

Example 39

Tuples

Tuples

Tuples are not really collections, but
they fit with the theme

Tuples are ordered containers
of values of different types

Tuples

Tuples are ordered containers of values of different types

Creating Tuples

Accessing Individual Elements

Iterating Over Tuples

Passing Tuples to Functions

Creating Tuples

Tuples are simply values enclosed in parentheses

```
val personInfo = ("Vitthal", "Srinivasan", 36, "M")
```

```
personInfo: (String, String, Int, String) = (Vitthal,Srinivasan,36,M)
```

Pairs can be created with a special notation ->

```
val genderPair = "Vitthal" -> "M"
```

```
genderPair: (String, String) = (Vitthal,M)
```

Tuples

Tuples are ordered containers of values of different types

 Creating Tuples

Accessing Individual Elements

Iterating Over Tuples

Passing Tuples to Functions

Accessing Individual Elements

To access an element of a tuple based on its position inside the tuple

```
val personInfo = ("Vitthal", "Srinivasan", 36, "M")
```

```
[scala] personInfo._1  
res48: String = Vitthal
```

```
[scala] personInfo._2  
res49: String = Srinivasan
```

Accessing Individual Elements

To access an element of a tuple based on its position inside the tuple

```
val personInfo = ("Vitthal", "Srinivasan", 36, "M")
```

```
[scala] personInfo._1  
res48: String = Vitthal
```

```
[scala] personInfo._2  
res49: String = Srinivasan
```

Accessing Individual Elements

To access an element of a tuple based on its position inside the tuple

```
val personInfo = ("Vitthal", "Srinivasan", 36, "M")
```

```
[scala] personInfo._1  
res48: String = Vitthal
```

```
[scala] personInfo._2  
res49: String = Srinivasan
```

Accessing Individual Elements

To access an element of a tuple based on its position inside the tuple

```
val personInfo = ("Vitthal", "Srinivasan", 36, "M")
```

```
[scala] personInfo._1  
res48: String = Vitthal
```

```
[scala] personInfo._2  
res49: String = Srinivasan
```

Accessing Individual Elements

To access an element of a tuple using variables

```
val personInfo = ("Vitthal", "Srinivasan", 36, "M")
```

```
scala> val (firstName, lastName, age, gender) = personInfo
```

```
firstName: String = Vitthal
lastName: String = Srinivasan
age: Int = 36
gender: String = M
```

Accessing Individual Elements

If you care about some tuple elements but not others

```
val personInfo = ("Vitthal", "Srinivasan", 36, "M")
```

```
val (firstName, lastName, _, gender) = personInfo
```

firstName: String = Vitthal

lastName: String = Srinivasan

gender: String = M

Tuples

Tuples are ordered containers of values of different types

 Creating Tuples

Iterating Over
Tuples

 Accessing Individual
Elements

Passing Tuples
to Functions

Iterating Over Tuples

*There is a rather clunky syntax:
productIterator.foreach that takes a function literal*

```
personInfo.productIterator.foreach{ i => println("Value = "+i)}
```

```
Value = Vitthal  
Value = Srinivasan  
Value = 36  
Value = M
```

Iterating Over Tuples

```
val personInfo = ("Vitthal", "Srinivasan", 36, "M")  
personInfo: (String, String, Int, String) = (Vitthal,Srinivasan,36,M)
```

```
[scala> personInfo.productArity  
res56: Int = 4
```

The number of elements: **productArity**

Iterating Over Tuples

```
val personInfo = ("Vitthal", "Srinivasan", 36, "M")  
personInfo: (String, String, Int, String) = (Vitthal,Srinivasan,36,M)
```

```
[scala] personInfo.productArity  
res56: Int = 4
```

The number of elements: **productArity**

Tuples

Tuples are ordered containers of values of different types

 Creating Tuples

 Accessing Individual Elements

 Iterating Over Tuples

Passing Tuples to Functions

Passing Tuples to Functions

Call the `.tupled()` method on the function object

```
val genderPair = "Vitthal" -> "M"
```

```
def printPersonGender(name:String, gender:String) =  
  println(s"Name:$name M/F:$gender")
```

```
(printPersonGender _).tupled(genderPair)
```

Passing Tuples to Functions

Call the `.tupled()` method on the function object

```
val genderPair = "Vitthal" -> "M"
```

```
def printPersonGender(name:String, gender:String) =  
  println(s"Name:$name M/F:$gender")
```

```
(printPersonGender _).tupled(genderPair)
```

Passing Tuples to Functions

Call the `.tupled()` method on the function object

```
val genderPair = "Vitthal" -> "M"
```

```
def printPersonGender(name:String, gender:String) =  
  println(s"Name:$name M/F:$gender")
```

```
(printPersonGender _).tupled(genderPair)
```

