32-Streams

Non-strict collections

- There are three special types of collection which are *non-strict* (lazy to you and me):
  - *Iterator*: evaluates elements as needed but they cannot be revisited;
  - *Stream*: evaluates element as needed and they can be revisited;
  - *SeqView* (actually, there are other types of view, too): essentially just “decorates” a collection with a transformation function
- Each of these has different behavior but what is generally common is that an element in the sequence will not be evaluated if you never actually need it.

Streams are lazy Lists

- We briefly mentioned *Streams* last week
  - Like a *List*, a *Stream* has a *head* and a *tail* but
  - unlike in a *List*, the *tail* of a *Stream* is call-by-name.

```
trait Stream[A] {
  def head: A
  def tail: Stream[A]
}
case class Cons[A](head: A, tail: => Stream[A]) extends Stream[A]
case object empty extends Stream[Nothing] {
  def head: throw NoSuchElementException(“head of empty stream”)
  def tail: throw UnsupportedOperationException(“tail…stream”)
}
```

Streams

- A *Stream* is just like a *List* except that the tail is evaluated lazily.
  - Here is the *Cons* method (note that *tl* is call-by-name):

```
final class Cons[+A](hd: A, tl: => Stream[A]) extends Stream[A] {
  override def isEmpty = false
  override def head = hd
   // rest of definition
}
```

- What should the signature of the *map* method be?

```
def map[B](f: A=>B): Stream[B]
```

- Note that *map* always preserves the shape (and nature) of *this* in the result.
- As we will discover, *map* is a rather fundamental method

Working with Streams

Ways to create a Stream:

```
Stream.empty
import Stream._
1 #:: 2 #:: empty
cons(1, cons(2, empty))
from(1)
continually(9)
range(1, 20, 3)
```

concatenation

turning an (infinite) Stream into a (finite) List:

```
from(1) take 10 toList
```

What do you think this function does?

Note: recursive even though ***f*** is a val.

A bit like foldLeft but retains shape

```
val f: Stream[Long] = 0L #:: f.scanLeft(1L)(_ + _)
```

```
val g: Stream[Long] = 0L #:: 1L #:: g.zip(g tail).map (n => n._1
+ n._2)
```

Should be a bit easier to understand.

Fibonacci Stream

```
scala> val f: Stream[BigInt] = BigInt(0) #:: f.scanLeft(BigInt(1))(_ + _)
f: Stream[BigInt] = Stream(0, ?)

scala> Stream.from(1) zip f take 100 foreach println
(1,0)
(2,1)
(3,1)
(4,2)
(5,3)
(6,5)
(7,8)
(8,13)
(9,21)
(10,34)
etc. etc.
(100,?)
```