First-class Functions

Scala has *first-class functions*.

- Not only can you define functions and call them,
- But you can write down functions as unnamed *literals* and then pass them around as *values*.

A simple example of a function literal that adds one to a number:

```
(x: Int) => x + 1
```

Function values are objects, so you can store them in variables if you like.

```
var increase = (x: Int) => x + 1
```

They are functions, too, so you can invoke them using the usual parentheses function-call notation.

```
scala> var increase = (x: Int) => x + 1
increase: (Int) => Int = <function1>

scala> increase(10)
res0: Int = 11
```

Function Values are Objects

Functions are values, and values are objects, so it follows that functions themselves are objects.

The function type S => T is equivalent to scala.Function1[S, T], where Function1 is defined as follows:

```
trait Function1[-S, +T] {
 def apply(x: S): T
}
```

So functions are interpreted as objects with apply methods.

For example, the function

(x: Int) => x + 1

is expanded to:

```
new Function1[Int, Int] {
  def apply(x: Int) =
    x + 1
}
```

Different Forms of Function Values

- A function value can be
  - A “=>” expression

```
(len => xs.take(len))
(lst => lst.take(3)
((lst, len) => lst.take(len))
```

  - Same with parameter types:

```
((len: Int) => xs.take(len))
```

  - Same with { ... }:

```
{ len =>
  println(“taking+len+”elems”)
  xs.take(len)
}
```

  - An expression with underscores

```
(xs.take(_))
(_ take 3)
(_ take _)
```

  - The name of a method:

```
xs.take
```

Higher-order Functions

Higher-order functions take function values as parameters or return them as results.

An example of a higher order function is the `filter` method. This method selects those elements of a collection that pass a test the user supplies That test is supplied using a function.

For example, the function `(x: Int) => x > 0` could be used for filtering.

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.filter((x: Int) => x > 0)
res4: List[Int] = List(5, 10)
```

Short forms of function literals

1) You can leave off the parameter types:

```
scala> someNumbers.filter((x) => x > 0)
res5: List[Int] = List(5, 10)
```

The Scala compiler knows that x must be an integer, because it sees that you are immediately using the function to filter a list of integers.

2) You can leave out parentheses around a parameter whose type is inferred.

```
scala> someNumbers.filter(x => x > 0)
res6: List[Int] = List(5, 10)
```

3) You can use underscores as placeholders for one or more parameters, so long as each parameter appears only one time within the function literal.

```
scala> someNumbers.filter(_ > 0)
res7: List[Int] = List(5, 10)
```

Partially Applied Methods

- In Scala, when you invoke a function, passing in any needed arguments, you *apply* that function *to* the arguments. For example, given the following function:

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
sum: (a: Int,b: Int,c: Int)Int
scala> sum(1, 2, 3)
res10: Int = 6
```

- A partially applied function is an expression in which you don't supply all of the arguments needed by the function. Instead, you supply some, or none, of the needed arguments.

```
scala> val a = sum _
a: (Int, Int, Int) => Int = <function3>
scala> a(1, 2, 3)
res11: Int = 6
scala> val b = sum(1, _: Int, 3)
b: (Int) => Int = <function1>
scala> b(2)
res13: Int = 6
```

Partially applied methods (2)

- However, try this:

```
scala> val x = sum
<console>:8: error: missing arguments for method sum;
follow this method with `_' if you want to treat it as a
partially applied function
       val x = sum
```

- Here, `sum` was not converted automatically to a function.
- (Why? Because forgetting arguments is quite common, so a silent conversion into a function value is often unintended.
- Methods are converted to function values only if the expected type of the expression is a function type.

Call By Name Arguments

- Function arguments are usually evaluated before the function call.
- There is one exception: If the function type starts with a `=>`, the argument is "*call by name*".
- This means that the argument is not evaluated when it is passed to the function.
- It is instead evaluated every time the parameter is referenced in the function.
- Every time can mean: never at all!

```
var checksEnabled = false
def check(cond: => Boolean) =
  if (checksEnabled && !cond)
    throw new AssertionError
```

Curried Functions

A curried function is applied to multiple argument lists, instead of just one.

