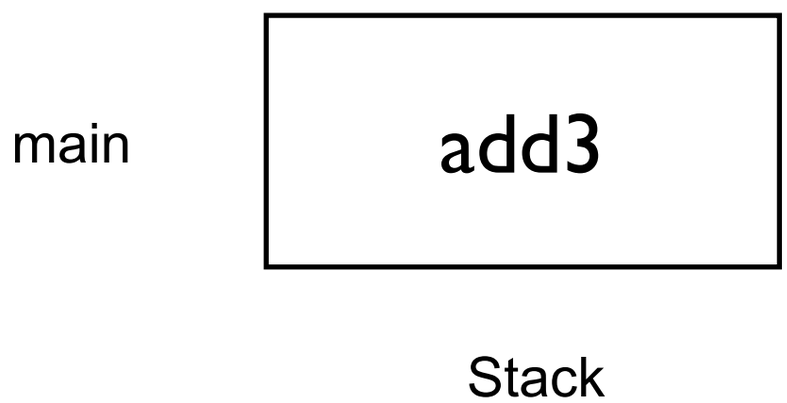
CS 696 Intro to Big Data: Tools and Methods
Fall Semester, 2017
Doc 4 Scala 2
Sep 5, 2017

Closure

```
def addN(n:Int):(Int => Int) = {
  def adder(k:Int):Int = {k + n}
  adder
}
```

```
var add3:(Int => Int) = null
```

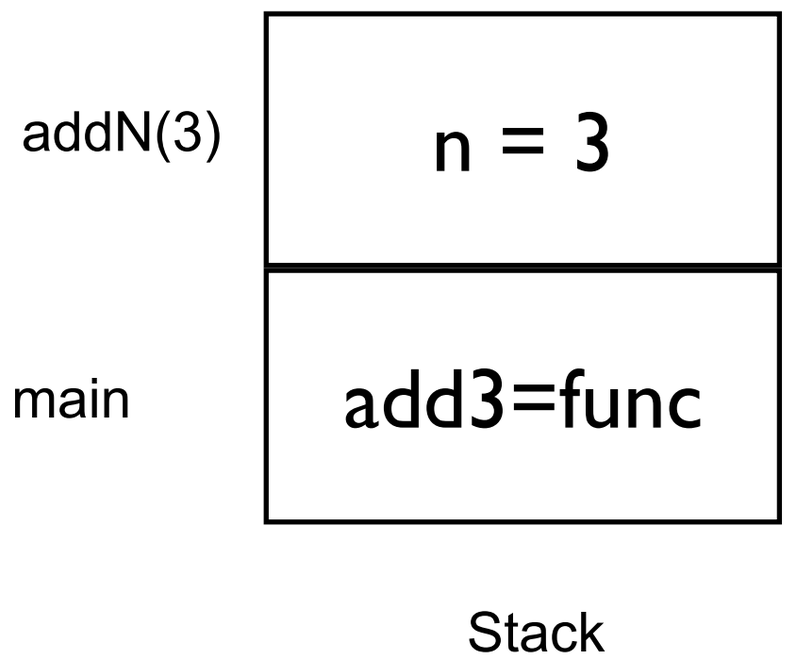
```
add3 = addN(3)

add3(2)
```



Closure

```
def addN(n:Int):(Int => Int) = {
  def adder(k:Int):Int = {k + n}
  adder
}
```

```
var add3:(Int => Int) = null

add3 = addN(3)

add3(2)
```

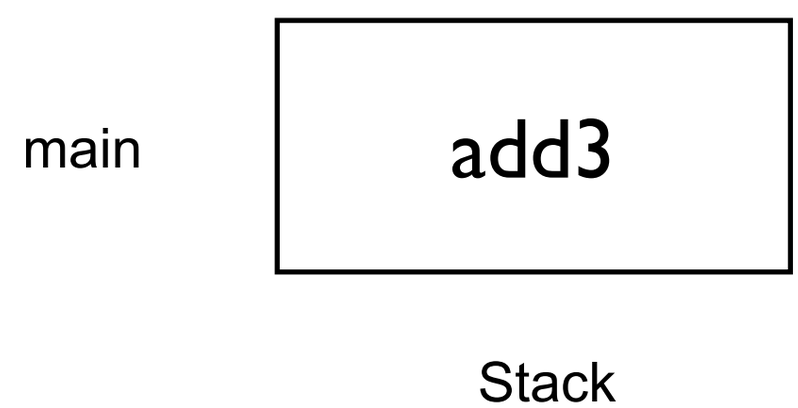


Closure

```
def addN(n:Int):(Int => Int) = {
  def adder(k:Int):Int = {k + n}
  adder
}
```

```
var add3:(Int => Int) = null

add3 = addN(3)

add3(2)
```

How does add3 access n?