Passing Tuples to Functions

Call the `.tupled()` method on the function object

```
val genderPair = "Vitthal" -> "M"
```

```
def printPersonGender(name:String, gender:String) =  
  println(s"Name:$name M/F:$gender")
```

```
(printPersonGender _).tupled(genderPair)
```

Passing Tuples to Functions

Call the `.tupled()` method on the function object

```
val genderPair = "Vitthal" -> "M"
```

```
def printPersonGender(name:String, gender:String) =  
  println(s"Name:$name M/F:$gender")
```

```
(printPersonGender _).tupled(genderPair)
```

Passing Tuples to Functions

Call the `.tupled()` method on the function object

```
val genderPair = "Vitthal" -> "M"
```

```
def printPersonGender(name:String, gender:String) =  
  println(s"Name:$name M/F:$gender")
```

```
(printPersonGender _).tupled(genderPair)
```

Passing Tuples to Functions

Call the `.tupled()` method on the function object

```
val genderPair = "Vitthal" -> "M"
```

```
def printPersonGender(name:String, gender:String) =  
  println(s"Name:$name M/F:$gender")
```

```
(printPersonGender _).tupled(genderPair)
```

Passing Tuples to Functions

Call the `.tupled()` method on the function object

```
val genderPair = "Vitthal" -> "M"
```

```
def printPersonGender(name:String, gender:String) =  
  println(s"Name:$name M/F:$gender")
```

```
(printPersonGender _).tupled(genderPair)
```

Name:Vitthal M/F:M

Passing Tuples to Functions

Call the `.tupled()` method on the function object

```
val genderPair = "Vitthal" -> "M"
```

```
def printPersonGender(name:String, gender:String) =  
  println(s"Name:$name M/F:$gender")
```

```
(printPersonGender _).tupled(genderPair)
```

```
Name:Vitthal M/F:M
```

Tuples

Tuples are ordered containers of values of different types

 Creating Tuples

 Accessing Individual Elements

 Iterating Over Tuples

 Passing Tuples to Functions

Tuples

Tuples are not really collections, but
they fit with the theme

Tuples are ordered containers
of values of different types

Tuples are not really collections

Collections in Scala (and Java) all
derive from the interface `Iterable`

Tuples, on the other hand, adhere
to traits `Tuple1, Tuple2, ...`

(More on traits later - not a biggie!)

Tuples are not really collections

Oh, and btw, tuples are
immutable

Example 40

Lists: Creating Them

Lists: Creating Them

A List is a list

A List is an immutable,
singly linked list

The end of the linked list is indicated by the
special value **Nil**

Nil is basically an empty list - it is a singleton
object of type `List[Nothing]`

A common way to create lists involves the
“cons” operator `::` and the special value **Nil**

A common way to create lists involves the “cons” operator :: and the special value Nil

Nil is basically an empty list - it is a singleton object of type List[Nothing]

The “cons” (construct) operator :: is right associative - it starts from the extreme right and works to the left

```
val weekDays2 = "Mon" :: "Tue" :: "Wed" :: "Thu" :: "Fri" :: Nil  
  
List[String] = List(Mon, Tue, Wed, Thu, Fri)
```

The “cons” (construct) operator :: is right associative - it starts from the extreme right and works to the left

```
val weekDays2 = "Mon" :: "Tue" :: "Wed" :: "Thu" :: "Fri" :: Nil
```

```
List[String] = List(Mon, Tue, Wed, Thu, Fri)
```

cons takes a list (initially Nil) and tacks on 1 element (initially Fri) to the head of that list

And the chain above continues until the entire list is set up

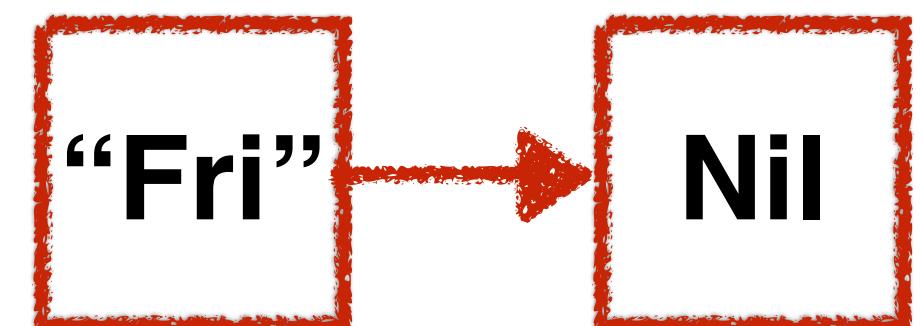
cons takes a list (initially Nil) and tacks on 1 element (initially Fri) to the head of that list

```
val weekDays2 = "Mon" :: "Tue" :: "Wed" :: "Thu" :: "Fri" :: Nil
```

