Week 6

Other collections

lists are linear - access to the first element is much faster than access to the middle/end elements.

but in Scala there is also an alternative sequence implementation, Vector - it has more evenly balanced access pattern than List

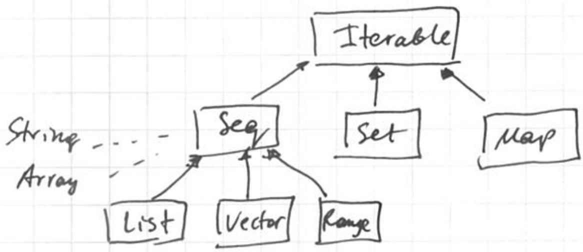
```
val  nums = Vector (1, 2, 3)
val  people = Vector ("Bob", "James", "Peter")
```

They support the same operations, with the exception of ::
Instead of :: there is

x +: xs create a new ~~element~~ vector with leading element x, followed by all elements of xs

xs :+ x create a new vector with trailing element x

A common base class for List~~s~~ and Vector is Seq



Seq is a subclass of Iterable

Arrays and Strings support the same operations as Seq - and can implicitly be converted to sequences when needed

But they can't be subclasses - because they come from Java.

```
val xs: Array[Int] = Array(1,2,3)
xs map (x => 2 * x)

~~val~~ val ys: String = "Hello, World!"
        ys.filter(_.isUpper)
```

Range
represents evenly spaced integers

three operators
- to (inclusive)
- until (exclusive)
- by (step)

```
val r: Range = 1 until 5    // 1, 2, 3, 4
val s: Range = 1 to 5       // 1, 2, 3, 4, 5

1 to 10 by 3                // 1, 4, 7, 10
6 to 1 by -2                // 6, 4, 2
```

More sequence operations:

xs exists p	true if at least for one el p holds true
xs forall p	true if for all elements p holds
xs zip ys	a sequence of pairs drawn from corresponding elements of sequences xs and ys
xs flatMap f	applies collection-valued f to all elements and concatenates the results

xs.sum
xs.product
xs.max
xs.min

Example:

(1 to M) flatMap (x => (1 to N) map (y => (x, y)))

- to list all combinations of numbers x and y where x is drawn from 1..M, y – from 1..N.

- scalar product of two vectors

```
def scalarProduct(xs: Vector[Double],
                  ys: Vector[Double]): Double =

    (xs zip ys).map(xy => xy._1 * xy._2).sum
```

$$\sum_{i=1}^{n} x_i * y_i$$

Alternative way - use pattern matching function value

```
def scalarProduct(.. ) .. =

  (xs zip ys).map { case (x,y) => x * y }.sum
```

Generally, the function value

```
{ case p1 => e1 ... case pn => en }
```

is equivalent to

```
x => x match { case p1 => e1 ...
               case pn => en }
```

- isPrime

```
def isPrime(n: Int): Boolean =
    (2 until n) forall (d => n % d != 0)
```

Combination Search and For-Expressions

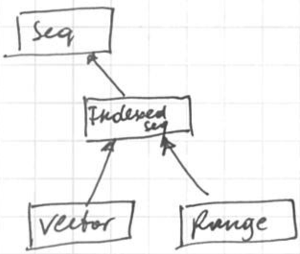
Handling Nested Sequences

We can extend the usage of higher order functions on sequences to many calculations
which are usually expressed using nested loops

Eg: given a positive int. n find all pairs of positive ints i and j, with
$1 \le j < i < n$,
such that $i+j$ is prime

for $n=7$ pairs are

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11



Algorithm

- generate the seq of all pairs (i, j) such that $1 \le j < i < n$
- filter ones for which $i+j$ is prime

```
(1 until n).map(i =>
    (1 until i).map(j => (i, j)))
```

it returns a sequence of sequences -
lets call it xss

we can combine all the subsequences using ++

```
(xss foldRight Seq[Int]())(_ ++ _)
```

or using "flatten"

```
xss.flatten
```

- Or, we can do flatMap

```
xs flatMap f = (xs map f).flatten
```

and

```
(1 until n) flatMap (i =>
    (1 until i) map (j => (i, j)))
```

```
xss.filter (pair =>
    isPrime (pair._1 + pair._2))
```

- Simpler?

For-Expression example

```
case class Person(name: String, age: Int)
```

to obtain the names of people over 20 years olds

```
for (p <- persons if p.age > 20) yield p.name
```

which is equivalent to

```
persons filter (p => p.age > 20) map (p => p.name)
```

It's similar to for-loops, but it builds a list as a result

Syntax

for (s) yield e

- s is a sequence of generators and filters
- e expression to return from each iteration

- generator : $p \leftarrow e$
  - p is a pattern
  - e is an expression

- filter : if f
  - f - is a boolean expression

- the seq. must start with a generator
- if there are several generators, the last generator vary faster than the first

Instead of (s) { s } can also be used - and seq. of generators/ ~~can be~~ filters can be written on multiple lines

```
for {
    i <- 1 until n
    j <- 1 until i
    if isPrime(i+j)
} yield (i, j)
```

Eg scalar product

```
(for ((x, y) <- xs zip ys) yield x * y).sum
```

Combinatorial Search Example

Sets

```
val fruit = Set("apple", "banana", "pear")
val s = (1 to 6).toSet
```

Most operations on seqs are also availabe for sets

```
s map (_ + 2)
fruit filter (_.startsWith == "app")
s.nonEmpty
```