A non-curried function:

```
scala> def plainOldSum(x: Int, y: Int) = x + y
plainOldSum: (Int,Int)Int
scala> plainOldSum(1, 2)
res4: Int = 3
```

A curried function

```
scala> def curriedSum(x: Int)(y: Int) = x + y
curriedSum: (Int)(Int)
Int scala> curriedSum(1)(2)
res5: Int = 3
```

When you invoke `curriedSum`, you get two function invocations back to back.

- The first function invocation takes a single Int parameter named x, and returns a function value for the second function.
- This second function takes the Int parameter y.

Higher-Order Functions on Lists

`xs foreach f`	applies function f to each list element, returns ().
`xs map f`	applies function f to each list element, returns list of results.
`xs flatMap p`	applies function f to each list element, concatenates the results. `xs flatMap p = (xs map f).flaten`
`xs filter p`	returns list of all elements for which p is true.
`xs exists p`	is there an element that satisfies p?
`xs forall p`	do all elements satisfy p?
`xs partition p`	same as (xs filter p, xs filter (!p(_)))
`xs takeWhile p`	the longest prefix of elements that satisfy p
`xs dropWhile p`	the rest of the list, starting with the first element that does not satisfy p
`xs span p`	same as (xs takeWhile p, xs dropWhile p)

Examples of map and flatMap

```
scala> List(1, 2, 3) map (_ + 1)
res32: List[Int] = List(2, 3, 4)

scala> val words = List("the", "quick", "brown", "fox")
words: List[java.lang.String] = List(the, quick, brown, fox)

scala> words map (_.length)
res33: List[Int] = List(3, 5, 5, 3)

scala> words map (_.toList)
res35: List[List[Char]] = List(List(t, h, e), List(q, u, i, c, k),
   List(b, r, o, w, n), List(f, o, x))

scala> words flatMap (_.toList)
res36: List[Char] = List(t, h, e, q, u, i, c, k, b, r, o, w,
n, f, o, x)
```

Examples of combination of map and flatMap

- list all combinations of numbers x and y where x is drawn from 1..5 and y is drawn from 1..3.

```
scala> List.range(1, 5) flatMap (i => List.range(1, 3) map (j => (i, j)))
res8: List[(Int, Int)] = List((1,1), (1,2), (2,1), (2,2), (3,1),
(3,2), (4,1), (4,2))
```

- This example shows how to construct a list of all pairs (*x*, *x*) such that 0<*y*<*x*<5:

```
scala> List.range(1, 5) flatMap ( i => List.range(1, i) map (j => (i, j)) )
res37: List[(Int, Int)] = List((2,1), (3,1), (3,2), (4,1), (4,2), (4,3))
```

Higher-Order Functions on Lists (2)

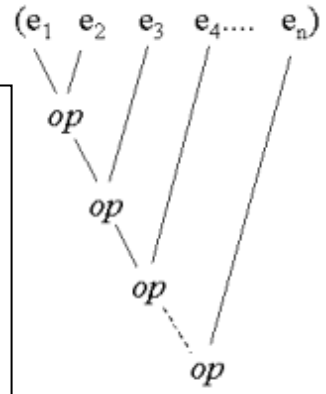
`xs reduceLeft op` Apply binary operation op between successive elements of non-empty collection xs, going left to right

`xs reduceRight op` Apply binary operation op between successive elements of non-empty collection xs, going right to left

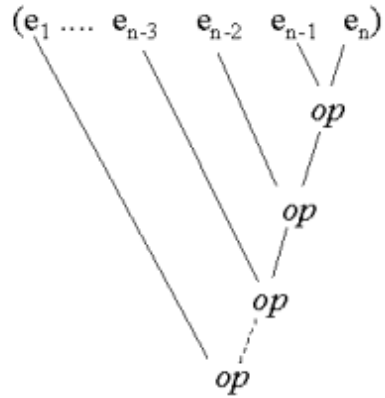
```
scala> val l = List (1, 2, 3, 4, 5, 6)
l: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> l.reduceLeft((x,y) =>(x+y))
res5: Int = 21
```

Or

```
scala> l.reduceLeft(_+_)
res6: Int = 21
```



`reduceLeft`



`reduceRight`

Exercises - 1

a. Use the `reduceLeft` function to calcuate the maximum value of a list of integers.

b. Evaluate `1.to(10).reduceLeft(_ * _)`.
What do you get? Write a function that computes n! in this way.

