Linked Lists

We already used small `Node` objects to implement a stack without using an array.

Now we generalize this idea to store a sequence of elements in a linked list.

The `Node` objects:

```
class Node(val head: String, val tail: Node)
```

This is a recursive definition!

Making a short list:

```
scala> var l = new Node("apples", null)
scala> l = new Node("oranges", l)
scala> l = new Node("strawberries", l)
```

What is `l.tail.head`?

List methods

It is more appropriate to make `display` and `length` methods of the `Node` class:

```
class Node(val head: String, val tail: Node) {
  def length: Int = if (tail == null) 1
                    else 1 + tail.length
  def display() {
    println(head)
    if (tail != null) tail.display()
  }
}
```

readlist2.scala

There are two problems:

- Stack-overflow for long lists
- How to handle empty lists?

Recursive list functions

Since lists are defined recursively, it is natural to handle them with recursive functions, for instance to display the list or to compute the length.

```
def displayList(L: Node) {
  if (L != null) {
    println(L.head)
    displayList(L.tail)
  }
}
```

readlist1.scala

Non-recursive list traversal

We rewrite `length` and `display` without recursion to avoid the stack-overflow:

```
  def display() {
    var p = this
    while (p != null) {
      println(p.head)
      p = p.tail
    }
  }
```

readlist3.scala

Representing empty lists

If we want empty lists to have methods, we cannot represent them with `null`.

So we make a special object representing any empty list:

```
object Empty {
  def length: Int = 0
  def display() = ()
}
```

readlist4.scala

But this doesn't work, since `Empty` cannot be referenced by a `Node`:

```
class Node(val head: String, val tail: Node) {
```

cannot be `Empty`

List trait

Here is our full definition of a list:

readlist5.scala

```
trait List {
  def length: Int
  def head: String
  def tail: List
  def display(): Unit
}
object Empty extends List {
  def length: Int = 0
  def head: String = throw new NoSuchElementException
  def tail: List = throw new NoSuchElementException
  def display() = ()
}
class Node(val head: String, val tail: List) extends List {
  def length: Int = 1 + tail.length
  def display() { println(head); tail.display() }
}
```

`length` and `display` are nice and simple!

Scala Lists

The Scala `List` works exactly like the list we designed.

The empty list (of any type) is called `Nil`.

The node object is of type `::`, which is a small class with the two fields `head` and `tail`.

```
scala> var l = ::(3, Nil)
l: scala.collection.immutable.::[Int] = List(3)
scala> l = ::(7, l)
l: scala.collection.immutable.::[Int] = List(7, 3)
```

Or, more elegantly:

```
scala> var l = 7 :: (3 :: Nil)
l: List[Int] = List(7, 3)
scala> val a = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
a: List[Int] = List(1, 2, 3, 4, 5)
scala> val b = List(2, 3, 5, 7, 11)
b: List[Int] = List(2, 3, 5, 7, 11)
```


Scala List features

Concatenate two lists with `:::`, and convert to and from arrays with `toArray` and `toList`.

Lists offer many of the same methods as arrays (in fact all Scala sequences have these):

- `L.length`
- `L.isEmpty`
- `L.nonEmpty`
- `L drop n`
- `L dropRight n`
- `L take n`
- `L splitAt n`, which returns a pair consisting of `L take n` and `L drop n`
- `L.mkString`
- `L.mkString(separator)`
- `L.mkString(prefix, separator, suffix)`
- `L.reverse`
- `L.sorted`