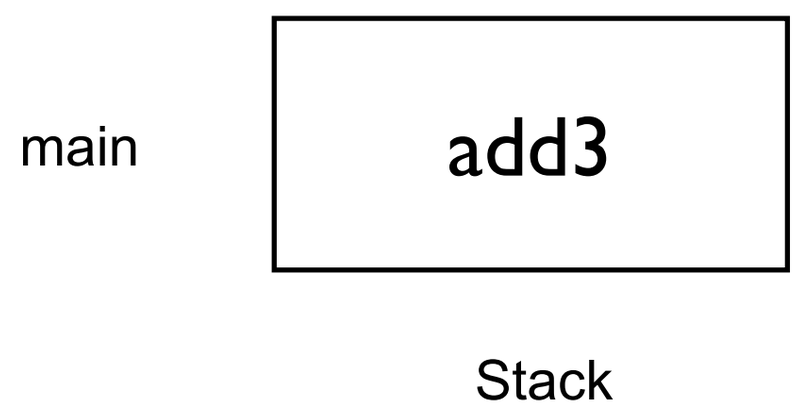
Closure

```
def addN(n:Int):(Int => Int) = {
  def adder(k:Int):Int = {k + n}
  adder
}
```

```
var add3:(Int => Int) = null

add3 = addN(3)

add3(2)
```

Closure

- Maintains data in creation environment
- Even when environment no longer exists



Partially Evaluated Functions

```
def sum(a: Int, b:Int, c:Int) = {
   println("Start")
   a + b + c
}

val partialSum = sum(1,_: Int,5)
println("Before call")
val result = partialSum(2)
println( result)
```

Output

```
Before call
Start
8
```

Partially Evaluated Functions

```
def sum(a: Int, b:Int, c:Int) = {
   println("Start")
   a + b + c
}

val partialSum = sum(_:Int,_: Int,5)
println("Before call")
val result = partialSum(1,2)
println( result)
```

Output

```
Before call
Start
8
```

Higher Order Functions

```
def sum(a: Int, b:Int, c:Int) =  a + b + c

def passSum(x: ((Int, Int, Int) => Int)): Int = {x(1, 2, 3)}

passSum(sum)


val newSum = sum _
newSum(1,2,3)

passSum(newSum)
```

Higher Order Function

Curried Functions

```
def curriedSum(x: Int)(y: Int) = x + y

val result = curriedSum(1)(2)

val partialSum = curriedSum(1)_
partialSum(2)

val noSumYet = curriedSum _
noSumYet(1)(2)
```

