Updated: 2021-02-07

3.4 Mutable vs. Immutable:
How to avoid using mutable anything.

So, you really want to use mutable variables

Nothing I can do to dissuade you?

- I recently was working on a method to force *x* into the range 0 through 2pi. I was in a hurry and couldn't think how best to do it without a *var*. This has rarely happened to me but I did it anyway:

```
// NOTE: using a var here!!
var result = z
while (result < min) result = nx.plus(result, nx.plus(max, nx.negate(min)))
while (result > max) result = nx.plus(result, nx.plus(min, nx.negate(max)))
result
```

- Note that I wrote a "NOTE" comment to be sure that everyone looking at this code would be warned about me using *var*.
- A little later, I decide to clean it up. This is what I wrote:

```
@tailrec
def inner(result: X): X =
  if (result < min) inner(nx.plus(result, nx.plus(max, nx.negate(min))))
  else if (result > max) inner(nx.plus(result, nx.plus(min, nx.negate(max))))
  else result
inner(z)
```

Was that so hard?

- No, it wasn't hard. It was easy. And it just looked so much more elegant.
- Here's another situation where you will want to use a mutable collection:

```
case class Mill (stack: mutable.Stack[Number]) {
  def push(n: Number): Unit = {
    stack.push(n)
  }
  def pop(): Option[Number] = if (stack.isEmpty) None else Some(stack.pop())
  def isEmpty: Boolean = stack.isEmpty
}
val mill = Mill()
mill.push(Number.one)
mill.pop() match {
  case None => fail("logic error")
  case Some(x) => x shouldBe Number.one
}
```

- But, surprisingly perhaps, you don't need to do it this way.

Proper way to define Mill

- Here we use an immutable List:

```
case class Mill(stack: List[Number]) {
  def push (x: Number): Mill = Mill(x :: stack)
  def pop: (Option[Number], Mill) = stack match {
    case Nil => (None, this)
    case h :: t => (Some(h), Mill(t))
  }
  def isEmpty: Boolean = stack.isEmpty
}
val mill = Mill().push(Number.one)
mill.pop match {
  case (None, _) => fail("logic error")
  case (Some(x), _) => x shouldBe Number.one
}
```

- Notice that there is no immutable *Stack* class in Scala. Why? Because a stack and a list are exactly the same thing!
- Notice also that both *push* and *pop* must return a *Mill*. In the case of *pop*, we must return a tuple of *Mill* and *Option[Number]*.

Hash tables and caches

- Another situation where you'll be tempted to use a mutable collection is when you set up a Hash table.
- Now, depending on your actual use case, this may be appropriate. But like the use of the stack/list in the *Mill*, you don't have to do it that way.
- The place where you will need to make it mutable is when you are defining a cache which has a relatively global scope. It might be awkward and inelegant to carry the latest version of the cache around in all of your logic. One way to get around this is to make the pointer to the cache a *var*.
- Fortunately, there's a pattern in functional programming which takes care of this situation. It's called an actor. An actor *hides* its state from other code because that other code does not have direct access to the state: the access is *indirect* via the actor manager (or actor system).

Let’s set up a cache: first the traits

```
trait Cache[K, V] extends (K => Future[V])
```

- This trait simply defines a cache as a function that takes a *K* (key type) and returns a *Future[V]* (the value type wrapped in the asynchronous wrapper called *Future*).

```
trait ExpiringKey[K] {
  def expire(k: K): Unit
}
```

- This trait defines a method which can be used to expire (remove) a key from a cache. We haven’t shown how this method might get called, based perhaps on a timer.

```
trait Fulfillment[K, V]{
  val fulfill: K => Future[V]
}
```

- This trait describes how a key-value pair actually gets fulfilled such that it can be stored in a cache.

Let’s set up a cache: next the case class

```
case class FulfillingCache[K, V](fulfill: K => Future[V]) extends
Cache[K, V] with ExpiringKey[K] with Fulfillment[K, V] {
  private def put(k: K, v: V): Unit = cache += (k -> v)
  def apply(k: K): Future[V] = if (cache.contains(k)) Future(cache(k))
else for (v <- fulfill(k); _ = put(k, v)) yield v
  def expire(k: K): Unit = cache -= k
  private val cache: mutable.Map[K, V] = mutable.Map.empty
}
```

- It doesn’t matter that this class has a difficult name. We will never actually use it in application code.
- Two code fragments to note: in method *put*, we see “`cache += (k -> v)`”
- That += method is the one that should be used when adding something to a mutable collection.
- Similarly, in method *expire*, we see “`cache -= k`” where the -= method is the one to remove an element from a mutable collection.

Using a factory class

- Factory classes are well-known from object-oriented programming.

```
object CacheFactory {
  def createCache[K, V](fulfill: K => Future[V]): Cache[K, V] =
FulfillingCache(fulfill)
  def lookupStock(k: String): Future[Double] = Future(MockStock.lookupStock(k))
  def createStockCache: Cache[String, Double] = createCache(lookupStock)
}
```

- We define a base factory method *createCache* and a specific factory method *createStockCache,* which defines the *fulfill* method based on some method called *lookupStock,* which we don't really need to know about.

- Because of encapsulation, and because *createCache* returns a *Cache*, not a *FulfillingCache*, the caller does not know anything about fields such as *cache* field.

Iteration and Recursion

Church-Turing Thesis

- Iteration requires mutation; recursion requires immutability.
- In general, we can always replace iteration with recursion or *vice versa*.
- For detail, see https://en.wikipedia.org/wiki/Recursion_(computer_science)