that new subtype could be used in the construction of a `List` using `merge()`, and when `toString()` on that list tries to call `toStringLst()` on that new subtype of `List`, it will fail to find it, breaking safety.

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

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

  override def hashCode () : Int = 41

  def toStringLst () : String = ""

  override def toString () : String = "[ ]"
}



class ListSingleton (i:Int) extends List {

  def isEmpty ():Boolean = false

  def first ():Int = i

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

  def isEqual (L:List):Boolean =
    (!L.isEmpty() && L.first()==i && L.rest().isEmpty())

  def length ():Int = 1

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

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

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

  def toStringLst () : String = " " + n
```

```
  override def toString () : String = "[" + toStringLst() + " ]"
}


class ListMerge (L:List, M:List) extends List {

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

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

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

  def isEqual (N:List):Boolean =
    ((L.isEmpty() && M.isEmpty() && N.isEmpty()) ||
     (!N.isEmpty() && !this.isEmpty() &&
      N.first()==this.first() &&
      N.rest().isEqual(this.rest())))

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

  def find (f:Int):Boolean = (L.find(f) || M.find(f))

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

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

```
  def toStringLst () : String = L.toStringLst() + M.toStringLst()

  override def toString () : String = "[" + toStringLst() + " ]"
}



abstract class List {
  // declarations (so, no code)

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

  // needed to satisfy the type checker
  def toStringLst ():String
}
```

We can test this as before:

```
scala> val l = List.merge(List.singleton(66),List.singleton(99))
l: List = [ 66 99 ]

scala> l.first()
res0: Int = 66

scala> l.rest()
res1: List = [ 99 ]

scala> l.rest().first()
res2: Int = 99

scala> l.length()
res3: Int = 2
```

That's a bit nicer.

Now, the problem with the above implementation is that it reveals two things: first, that there is a helper function called `toStringLst()`, even though it's not actually part of the

ADT, and second, that the implementation is in terms of three concrete classes `ListEmpty`, `ListSingleton`, and `ListMerge`.

To wit, continuing with the above example:

```
   // accessing a function not in the ADT signature:
scala> l.toStringLst()
res0: String =  66 99

   // accessing the representation classes directly
scala> val m = new ListMerge(l,new ListSingleton(33))
m: ListMerge = [ 66 99 33 ]
```

That's bad: if later on we decide on a new implementation of `toString()` and get ride of `toStringLst()`, then anyone that relied on that helper function existing will have their code suddenly stop working. The function is not in the signature, so it should not be available to anyone else. Similarly, if later on we decide on a new implementation of lists altogether, one that does not rely on three concrete classes, but perhaps on four, or just one, not necessarily using named `ListMerge` (say), then anyone creating a direct form of `ListMerge` will have their code suddenly stop working.

So: how do we hide the helper method, and how do we hide the fact that there are those two representation classes?

Let's deal with the second question first, partly because we cannot really hide helper functions in an abstract/concrete class combo without doing this first.

8.2 Hiding Representation Classes

So how do we hide information. The main tool we have for hiding information is to make that information private to some area of the code. For instance, if we mark a method as `private`, as in:

```
class Foo1 {

  def method1 (x:Int):Int = x+method2(x)

  private def method2 (x:Int):Int = x*2
}
```

then `method2` is only visible from within an instance of `Foo1`, while `method1` is visible from outside as well as from within. Thus, while this works:

```
val f = new Foo1
f.method1(10)
```

this causes a compile-time error saying that `method2` is inaccessible:

```
val f = new Foo1
f.method2(10)
```

We can similarly make fields private. For instance,

```
class Bar (x:Int) {

  val field1:Int = x+1

  private val field2:Int = 2*x
}
```

and the following works:

```
val b = new Bar(5)
b.field1
```

while the following fails for the same reason as `f.method2(10)` failed earlier:

```
val b = new Bar(5)
b.field2
```