Scala Collections

Mutable

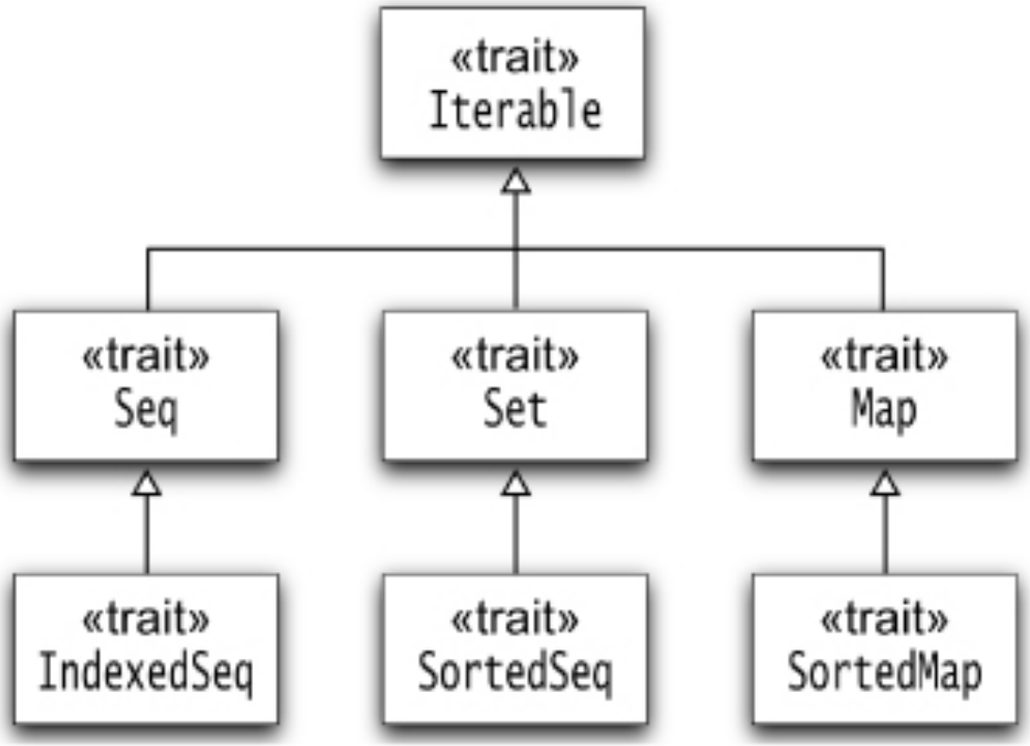
scala.collection.mutable

ArrayBuffer, StringBuilder
HashMap, HashSet
Stack, Queue, PriorityQueue

Immutable

scala.collection.immutable
scala.collection

Vector, List, Range, Stream
HashMap, HashSet, Stack, Queue



Array & ArrayBuilder

Array
- Indexed collection
- Fixed size
- Can change elements

ArrayBuffer
- Mutable
- Grows at end

Arrays

```
val numbers = new Array[Int](10)
numbers(0) = 42
numbers(10)                //exception thrown
numbers.length             // 10

val moreNumbers = Array(1,1,2,3,5,8)

moreNumbers.mkString("<", ";", ">")              // <1;1;2;3;5;8>

val five = moreNumbers(4)

val strings = Array("Hi", "Mom")

val mixed = Array(1, "test")
```

Arrays

```
val newArray = Array[String]("cat", "dog", "mouse")
val inferredType = Array("rat", "mat")
val both = newArray ++ inferredType
val mixedType = Array("cat", 1, true)
```

ArrayBuffer

```
import scala.collection.mutable.ArrayBuffer
val b = ArrayBuffer[Int]()
b += 1                                  // add an element
b.append(2)                             // add an element
b += (1, 2, 3, 5)                       // add multiple elements
b ++= Array(8, 13, 21)                  // add a collection
b(1)
b.length
b.toArray
```

Map, Filter, Reduce

Map

Applies an function to each element of a collection

Returns a new collection containing the result

Filter

Returns elements of a collection that make a boolean function true

Reduce

Combines elements of a collection into single value

Map

```
import scala.math._

val data = Array(1,2,3,4,5)

def inc(n:Int) = { n + 1 }
val plus1 = data.map(inc)                    // Array(2, 3, 4, 5, 6 )

val plus10 = data.map(_ + 10)                //Array(11, 12, 13, 14, 15)

val b = data.map(pow(_,2))                   // Array(1.0, 4.0, 9.0, 16.0, 25.0)

val c = data.map( x=> 2*x - 3)               // Array(-1, 1, 3, 5, 7)

val d = data.map { x => 4 * x - 2 * x + 5}   // Array(7, 9, 11, 13, 15)

"cat".map(_.toUpper)                         // “CAT”
```

Map Verses Loop

```
val data = Array(1,2,3,4,5)
val plus10 = data.map(_ + 10)
```

```
val data = Array(1,2,3,4,5)
val plus10 = ArrayBuffer[Int]()

for (k <- 0 to data.length)
  plus10 += 10 + data(k)
```

```
val data = Array(1,2,3,4,5)
val plus10 = ArrayBuffer[Int]()

for (k <- data)
  plus10 += 10 + k
```

Why This Matters

```
val data = Array(1,2,3,4,5)
val plus10 = data.map(_ + 10)
```