Nil

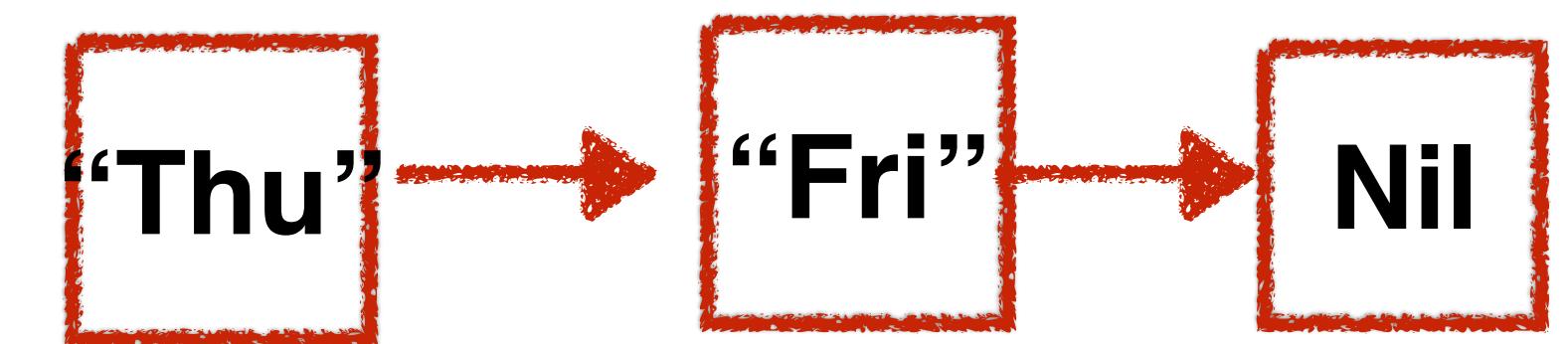
cons takes a list (initially Nil) and tacks on 1 element (initially Fri) to the head of that list

```
val weekDays2 = "Mon" :: "Tue" :: "Wed" :: "Thu" :: "Fri" :: Nil
```



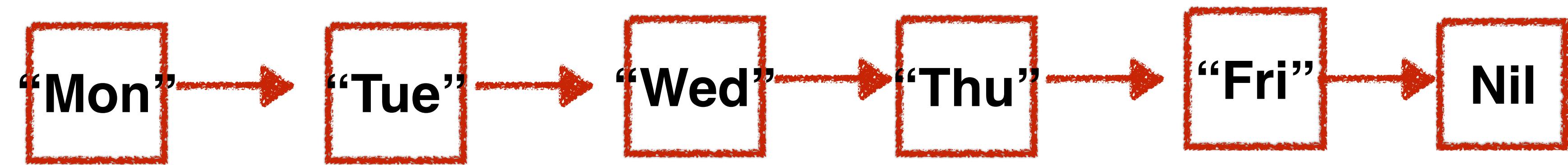
cons takes a list (initially Nil) and tacks on 1 element (initially Fri) to the head of that list

```
val weekDays2 = "Mon" :: "Tue" :: "Wed" :: "Thu" :: "Fri" :: Nil
```



cons takes a list (initially Nil) and tacks on 1 element (initially Fri) to the head of that list

```
val weekDays2 = "Mon" :: "Tue" :: "Wed" :: "Thu" :: "Fri" :: Nil
```



```
List[String] = List(Mon, Tue, Wed, Thu, Fri)
```

**And the chain above continues until
the entire list is set up**

The “cons” (construct) operator :: is right associative - it starts from the extreme right and works to the left

```
val weekDays2 = "Mon" :: "Tue" :: "Wed" :: "Thu" :: "Fri" :: Nil
```

```
List[String] = List(Mon, Tue, Wed, Thu, Fri)
```

cons takes a list (initially Nil) and tacks on 1 element (initially Fri) to the head of that list

And the chain above continues until the entire list is set up

A List is an immutable,
singly linked list

A common way to create lists involves the
“cons” operator :: and the special value **Nil**

An equivalent way is to use the List
“constructor” (actually called class parameters
in Scala)

A common way to create lists involves the “cons” operator :: and the special value `Nil`

An equivalent way is to use class parameters

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

```
List[String] = List(Mon, Tue, Wed, Thu, Fri)
```

```
val weekEndDays = List("Sat", "Sun")
```

```
List[String] = List(Sat, Sun)
```

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

```
val weekEndDays = List("Sat", "Sun")
```

Given 2 lists, they can be concatenated in 2 ways

```
val days = weekDays :::: weekendDays
```

```
val days = weekDays ++ weekendDays
```

```
days: List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

Given 2 lists, they can be concatenated in 2 ways

```
val days = weekDays :::
```

```
val days = weekDays ++ weekendDays
```

```
days: List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

So, what's the difference between
++ and :: : ?