So in general, we can make any component of a class private and thus hide it from whatever is outside the class. Thus, to hide the representation classes `ListEmpty`, `ListSingleton`, and `ListMerge`, then, it turns out we can simply them component of `List`! Such classes, nested inside other classes, are called *nested classes*.[14] Here is a simple example of a nested class, not hidden:

```
object Foo2 {

  def method1 (x:Int):Bar = new Bar(2*x)

  class Bar (value:Int) {

    def method2 (y:Int):Int = value+y
  }
}
```

[14]Following Java-based terminology, a class defined inside a module is sometimes called a *nested class*, while a class defined inside another class is a special kind of nested class called an *inner class*. Inner classes are exceedingly expressive, and are related to closures. In other words, inner classes give you `lambda`.

If we call `method1` in `Foo2`, the result with be an instance of class `Foo2.Bar`:

```
val b = Foo2.method1(10)
b.method2(30)
```

and the result should be 50. Note the type of `b`: `Foo2.Bar` — it has type `Bar` defined inside of `Foo2`. Classes accessed just like fields or methods when they occur inside `Foo2`. We can create instances of `Foo2.Bar` directly too:

```
val b = new Foo2.Bar(10)
b.method2(30)
```

which returns value 40.

Now, if we hide the nested class `Bar`:

```
object Foo3 {

  def method1 (x:Int):Bar = new Bar(2*x)

  private class Bar (value:Int) {

    def method2 (y:Int):Int = value+y
  }
}
```

then we get a problem trying to compile because our hiding worked too well: we do not have access to type `Bar` anymore (since `Bar` is hidden) and the system does not know how to refer to the value returned by `method1()` — the Scala compiler complains that the private class `Bar` *escapes* the class in which it is defined to be private. So one trick there is to simply use an abstract class to tell the compiler that yes, there is such a class, and it implements the following functions, but the implementation remains hidden:

```
object Foo4 {

  def method1 (x:Int):Bar = new BarImplementation(2*x)

  private class BarImplementation (value:Int) extends Bar {

    def method2 (y:Int):Int = value+y
  }
}
```

```
  abstract class Bar {

    def method2 (y:Int):Int
  }
```

Now we can still create instances of `BarImplementation` by calling `Foo4.method1()`, but not directly by calling `new Foo4.Bar()`.

Make sure you understand how the above works, since it's the structure we are going to be using. *Question: what happens if you make `FOOx` above a class instead of an object? Try it — see how you can make it work. Again, use the analogy that such defined classes are accessed just like fields or methods.*