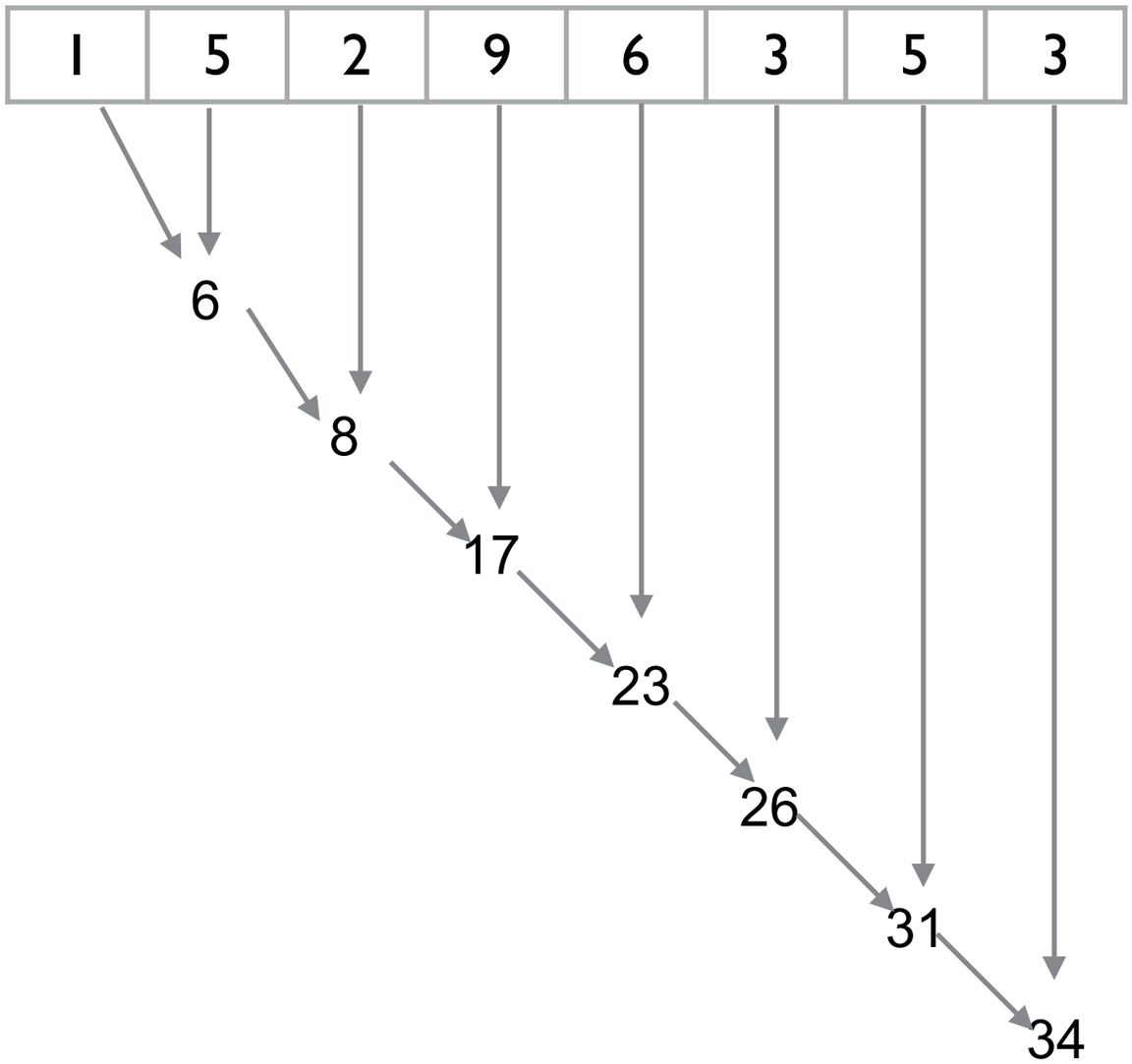
```
val data = Array(1,2,3,4,5)
val plus10 = ArrayBuffer[Int]()

for (k <- 0 to data.length)
  plus10 += 10 + data(k)
```

Less typing :)