Given 2 lists, they can be concatenated in 2 ways

```
val days = weekDays :::
```

```
val days = weekDays ++ weekendDays
```

So, what's the difference between
++ and ::?:

::: can only be used with lists, while ++
works with any Traversable

So, what's the difference between
++ and ::?

:: can only be used with lists, while ++
works with any Traversable

:: is right-associative, btw, as is ::

Given 2 lists, to create a list of tuples of the corresponding elements, use

zip

```
val allDays = List("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
```

```
val dayIndices = List(1, 2, 3, 4, 5, 6, 7)
```

```
dayIndices zip allDays
```

```
val allDays = List("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
val dayIndices = List(1, 2, 3, 4, 5, 6, 7)
```

Given 2 lists, to create a list of tuples of
the corresponding elements, use

zip

dayIndices zip allDays

```
List[(Int, String)] = List((1,Mon), (2,Tue), (3,Wed), (4,Thu),
(5,Fri), (6,Sat), (7,Sun))
```

Given 2 lists, to create **a list of tuples** of
the corresponding elements, use

zip

List[(Int, String)] = List((1,Mon), (2,Tue), (3,Wed), (4,Thu),
(5,Fri), (6,Sat), (7,Sun))

Given 2 lists, to create a list of tuples of
the corresponding elements, use

zip

List[(Int, String)] = List((1,Mon), (2,Tue), (3,Wed), (4,Thu),
(5,Fri), (6,Sat), (7,Sun))

Given 2 lists, to create a list of tuples of the corresponding elements, use

zip

```
val allDays = List("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
```

```
val dayIndices = List(1, 2, 3, 4, 5, 6, 7)
```

```
dayIndices zip allDays
```

Given a list of lists, to
combine into 1 list use

flatten

```
val daysAgain = List(weekDays, weekendDays).flatten  
List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

Given a **list of lists**, to
combine into 1 list use
flatten

```
val daysAgain = List(weekDays, weekendDays).flatten  
List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

```
List[List[String]] = List(List(Mon, Tue, Wed, Thu, Fri),  
List(Sat, Sun))
```

Given a **list of lists**, to
combine into 1 list use
flatten

```
val daysAgain = List(weekDays, weekendDays).flatten  
List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

```
List[List[String]] = List(List(Mon, Tue, Wed, Thu, Fri),  
List(Sat, Sun))
```

Given a **list of lists**, to
combine into 1 list use
flatten

```
val daysAgain = List(weekDays, weekendDays).flatten  
List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

```
List[List[String]] = List(List(Mon, Tue, Wed, Thu, Fri),  
List(Sat, Sun))
```

Given a **list of lists**, to
combine into 1 list use
flatten

```
val daysAgain = List(weekDays, weekendDays).flatten  
List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

```
List[List[String]] = List(List(Mon, Tue, Wed, Thu, Fri),  
List(Sat, Sun))
```

Given a list of lists, to
combine into 1 list use
flatten

```
val daysAgain = List(weekDays, weekendDays).flatten  
List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

Given a **list of lists**, to
combine into 1 list use
flatten

```
val daysAgain = List(weekDays, weekendDays).flatten  
List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

Given a list of lists, to
combine into 1 list use

flatten

```
val daysAgain = List(weekDays, weekendDays).flatten  
List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

Example 41

Lists: Using Them

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.head returns the first element

```
weekDays.head  
String = Mon
```

.tail returns the rest of the list

```
weekDays.tail  
List[String] = List(Tue, Wed, Thu, Fri)
```

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.size returns the size:-)

```
weekDays.size
```

```
Int = 5
```

.reverse reverses the list:-)

```
weekDays.reverse
```

```
List[String] = List(Fri, Thu, Wed, Tue, Mon)
```

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.contains tests for a specific element

```
weekDays.contains("Mon") // infix dot notation
```

```
weekDays contains "Mon" // operator notation
```

```
Boolean = true
```

// infix dot notation

weekDays . contains("Mon")

// operator notation

weekDays contains "Mon"

// Same result

Boolean = true

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

for loops work as usual

```
for (c <- weekDays) println(c)
```

```
Mon  
Tue  
Wed  
Thu  
Fri
```