Here is the above structure applied to our implementation of the List ADT — where I've compressed the nested classes to make the structure more apparent.

```
/*
 * The default result of the Interpreter Design Pattern
 *   with a nicer toString() method
 *   modified to hide concrete representation classes
 *
 */

object List {

  def empty ():List = new ListEmpty()
  def singleton (i:Int):List = new ListSingleton(i)
  def merge (L:List, M:List):List = new ListMerge(L,M)


  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 isEqual (L:List):Boolean = L.isEmpty()
    def length ():Int = 0
    def find (f:Int):Boolean = false
    override def equals (other : Any) : Boolean =
      other match {
        case that : List => this.isEqual(that)
        case _ => false
      }
```

```
  override def hashCode () : Int = 41
  def toStringLst () : String = ""
  override def toString () : String = "[ ]"
}


private class ListSingleton (i:Int) extends List {
  def isEmpty ():Boolean = false
  def first ():Int = i
  def rest ():List = List.empty()
  def isEqual (L:List):Boolean =
    (!L.isEmpty() && L.first()==i && L.rest().isEmpty())
  def length ():Int = 1
  def find (f:Int):Boolean = (i==f)
  override def equals (other : Any) : Boolean =
    other match {
      case that : List => this.isEqual(that)
      case _ => false
    }
  override def hashCode () : Int = 41 + i.hashCode()
  def toStringLst () : String = " " + n
  override def toString () : String = "[" + toStringLst() + " ]"
}


private class ListMerge (L:List, M:List) extends List {
  def isEmpty ():Boolean = (L.isEmpty() && M.isEmpty())
  def first ():Int =
    if (L.isEmpty())
      M.first()
    else
      L.first()
  def rest ():List =
    if (L.isEmpty())
      M.rest()
    else
      List.merge(L.rest(),M)
  def isEqual (N:List):Boolean =
    ((L.isEmpty() && M.isEmpty() && N.isEmpty()) ||
     (!N.isEmpty() && !this.isEmpty() &&
      N.first()==this.first() &&
      N.rest().isEqual(this.rest())))
```

```
    def length ():Int = L.length() + M.length()
    def find (f:Int):Boolean = (L.find(f) || M.find(f))
     override def equals (other : Any) : Boolean =
      other match {
        case that : List => this.isEqual(that)
        case _ => false
      }
    override def hashCode () : Int =
      41 * (
        41 + L.hashCode()
      ) + M.hashCode()
    def toStringLst () : String = L.toStringLst() + M.toStringLst()
    override def toString () : String = "[" + toStringLst() + " ]"
  }
}

abstract class List {
  def isEmpty ():Boolean
  def first ():Int
  def rest ():List
  def isEqual (L:List):Boolean
  def length ():Int
  def find (f:Int):Boolean

  // needed to satisfy the type checker
  def toStringLst ():String
}
```

8.3 Hiding Helper Functions

Now there only remain the question of how to hide the helper function.

If a helper function was only needed in one particular class, then hiding that helper function is easy — it's just like hiding `method2()` in `Foo1` above: `method2()` is only needed inside `Foo1`, by method `method1()` in particular, so it is marked `private`. So hiding a helper function used only within a class is trivial.

The problem, as we saw, is that `toStringLst()` is called across the concrete classes that are subtypes of `List`, and because of that, we had to declare it in the abstract class `List`. So we want to say that `toStringLst()` is only available within `List`. The way to do this is to make it the method private. In an abstract class, though, Scala will not let us use the keywork private. In an abstract class, for a declaration (not a definition) we have to use

protected.[15]

```
/*
 * The default result of the Interpreter Design Pattern
 *   with a nicer toString() method
 *   modified to hide concrete representation classes
 *   and to hide helper functions
 *
 */


object List {

  def empty ():List = new ListEmpty()
  def singleton (i:Int):List = new ListSingleton(i)
  def merge (L:List, M:List):List = new ListMerge(L,M)


  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 isEqual (L:List):Boolean = L.isEmpty()
    def length ():Int = 0
    def find (f:Int):Boolean = false
    override def equals (other : Any) : Boolean =
      other match {
        case that : List => this.isEqual(that)
        case _ => false
      }
    override def hashCode () : Int = 41
    def toStringLst () : String = ""
    override def toString () : String = "[ ]"
  }


  private class ListSingleton (i:Int) extends List {
    def isEmpty ():Boolean = false
```