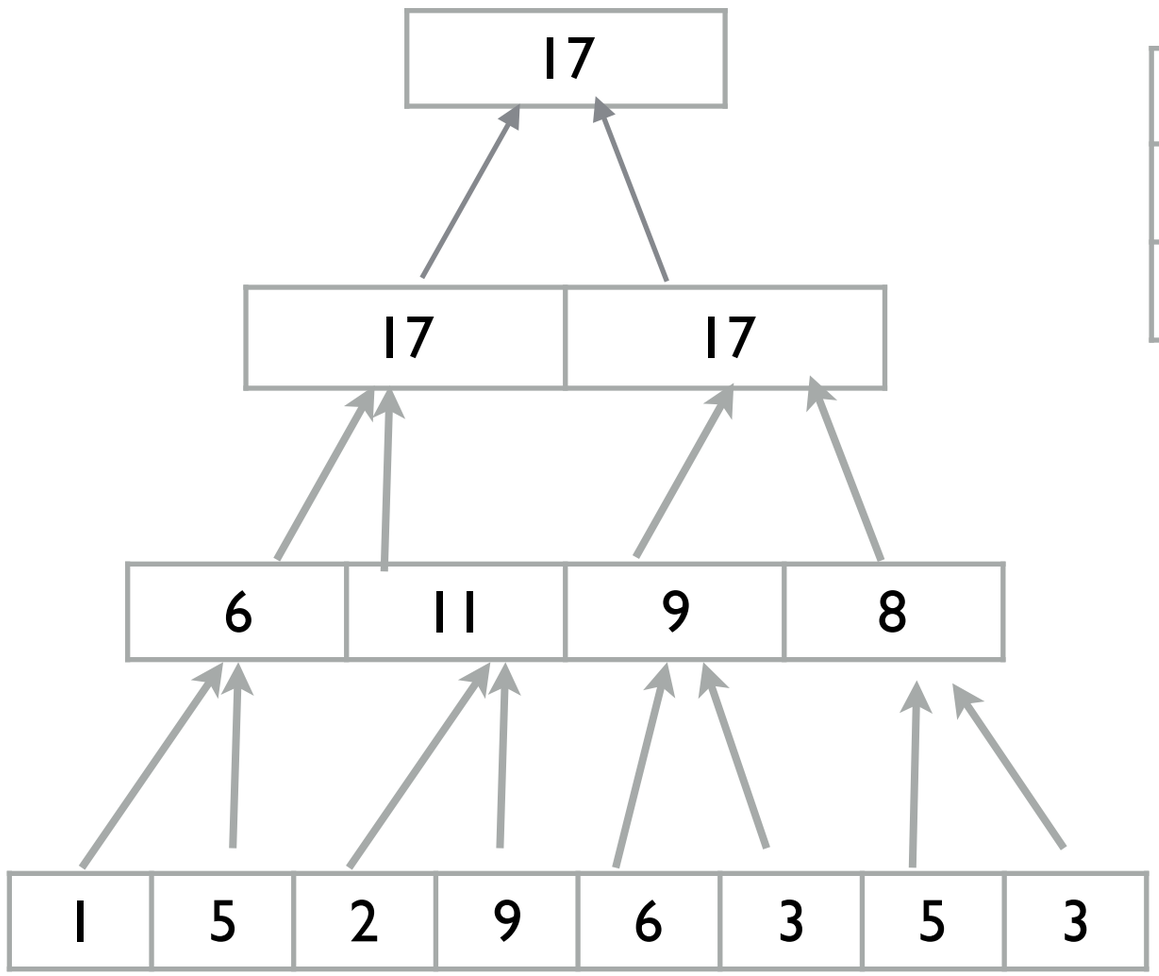
Low level details done for you
- Less boiler plate code
- Fewer mistakes

Library can optimize computation
- Standard optimizations
  - Unrolling loops
- Rearrange operations
- Parallelize or distribute computation for you

Normal Sum



Pairwise Sum



	Error Growth Rate
Normal Sum	$O(\varepsilon n)$
Pairwise Sum	$O(\varepsilon \log(n))$

ε = machine precision

n = number of floats to add

NumPy & Julia default sum - pairwise

When n < k use normal sum

Pairwise Sum verses Normal Sum (Julia)

```
function linear_sum(a)
  sum = 0.0
  for k in a
    sum += k
  end
  sum
end
```

```
tenth = fill(0.1,10_000_000)
```

	Result	Time - Seconds
sum(tenth) - builtin	999999.999999997	0.008
linear_sum(tenth) - above	999999.999838975	0.017
reduce(+, tenth) - builtin	999999.999999997	0.007

Scala

```
def usingLoop(ints:ArrayBuffer[Double]) = {
  val start = System.currentTimeMillis()
  var sum = 0.0
  for( x <- ints) {
    sum += x
  }
  val runtime = System.currentTimeMillis() - start
 runtime
}

def usingSum(ints:ArrayBuffer[Double]) = {
  val start = System.currentTimeMillis()
  var sum = ints.sum
  val runtime = System.currentTimeMillis() - start
  runtime
}

def usingReduce(ints:ArrayBuffer[Double]) = {
  val start = System.currentTimeMillis()
  var sum = ints.reduce(_ + _)
  val runtime = System.currentTimeMillis() - start
  runtime
}
```