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

while loops need to be terminated correctly

```
var rest0fWeek = weekDays
while (! rest0fWeek.isEmpty)
{
    println(s"Grr.. today is ${rest0fWeek.head}, ${rest0fWeek.size}
days left for the weekend")
    rest0fWeek = rest0fWeek.tail
}
```

while loops need to be terminated correctly

```
var restOfWeek = weekDays
while (! restOfWeek.isEmpty)
{
    println(s"Grr..today is ${restOfWeek.head}, ${restOfWeek.size}
days left for the weekend")
    restOfWeek = restOfWeek.tail
}
```

while loops need to be terminated correctly

```
var restOfWeek = weekDays
while (! restOfWeek.isEmpty)
{
    println(s"Grr..today is ${restOfWeek.head}, ${restOfWeek.size}
days left for the weekend")
    restOfWeek = restOfWeek.tail
}
```

while loops need to be terminated correctly

```
var rest0fWeek = weekDays
while (! rest0fWeek.isEmpty)
{
    println(s"Grr..today is ${rest0fWeek.head}, ${rest0fWeek.size}
days left for the weekend")
rest0fWeek = rest0fWeek.tail
}
```

while loops need to be terminated correctly

```
var rest0fWeek = weekDays
while (! rest0fWeek.isEmpty)
{
  println(s"Grr..today is ${rest0fWeek.head}, ${rest0fWeek.size} days left for the weekend")
  rest0fWeek = rest0fWeek.tail
}
```

Grr..today is Mon, 5 days left for the weekend
Grr..today is Tue, 4 days left for the weekend
Grr..today is Wed, 3 days left for the weekend
Grr..today is Thu, 2 days left for the weekend
Grr..today is Fri, 1 days left for the weekend

while loops need to be terminated
correctly

`while (! restOfWeek.isEmpty)`

is equivalent to

`while (restOfWeek != Nil)`

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Mon")
```

.distinct returns distinct
values :-)

```
weekDays.distinct // infix dot notation
```

```
List[String] = List(Mon, Tue, Wed, Thu)
```

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.drop returns a new list with fewer elements

```
weekDays drop 2 // infix dot notation
```

```
List[String] = List(Wed, Thu, Fri)
```

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.slice returns a new list, subset
of the old one

```
weekDays slice (2, 4)           // operator notation
```

```
List[String] = List(Wed, Thu)
```

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.splitAt returns 2 new lists

```
weekDays.splitAt(2) // infix dot notation
```

```
(List[String], List[String]) =  
(List(Mon, Tue), List(Wed, Thu, Fri))
```

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.take returns a new list starting at the head

```
weekDays take 2 // operator notation
```

```
List[String] = List(Mon, Tue)
```

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

. sorted returns a new list
sorted in “natural” order

```
weekDays.sorted // infix dot notation
```

```
List[String] = List(Fri, Mon, Thu, Tue, Wed)
```

Given a list

. sum

. product

. min

. max

All do what you would expect them to!

Example 42

Lists: Higher Order Functions

Lists: Higher Order Functions

Recall that

Higher order functions are functions take in other functions (as arguments), or return functions

Higher order functions on collections are very important in functional programming

Lists: Higher Order Functions

foreach

map

reduce

filter

partition

sortBy

fold

scan

Lists: Higher Order Functions

All these functions, in common, take a function and apply it to a list

map functions yield a new list

reduce functions shrink lists into value

Lists: Higher Order Functions

foreach

map

reduce

filter

partition

sortBy

fold

scan

foreach

Given a list

.foreach takes a procedure

and applies it to each element in the list

foreach

Given a list

no return value, i.e.
foreach is a statement

.foreach takes a **procedure**

and applies it to each element in the list

**NO RETURN VALUE, I.E.
FOREACH IS A STATEMENT**

foreach

Given a list

no return value, i.e.
foreach is a statement

.foreach takes a **procedure**

and applies it to each element in the list

foreach

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.foreach takes a procedure

```
println(_)
```

and applies it to each element in the list

```
weekDays.foreach(println(_))
```

foreach

.foreach takes a procedure

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

```
weekDays.foreach(println(_))
```

and applies it to each element in the list

```
Mon  
Tue  
Wed  
Thu  
Fri
```

foreach

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.foreach takes a procedure

```
val printValue = (x:Any) => {println(x)}:Unit
```

and applies it to each element in the list

```
weekDays.foreach(printValue)
```

foreach

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.foreach takes a procedure

```
val printValue = (x:Any) => {println(x)}:Unit
```

and applies it to each element in the list

```
weekDays.foreach(printValue)
```

foreach

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.foreach takes a procedure

```
val printValue = (x:Any) => {println(x)}:Unit
```

and applies it to each element in the list

```
weekDays.foreach(printValue)
```

Lists: Higher Order Functions

 **foreach** **map** **reduce**

filter **partition**

sortBy **fold** **scan**

map

Given a list

.map takes a function

and applies it to each element in the list

map

Given a list there is a return value,
i.e. map returns a list

.map takes a function

and applies it to each element in the list

THERE IS A RETURN VALUE,
I.E. MAP RETURNS A LIST

map

Given a list there is a return value,
i.e. map returns a list

.map takes a function

and applies it to each element in the list

map

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.map takes a function

_ == "Mon"

and applies it to each element in the list