[15]We will returned to `protected` later — it has to do with inheritance, and the use of `protected` in abstract classes instead of `private` is to maintain consistency with the rest of the language. But we haven't seen the rest of the language, so the consistency is not apparent yet. So for now, just remember that in abstract classes, `protected` is equivalent to `private`.

```
  def first ():Int = i
  def rest ():List = List.empty()
  def isEqual (L:List):Boolean =
    (!L.isEmpty() && L.first()==i && L.rest().isEmpty())
  def length ():Int = 1
  def find (f:Int):Boolean = (i==f)
  override def equals (other : Any) : Boolean =
    other match {
      case that : List => this.isEqual(that)
      case _ => false
    }
  override def hashCode () : Int = 41 + i.hashCode()
  def toStringLst () : String = " " + n
  override def toString () : String = "[" + toStringLst() + " ]"
}


private class ListMerge (L:List, M:List) extends List {
  def isEmpty ():Boolean = (L.isEmpty() && M.isEmpty())
  def first ():Int =
    if (L.isEmpty())
      M.first()
    else
      L.first()
  def rest ():List =
    if (L.isEmpty())
      M.rest()
    else
      List.merge(L.rest(),M)
  def isEqual (N:List):Boolean =
    ((L.isEmpty() && M.isEmpty() && N.isEmpty()) ||
     (!N.isEmpty() && !this.isEmpty() &&
      N.first()==this.first() &&
      N.rest().isEqual(this.rest())))
  def length ():Int = L.length() + M.length()
  def find (f:Int):Boolean = (L.find(f) || M.find(f))
   override def equals (other : Any) : Boolean =
    other match {
      case that : List => this.isEqual(that)
      case _ => false
    }
  override def hashCode () : Int =
```

```
      41 * (
        41 + L.hashCode()
      ) + M.hashCode()
    def toStringLst () : String = L.toStringLst() + M.toStringLst()
    override def toString () : String = "[" + toStringLst() + " ]"
  }
}

abstract class List {
  def isEmpty ():Boolean
  def first ():Int
  def rest ():List
  def isEqual (L:List):Boolean
  def length ():Int
  def find (f:Int):Boolean

  protected def toStringLst ():String
}
}
```

And there: we cannot create instances of `ListEmpty`, `ListSingleton`, and `ListMerge` from outside `List` (and therefore we have to use the creators to create lists), and once we have a list we cannot call its `toStringLst()` function, as the compiler will prevent. For instance, if we try what we tried before but with our new code:

```
scala> val l = List.merge(List.singleton(66),List.singleton(99))
l: List = [ 66 99 ]

scala> l.toStringLst()
<console>:7: error: method toStringLst cannot be accessed in List
       l.toStringLst()
         ^

scala> val m = new ListMerge(l,new ListSingleton(33))
<console>:6: error: not found: type ListMerge
       val m = new ListMerge(l,new ListSingleton(33))
                   ^

scala> val m = new List.ListMerge(l,new List.ListSingleton(33))
<console>:6: error: class ListMerge cannot be accessed in object List
       val m = new List.ListMerge(l,new List.ListSingleton(33))
scala>
```

Things fail miserably, as they should.

Now, note something interesting. The above *should not* work! We're declaring `toStringLst()` protected in `List`. Which is the same as making it `private` — it means, in particular, that there we are not allowed to refer to that method from outside the class `List` itself. But we're invoking `toStringLst()` on something of type `List` in `ListMerge` in method `toStringLst()`. Why does the compiler let us do that? The answer is that we're made use of a particular back-door in Scala. The fact that the module in which `ListEmpty`, `ListSingleton`, and `ListMerge` are defined has the same name `List` as the abstract class is the key point — as I mentioned earlier in the course, they are called *companions*: a module and a class of the same name. Being companions means that they can access each other's private components as though they were defined within themselves. So in the case of `List`, for the purposes of what's private/public, it's as if we had defined:

```
object/class List {

  def empty ():List = new ListEmpty()
  def singleton (i:Int):List = new ListSingleton(i)
  def merge (L:List, M:List):List = new ListMerge(L,M)

  private class ListEmpty () extends List {
    ...
  }

  private class ListSingleton (i:Int) extends List {
    ...
  }

  private class ListMerge (L:List, M:List) extends List {
    ...
  }

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

  protected def toStringLst () : String
}
```

meaning that whatever is inside this combo `List` can access `toStringLst()` since it is defined

within the same scope. And since `ListCons` is inside the scope (it is a component within `List`), it can access `toStringLst()`.

This works only because the module and the class have the same name `List`. If we change that, that is, if we change the name of module `List` to something like `ListCreators`, our code breaks: the system complains that `toString()` in `ListCreators.ListMerge` attempts to call the inaccessible method `toStringLst()` in `List`. If you ever get that error, you'll know where to look.