Scala

	Result	Time - Seconds
Sum	999999.999838975	3.875
Loop	999999.999838975	0.070
Reduce	999999.999838975	0.093

Julia

	Result	Time - Seconds
sum(tenth) - builtin	999999.999999997	0.008
linear_sum(tenth) - above	999999.999838975	0.017
reduce(+, tenth) - builtin	999999.999999997	0.007

par

Produces parallel implementation of a collection
Parallelizes collection methods when possible

```
for (i <- (0 to 10).par) print(s" $i")
```

```
0 2 1 3 4 8 5 9 6 10 7
```

par

```
val integers = ArrayBuffer[Int]()

for (k <- (1 to 10000))
  integers += (random()*1000).toInt

integers.par
          .filter(_ % 2 == 0)
          .sum
```

find all odd integers and sum them in parallel

It doesn’t make sense to do this in parallel

Comunication c

Filter, FilterNot

```
val data = Array(1,2,3,4,5)

data.filter(_ > 3)                          // Array(4, 5)
data.filterNot( _ > 3)                      // Array(1, 2, 3)

data.filter( _ % 2 == 0)                // Array(2, 4)
data.filterNot( _ % 2 == 0)                 // Array(1, 3, 5)

def isVowel(c:Char) = {
  val vowels = Set('a', 'e', 'i' , 'o', 'u')
  vowels.contains(c)
}

"a cat in the hat".filter( isVowel )        // aaiea
```

Reduce

```
val data = Array(1,2,3,4,5)

data.reduce(_ + _)                                  // 15
data.reduce(_ * _)                                  // 120

data.reduce( (x, y) => { if ( x < y ) x  else y})      // 1
```

Reduce

```
val data = Array(1,2,3,4,5)
data.filterNot( _ % 2 == 0)
data.reduce(_ + _)
data.reduce(_ * _)

data.reduce( (x, y) => { if ( x < y ) x  else y})

data.reduce( (partial, x) => {partial + x})
```

Partition, GroupBy

```
val data = Array(1,2,3,4,5)

val x = data.partition( _ % 3 == 0)          // (Array(3), Array(1, 2, 4, 5))
x._1
x._2

val y = data.groupBy( _ % 3 )          Map(1 -> Array(1, 4), 2 -> Array(2, 5)
y(0)                                        0 -> Array(3))
y(1)
y(2)
```

And More

```
val data = Array(1,2,3,4,5)
```

data.head	res10: Int = 1
data.last	res11: Int = 5
data.init	res12: Array[Int] = Array(1, 2, 3, 4)
data.tail	res13: Array[Int] = Array(2, 3, 4, 5)
data.sum	res14: Int = 15
data.product	res15: Int = 120
data.min	res16: Int = 1
data.max	res17: Int = 5
data.count(_ > 2)	res18: Int = 3
data.takeWhile(_ < 3)	res19: Array[Int] = Array(1, 2)
data.takeWhile(_ > 2)	res20: Array[Int] = Array()
data.dropWhile(_ < 3)	res21: Array[Int] = Array(3, 4, 5)
data.take(2)	res22: Array[Int] = Array(1, 2)
data.drop(2)	res23: Array[Int] = Array(3, 4, 5)
data.takeRight(2)	res24: Array[Int] = Array(4, 5)
data.dropRight(2)	res25: Array[Int] = Array(1, 2, 3)

Chaining

```
val data = Array(1,2,3,4,5)
```

```
data.map( x => 2*x - 2)
      .filter( _ % 2 == 0)
      .sum
```

```
res0: Int = 20
```

Chaining & Multiple Passes

```
data.map( x => {println("map"); 2*x - 2})
      .filter( x => {println("filter"); x % 2 == 0})
      .sum
```

```
map
map
map
map
map
filter
filter
filter
filter
filter
x: Int = 20
```

Lazy Evaluation

```
data.view
    .map( x => {println("map"); 2*x - 2})
    .filter( x => {println("filter"); x % 2 == 0})
    .sum
```

```
map
filter
map
filter
map
filter
map
filter
map
filter
z: Int = 20
```

view

Returns a collection that is evaluated lazy

Spark operations are lazy