```
weekDays.map(_ == "Mon")
```

map

.map takes a function

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")  
weekDays.map(_ == "Mon")
```

and applies it to each element in the list

```
List[Boolean] = List(true, false, false, false, false)
```

map

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.map takes a function

```
val IsManicMonday = (x:String) => {x == "Mon"}:Boolean
```

and applies it to each element in the list

```
weekDays.map(IsManicMonday)
```

map

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.map takes a function

```
val IsManicMonday = (x:String) => {x == "Mon"}:Boolean
```

and applies it to each element in the list

```
weekDays.map(IsManicMonday)
```

map

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.map takes a function

```
val IsManicMonday = (x:String) => {x == "Mon"}:Boolean
```

and applies it to each element in the list

```
weekDays.map(IsManicMonday)
```

map

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.map takes a function

```
val IsManicMonday = (x:String) => {x == "Mon"}:Boolean
```

and applies it to each element in the list

```
weekDays.map(IsManicMonday)
```

Lists: Higher Order Functions

 **foreach**

 **map**

reduce

filter

partition

sortBy

fold

scan

filter

Given a list

.filter takes a predicate

and returns each element in the list that
satisfies the predicate

filter

Given a list

.filter takes a

A function that
returns true or false

predicate

and returns each element in the list that
satisfies the predicate

filter

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.filter takes a predicate

_ != "Mon"

and returns each element in the list that
satisfies the predicate

```
weekDays.filter(_ != "Mon")
```

filter

.filter takes a predicate

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")  
weekDays.filter(_ != "Mon")
```

and returns each element in the list that
satisfies the predicate

```
List[String] = List(Tue, Wed, Thu, Fri)
```

filter

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.filter takes a predicate

```
val IsNotManicMonday = (x:String) => {x != "Mon"}:Boolean
```

and returns each element in the list that
satisfies the predicate

```
weekDays.filter(IsNotManicMonday)
```

filter

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.filter takes a predicate

```
val IsNotManicMonday = (x:String) => {x != "Mon"}:Boolean
```

and returns each element in the list that
satisfies the predicate

```
weekDays.filter(IsNotManicMonday)
```

filter

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.filter takes a predicate

```
val IsNotManicMonday = (x:String) => {x != "Mon"}:Boolean
```

and returns each element in the list that
satisfies the predicate

```
weekDays.filter(IsNotManicMonday)
```

Lists: Higher Order Functions

 **foreach**

 **map**

reduce

 **filter**

partition

sortBy

fold

scan

partition

Given a list

.partition takes a predicate

and returns 2 lists - elements that satisfy, and those that don't

partition

Given a list

A function that
returns true or false

- partition takes a

predicate

and returns 2 lists - elements that satisfy,
and those that don't

partition

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

• **partition takes a predicate**

_ != "Mon"

and returns 2 lists - elements that satisfy, and those that don't

```
weekDays.partition(_ != "Mon")
```

partition

- **partition takes a predicate**

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")  
weekDays.partition(_ != "Mon")
```

and returns 2 lists - elements that satisfy, and those that don't

```
(List[String], List[String]) = (List(Tue, Wed, Thu,  
Fri), List(Mon))
```

partition

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.partition takes a predicate

```
val IsNotManicMonday = (x:String) => {x != "Mon"}:Boolean
```

and returns 2 lists - elements that satisfy, and those that don't

```
weekDays.partition(IsNotManicMonday)
```

partition

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.partition takes a predicate

```
val IsNotManicMonday = (x:String) => {x != "Mon"}:Boolean
```

and returns 2 lists - elements that satisfy,
and those that don't

```
weekDays.partition(IsNotManicMonday)
```

partition

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

.partition takes a predicate

```
val IsNotManicMonday = (x:String) => {x != "Mon"}:Boolean
```

and returns 2 lists - elements that satisfy, and those that don't

```
weekDays.partition(IsNotManicMonday)
```

Lists: Higher Order Functions

 **foreach**

 **map**

reduce

 **filter**

 **partition**

sortBy

fold

scan

sortBy

Given a list

.sortBy takes a function

and sorts the list elements based on it

sortBy

Given a list

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

. sortBy takes a function

$_()$

and sorts the list elements based on it

```
weekDays.sortBy(_())
```

sortBy

. **sortBy** takes a function

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")  
weekDays.sortBy(_(0))
```

and sorts the list elements based on it

```
List[String] = List(Fri, Mon, Tue, Thu, Wed)
```

sortBy .map takes a function

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")  
weekDays.sortBy(_(0))
```

and sorts the list elements based on it

```
List[String] = List(Fri, Mon, Tue, Thu, Wed)
```

sortBy

.**sortBy** takes a function

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")  
weekDays.sortBy(_(0))
```

and sorts the list elements based on it

