Managing Mutable State

Part 1: Capturing State

Problem with Random Number Generation

- Random numbers (for testing)
  - are usually generated by a PRNG (pseudo-random-number-generator), e.g. *java.util.Random*
  - But, by definition, such a PRNG is **non-idempotent** and therefore **non-referentially-transparent**;
    - That's to say:
      `random.getNext != random.getNext`
    - and this is anathema to functional programming.
  - In the first week, I referred to the "evils" of mutable state for testing purposes.

Problem with PRNGs continued

- Is there anything we can do about this?
  - Or, are we forever going to be limited by this lack of referential transparency?
- Let's think about the root cause of the problem?
  - It's that a mutable object can change its state without a referrer knowing about it—the referrer is *in the dark:*
  - There are two ways this can happen:
    - the mutable object *spontaneously* mutates (e.g. the system clock or a remote web service);
    - the mutable object is referenced in another thread and is updated there (e.g. draws the next random number).

Carry your protection with you

- Maybe tortoises used to protect themselves by hiding under a rock—then they found it more convenient to carry the rock around with them on their backs ?!
    - *BTW, I don't believe this of course!*

What if we carry our state with us?

- Let's define the following trait:

```
trait RNG[A] {
  def next: RNG[A]
  def value: A
}
```

  - It has two properties: its (random) value and the next RNG in the series…
    - … from which, of course, we can get another random value.

- What we've defined is like the old concept of a "one-time pad" used for setting an encryption key.
  - We only ever call *value* on an instance of *RNG once* (it always yields the same result because its immutable)!
  - To get the next value in the series we invoke `next.value`

Using it in practice

- Here we test in a Spec file:

```
behavior of "RNG"
it should "allow predictable sequential usage" in {
  val r0 = LongRNG(0L)
  val r1 = r0.next
  r1.value shouldBe -4962768465676381896L
  val r2 = r1.next
  r2.value shouldBe 4804307197456638271L
  val r3 = r2.next
  r3.value shouldBe -1034601897293430941L
}
```

- Still, that’s not super-useful. For practical purposes, we will need either:
    - a (mutable) *Stream* of random values that can be consumed in different places; *Or*
    - an (immutable) *Stream* of random values that can be consumed in one place.

Streamer

- I call the first of these a *Streamer* (a general concept):

```
/**
  * This class is based on a mutable Stream.
  * Its purpose is like a one-time-pad: each value is yielded by the Streamer once and once only.
 *
  * @param s the Stream
  * @tparam X the underlying type and the type of the result
  */
case class Streamer[X](private var s: Stream[X]) extends (()=>X) {

  apply() // We need to skip over the first value

  /**
    * This method mutates this Streamer by resetting the value of s to its tail and returning its head.
    * @return the head of the Stream
    */
  override def apply(): X = s match {
    case x #:: tail => s = tail; x
  }

  def take(n: Int): Seq[X] = {
    @tailrec def inner(xs: Seq[X], i: Int): Seq[X] = if (i==0) xs else inner(xs :+ this(), i-1)
    inner(Seq.empty,n)
  }
}

============================================================

behavior of "take"
it should "work with random number generator" in {
  val target = Streamer(RNG.values(LongRNG(0L)))
  target.take(4) shouldBe Seq(-4962768465676381896L, 4804307197456638271L, -1034601897293430941L,
7848011421992302230L)
}
```

Random Stream

- The second is simply a *Stream* of random numbers, generated from an RNG[T] object

```
trait RNG[T] {
  /**
    * @return the next random state in the pseudo-random series
    */
  def next: RNG[T]

  /**
    * @return the value of this random state (renamed from value)
    */
  def get: T

 /**
    * @return a stream of T values
    */
  def toStream: Stream[T]
}
```

Guess what?

- You’re going to implement the second of these in Assignment 3.