Sets vs Seqs

1. Sets are unordered
2. Sets don't have dublocates
3. fundamental operation is contains

```
s contains 5
```

Example: N-Queens

how to place N queens on a chessboard - so that noone is theatened by another -

can't be two queens in the same raw, col, diagonal

A way to solve:

once we placed $k-1$ queens, we plase the k^{th} in a column where it's not "in check" with any other queen

Algorithm

- Suppose we generated all solutions for $k-1$ queens for a board of size n
- Each solution - is a list (of len $k-1$) containing the number of col (0 to $n-1$)
- the col number of any queen in the $(k-1)^{th}$ row comes first in the list, followed by the col number of the queen in row $(k-2)^{th}$, etc
- the solution set - is a set of lists, with one element for each solution
- to place n^{th} queen we generate all possible extensions of each solution - with a new queen

```
def queens(n: Int) = {

    def placeQueens(k: Int): Set[List[Int]] = {

        if (k == 0) Set(List())
        else
            for {
                queens <- placeQueens(k-1)
                col <- 0 until n
                if isSafe(col, queens)
            } yield col :: queens

    }

    placeQueens(n)
}
```

Exercise:
write a function

```
def isSafe(col: Int, queens: List[Int]): Boolean
```

Queries with For

```
case class Book(title: String, authors: List[String])
```

```
for (b <- books; a <- b.authors if
            a startsWith "Bird")
        yield b.title
```

For-expressions and Higher-Order Functions

mapFun, flatMap, filter

```
def mapFun[T, U](xs: List[T], f: T => U]): List[U] =
    for (x <- xs) yield f(x)

def flatMap[T, U](xs: List[T], f: T => U]): List[U] =
    for (x <- xs; y <- f(x)) yield y

def filter[T](xs: List[T], p: T => Boolean): List[T] =
    for (x <- xs if p(x)) yield x
```

But in Scala for-expressions are implemented in terms of map, flatMap and a lazy variant of filter

1. for (x <- e1) yield e2

$\Downarrow$ translated into

e1.map(~~e1~~) x => e2)

2. for (x <- e1; y <- e2; s) yield e3

$\Downarrow$

e1.flatMap(x => for (y <- e2; s) yield e3)

Eg.

```
for {
    i <- 1 until n
    j <- 1 until i
    if isPrime (i+j)
} yield (i, j)
```

$\Downarrow$

```
(1 until n).flatMap (i =>
    (1 until i) withFilter (j => isPrime (i+j)).
  map(j => (i, j)))
```

Eg Maps

Map[Key, Value] - associates a key with a value

```
val romal = Map("I" -> 1, "V" -> 5, "X" -> 10)
val capitals = Map("US" -> "Washington",
                   "Switzerland" -> "Bern")
```

Class Map extends Iterable [(key, Value)]

So you can do everything with maps

```
val countries = capitals map {
    case (x, y) => (y, x)
}
```

~~Ex~~ Querying:

capital("Andorra")

applying map to non-existent key gives an error

(java.util.NoSuchElement)

You can use get

capital get "US" → Some("Washington")
capital get "Andorra" → None

returns an ~~Optional~~ Option value

Option type

```
trait Option [+A]
case class Some [+A] (value: A] extends Option [A]
object None extends Option [Nothing]
```

get returns

- None if a map doesn't contain key
- Some(x) if does

decomposing

```
def showCapital (country: String) =
    capital.get(country) match {
  case Some(capital) => capital
  case None => "missing data"
}
```

Sorted and GroupBy

```
val fruit = List(..)

fruit sortWith (_.lenth < _.length)
  or
fruit sorted
```

GroupBy partitions a collection into a map of collections according to a discriminator function f

```
fruit groupBy (_.head)

        ⇓

  Map(p → List(pear, pineapple),
      a → List(apple),
      o → List(orange))
```

Default Values

```
val cap1 = capitals withDefaultValue "<unknown>"
cap1("Andorra") → "unknown"
```

Variable Length Arguments Lists

Polynom(Map(1 -> 2.0, 3 -> 4.0, 5 -> 6.2))

Can we do without map?

We can use a repeated parameter:

```
def Polynom(bindings: (Int, Double)*) =
    new Polynom(bindings.toMap withDefaultValue 0)
```

Polynom(1 -> 2.0, 3 -> 4.0, 4 -> 6.2)

Inside, bindings is seen as a Seq[(Int, Double)]

Implementation of Polynom

```
class Poly(terms0: Map[Int, Double]) {
    def this(bindings: (Int, Double)*) =
        this(bindings.toMap)

    val terms = terms0 withDefaultValue 0.0

    def + (other: Poly) = new Poly(
        terms ++ (other.terms map adjust))

    def adjust(term: (Int, Double)): (Int, Double) = {
        val (exp, coef) = term
        exp -> (coeff + terms(exp))
    }
    // toString
}
```

or

```
    def +(other: Poly) =
        new Poly((other.terms foldLeft ???)(addTerm))

def addTerm(terms: Map[Int, Double], term: (Int, Double)) =
    ???
```

Eas **Task**

Phone keys mnemonics

```
val mnemonics = Map(
    '2' -> "ABC", '3' -> "DEF",
    '4' -> "GHI", '5' -> "JKL",
    '6' -> "MNO", '7' -> "PQRS",
    '8' -> "TUV", '9' -> "WXYZ")
```

Assume you have a dictionary of words

design a method

translate(number)

that produces all phrases of words

eg "7225247386" should have

„Scala is fun"