c. Now we'd like to compute 2^n with the same trick. How can you get a sequence of n copies of the number ? Hint: map.
What is your function that computes 2^n?

d. Given a `Seq[String]`, how can you use `reduceLeft` to concatenate them with spaces in between? Write a function `catSpace` that does this.
For example, `catSpace(Vector(”I", "have", "a", ”dream"))` should give a string ”I have a dream"

Exercises - 2

a. Given a matrix represented as a list of lists, write a function that returns the first, or an arbitrary column of the matrix.

```
def firstColumn(xs: List[List[Int]]): List[Int] = ???
def column(xs: List[List[Int]], col: Int): List[Int] = ???
```

b. Given a matrix represented as a list of lists, write a function that returns the diagnonal of the matrix.

```
def diagonal(xs: List[List[Int]]): List[Int] = ???
```

c. Given a matrix represented as a list of lists, write a function that checks whether the matrix has a row consisting only of zeroes:

```
def hasZeroRow(matrix: List[List[Int]]): Boolean = ???
```

Exercises - 3

a. Write a test whether a number is prime. A number n is prime if the only divisors of n are 1 and n itself. Keep it simple, efficiency is not important for now:

```
def isPrime(x: Int): Boolean = ???
```

b. Given list of strings, find all lines longer than a minimum length

```
def linesLonger(lines: List[String], len: Int): List[String]= ???
```

c. Given list of strings, find the length of the longest line

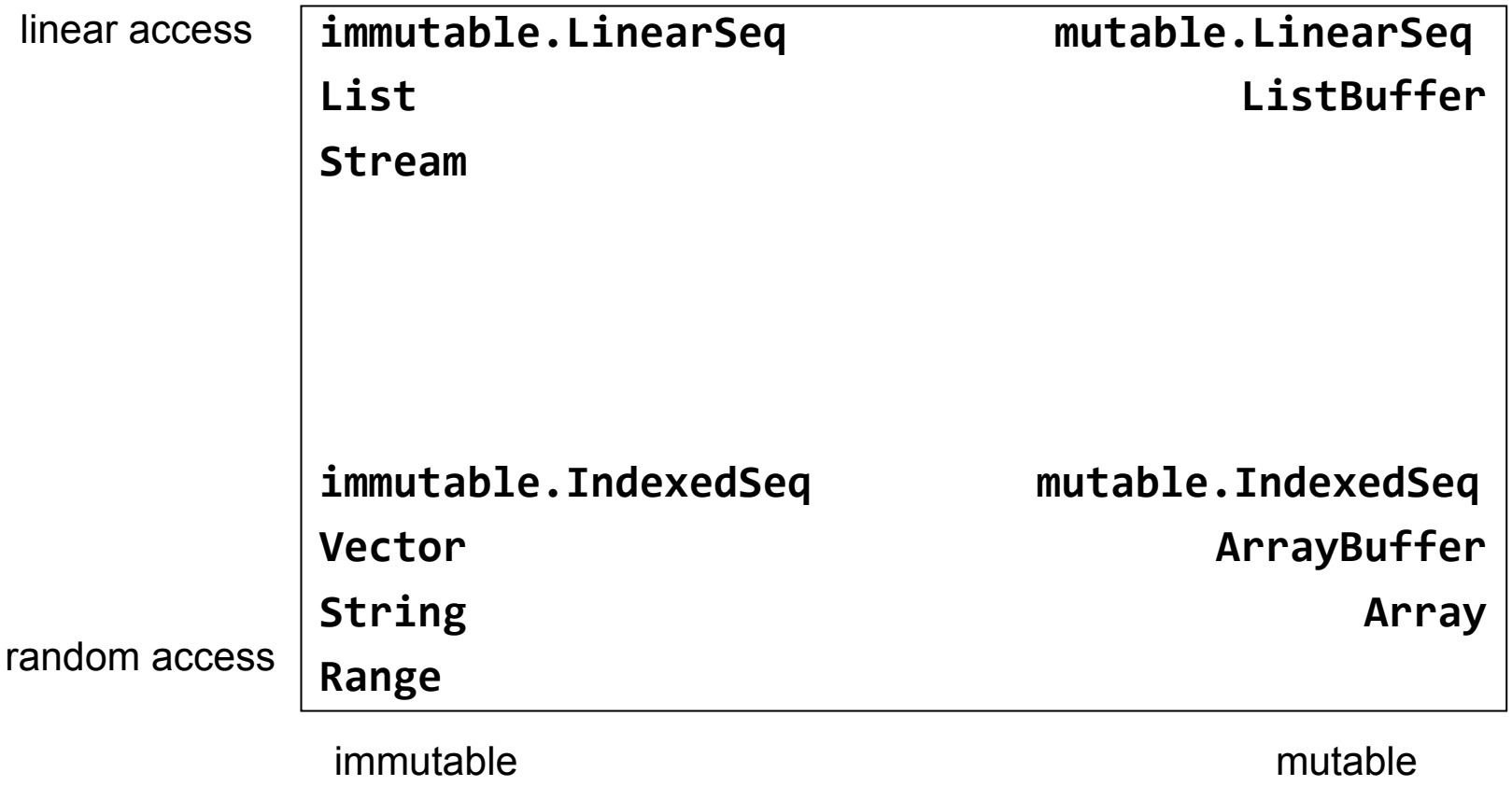
```
def longestLineLength(lines: List[String]): Int = ???
```

