More On For-Expressions
Recap: Collections

Core classes

Iterable
  |-- Seq
     |-- List
        |-- Stream
        |-- Vector
        |-- Range
        |-- Array
        |-- String
     |-- Set
        |-- HashSet
               |-- TreeSet
        |-- Map
               |-- HashMap
               |-- TreeMap
Recap: Collection Methods

Core methods:

- map
- flatMap
- filter

and also

- foldLeft
- foldRight
For-Expressions

Simplify combinations of core methods map, flatMap, filter.

Instead of:

```scala
(1 until n) flatMap (i =>
  (1 until i) map (j => (i, j))) filter (pair =>
  isPrime(pair._1 + pair._2))
```

one can write:

```scala
for {
  i <- 1 until n
  j <- 1 until i
  if isPrime(i + j)
} yield (i, j)
```
Other Uses of For-Expressions

Operations of sets, or databases, or options.

**Question:** Are for-expressions tied to collections?

**Answer:** No! All that is required is some interpretation of `map`, `flatMap` and `withFilter`.

There are many domains outside collections that afford such an interpretation.

Two examples: Random values and futures.
Random Values

You know about random numbers:

```java
import java.util.Random
val rand = new Random
rand.nextInt
```

Question: What is a systematic way to get random values for other domains:

- booleans
- strings
- pairs and tuples
- lists
- sets
Generators

Let's define a class `Generator[T]` that can generate random values of type `T`:

```scala
trait Generator[+T] {
  def generate: T
}
```

Some instances:

```scala
val integers = new Generator[Int] {
  def generate = scala.util.Random.nextInt()
}

val booleans = new Generator[Boolean] {
  def generate = integers.generate >= 0
}

val pairs = new Generator[(Int, Int)] {
  def generate = (integers.generate, integers.generate)
}
```
Streamlining It

Can we avoid the new Generator ... boilerplate?

Ideally, would like to write:

```scala
val pairs = for {
  x <- integers
  y <- integers
} yield (x, y)
```

Need `map` and `flatMap` for that!
Here's a more convenient version of `Generator`:

```scala
trait Generator[+T] {
  self =>  // an alias for "this".

  def generate: T

  def flatMap[S](f: T => Generator[S]): Generator[S] = new Generator[S] {
    def generate = f(self.generate).generate
  }

  def map[S](f: T => S): Generator[S] = new Generator[S] {
    def generate = f(self.generate)
  }
}
```
Some Generators

```scala
iplicit def integers: Generator[Int] = new Generator[Int] {
  def generate = scala.util.Random.nextInt()
}

implicit def choose(lo: Int, hi: Int): Generator[Int] = new Generator[Int] {
  def generate = scala.util.Random.nextInt(hi - lo) + lo
}

implicit def single[T](x: T): Generator[T] = new Generator[T] {
  def generate = x
}
```
More Generators

```scala
implicit def booleans: Generator[Boolean] = integers.map(_ >= 0)

implicit def pairs[T, U](implicit t: Generator[T], u: Generator[U]): Generator[(T, U)] = 
  for {
    x <- t
    y <- u
  } yield (x, y)
```
You know about units tests:

- Come up with some test inputs to program functions and a postcondition.
- The postcondition is a property if the expected result.
- Verify that the program satisfies the postcondition.

**Question:** Can we do without the test inputs?

Yes, by generating random test inputs
Using generators, we can write a random test function:

```scala
def test[T](g: Generator[T], numTimes: Int = 100)(test: T => Boolean): Unit = {
  for (i <- 0 until numTimes) {
    val value = g.generate
    assert(test(value), "test failed for " + value)
  }
  println("passed " + numTimes + " tests")
}
```

Example usage:

```scala
test(lists[Int]) {{xs: List[Int]} =>
  xs.reverse == xs
}
Shift in viewpoint: Instead of writing tests, write properties that are assumed to hold.

This idea is implemented in the ScalaCheck tool.

It can be used either stand-alone or as part of ScalaTest.

See ScalaCheck tutorial on the course page.
Asynchronous Processing

Programs are often *asynchronous*: Several tasks, some results need waiting.

Examples:

- I/O
- Webservices
- Inter-process communication

Want to avoid blocking waits.

```java
SlowService(request).get()
// System hangs until SlowService has finished
```
A Future represents a value that will be computed in the future.

First version:

```scala
class Future[+T] {
  def get: T
}
```

If SlowService returns a future, we can now do something useful in the meantime:

```scala
val myFuture = MySlowService(request) // returns right away
...do other things...
val result = myFuture.get() // blocks until service "fills in" myFuture
```
Asynchronous Use of Futures

Problem: Once we call `get`, we still block!

Would like to use a *call-back*, be notified when future is ready.

Here’s how this works:

```scala
val future = MySlowService(request)
future onSuccess { reply =>
  // when the future gets ”filled”, use its value
  println(reply)
}
```

This assumes an `onSuccess` operation in class `Future`:

```scala
def onSuccess[U](cont: T => U): U
```
Downside of Callbacks

Problem with too many callbacks: spaghetti-code.

Would like to write code like:

```scala
val user = getUserById(id)
val orders = getOrdersForUser(user.email)
val products = getProductsForOrders(orders)
val stock = getStockForProducts(products)
```

But have it work asynchronously out of the box.
Composition of Futures

We can do better with (you guessed it!) for expressions.

```
for {
    user <- getUserById(id)
    orders <- getOrdersForUser(user.email)
    products <- getProductsForOrders(orders)
    stock <- getStockForProducts(products)
} yield stock
```

To make this work, futures need map and flatMap operations.
Outline of Class Future

class Future[+T] { self =>
    def get: T
    def onSuccess[U](cont: T => U): U
    def map[U](f: T => U): Future[U] =
    def flatMap[U](f: T => Future[U]): Future[U]
}
Monads

Data structures with map and flatMap seem to be quite common.

In fact there’s a name that describes this class of data structures together with some algebraic laws that they should have.

They are called *monads*.

Monads are very popular in the Haskell programming language.