d. Given list of strings, eliminate all empty lines.

```
def elimEmptyLines(lines: List[String]): List[String] = ???
```

e. Given list of strings, find the longest line.

Other Kinds of Sequences



Addition and Modification Operations

Immutable:

`x +: xs`	Prepend element, yielding new sequence
`xs :+ x`	Append element, yielding new sequence
`xs ++ ys`	Concatenate two sequences
`xs updated (i, x)`	New sequence with some element changed

Mutable:

`xs(i) = x`,	Update one element
`xs.update(i, x)`	
`xs += x`	Append at end
`x +=: xs`	Prepend at beginning
`xs insert (i, x)`	Insert in middle

The Uniform X Principle

- All kinds of sequences support the same protocol methods that you have already seen (exception: `::`, and pattern matching is only for lists, and with `#::` for Streams)

```
scala> val v = Vector(1, 2, 3)
v: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

scala> v map (_ + 1)
res3: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4)

scala> val s = "abc"
s: java.lang.String = abc

scala> s map (_.toUpper)
res4: String = ABC
```

Ranges and their use in For Loops

A typical for loop:

```
for (x <- 0 until 10) println(x)
```

Here, ... <- .. is called a generator, which in this case goes over a range.

Ranges can be defined like this

```
0 until 10          0, 1, 2, 3, 4, 5, 6, 7, 8, 9
1 to 10             1, 2, 3, 4, 5, 6, 7, 8, 9, 10
1 to 10 by 2        1, 3, 5, 7, 9
```

One can also iterate directly over a sequence:

```
val xs = List(1, 2, 3)
for (x <- xs) println(x)
```

Sets

- Sets are collections without duplicated elements
- Elements in a set do not necessarily have a fixed order.
- Sets support an efficient `contains` method.
- New elements are added with +, removed with − (immutable), or with `+=, -=` (mutable).
- Sets support also most operations on sequences.

```
scala> val s = Set(1, 2, 2, 3)
s: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> s contains 2
res6: Boolean = true

scala> s + 4
res7: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)

scala> s map (_ * 2)
res8: scala.collection.immutable.Set[Int] = Set(2, 4, 6)
```

Maps

- Maps associate keys with values.

```
scala> val m = Map('a' -> 1, 'b' -> 2, 'c' -> 3)
m: scala.collection.immutable.Map[Char,Int] = Map(a -> 1, b -> 2, c -> 3)

scala> m + ('d' -> 4, 'a' -> 0)
res10: scala.collection.immutable.Map[Char,Int] = Map('a' -> 0, b -> 2,
c -> 3, d -> 4)
scala> m('a')
res11: Int = 1

scala> m get 'a'
res12: Option[Int] = Some(1)

scala> m get 'a' match { // pattern matching, will see later
     |   case Some(key) => key
     |   case None => -1
     | }
res13: Int = 1
```

Operations on Maps

`m + (k -> v)`	Add/overwrite key/value pair
`m + (k, v)`	
`m - k`	Remove key
`m(k), m.apply(k)`	Selection (key must be in map)
`m get k`	Selection returning Option result

For mutable maps:

`m += (k -> v)`	Destructive add/overwrite
`m -= k`	Destructive remove key