```
List[String] = List(Fri, Mon, Tue, Thu, Wed)
```

Lists: Higher Order Functions

 **foreach**

 **map**

reduce

 **filter**

 **partition**

 **sortBy**

fold

scan

Lists: Higher Order Functions

reduce

fold

scan

More on these in a bit!

Example 43

Scan, ScanLeft,
ScanRight

Scan, ScanLeft, ScanRight

Given a list

.scan , .scanLeft , .scanRight
take a function

And apply that function in rather
complicated ways

Scan, ScanLeft, ScanRight

Given a list

```
val someNumbers = List(10,20,30,40,50,60)
```

. scan , . scanLeft , . scanRight
take a function

— — —
And apply that function in rather
complicated ways

Scan, ScanLeft, ScanRight

apply that function in rather complicated ways

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers.scanRight(0)(_ - _)
```

```
List[Int] = List(-30, 40, -20, 50, -10, 60, 0)
```

```
someNumbers.scanLeft(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

```
someNumbers.scan(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

Notice that all 3 functions have 2 parameter groups

someNumbers.scanRight(0)(_ - _)

someNumbers.scanLeft(0)(_ - _)

someNumbers.scan(0)(_ - _)

The second is always the function to be applied to the list - called the **reduction function**

someNumbers.scanRight(0) (_ - _)

someNumbers.scanLeft(0) (_ - _)

someNumbers.scan(0) (_ - _)

The second is always the function to be applied
to the list - called the **reduction function**

The reduction function will be applied to
elements in the list..

scanLeft is left associative, **scanRight**
is right associative

The second is always the function to be applied to the list - called the **reduction function**

someNumbers.scanRight(0) (_ - _)

someNumbers.scanLeft(0) (_ - _)

someNumbers.scan(0) (_ - _)

The first is the initial value for the first application of the reduction function

someNumbers.scanRight **(0)** (_ - _)

someNumbers.scanLeft **(0)** (_ - _)

someNumbers.scan **(0)** (_ - _)

Scan, ScanLeft, ScanRight
apply that function in rather complicated ways

```
val someNumbers = List(10,20,30,40,50,60)
```

Let's take these one by one

```
someNumbers.scanRight(0)(_ - _)
```

```
List[Int] = List(10, 20, 30, 40, 50, 60, 0)
```

```
someNumbers.scan(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers.scanRight(0)(_ - _)
```

```
List[Int] = List(-30, 40, -20, 50, -10, 60, 0)
```

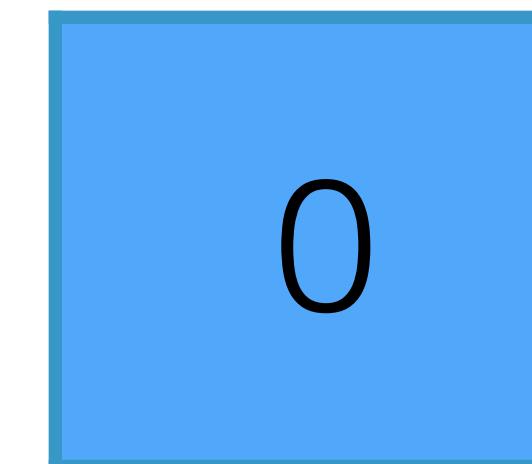
ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

```
someNumbers.scanRight(0)(_ - _)
```



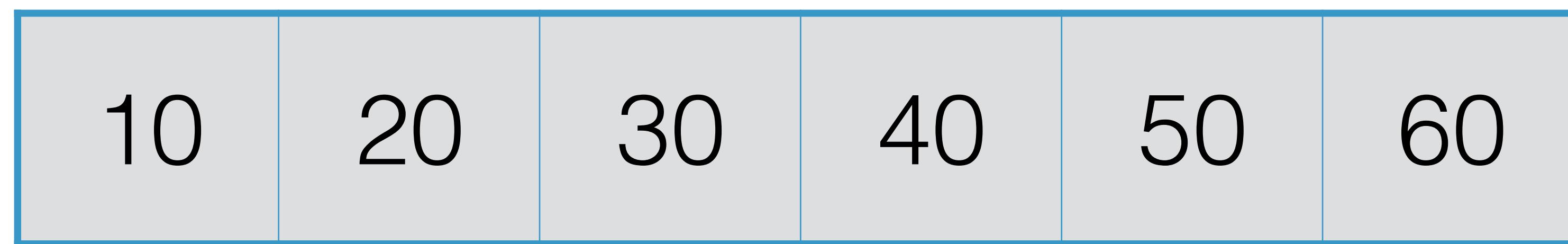
Initial Value

Function (a - b)

ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



Initial Value

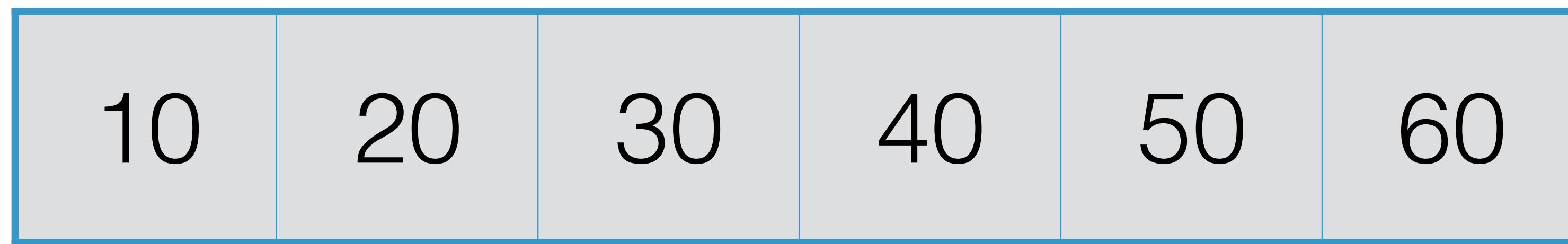
Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



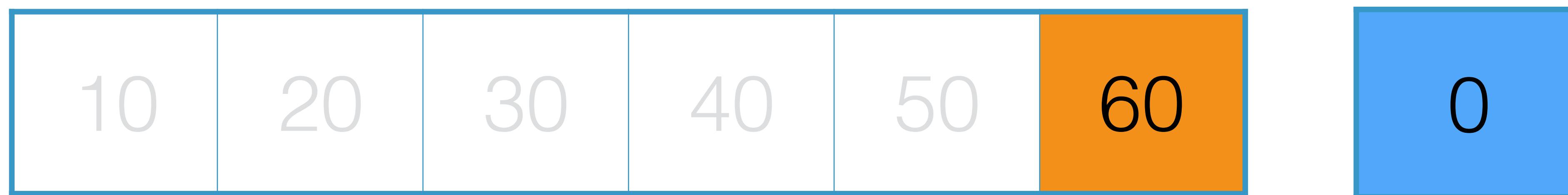
Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



$$60 - 0 = 60$$

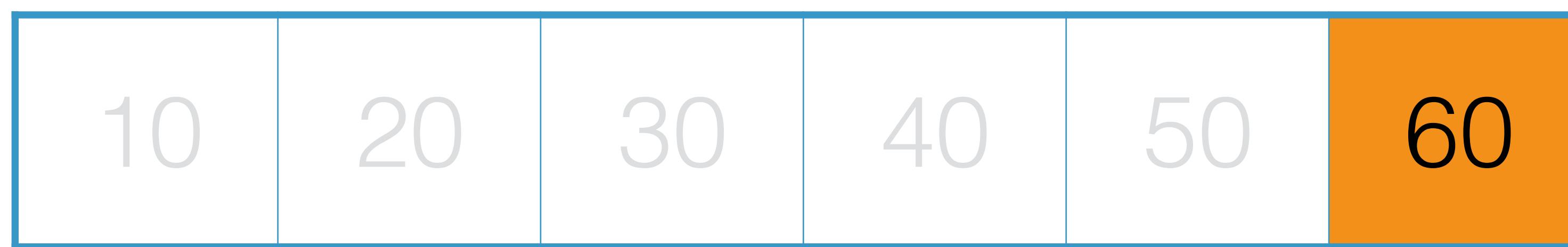
Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



Initial Value

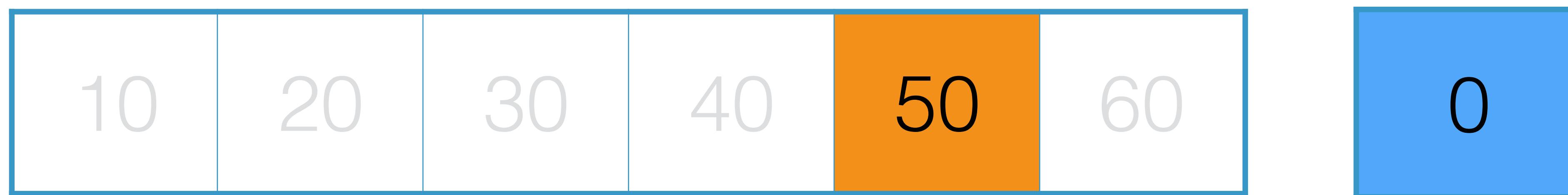
Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



$$50 - 60 = -10$$

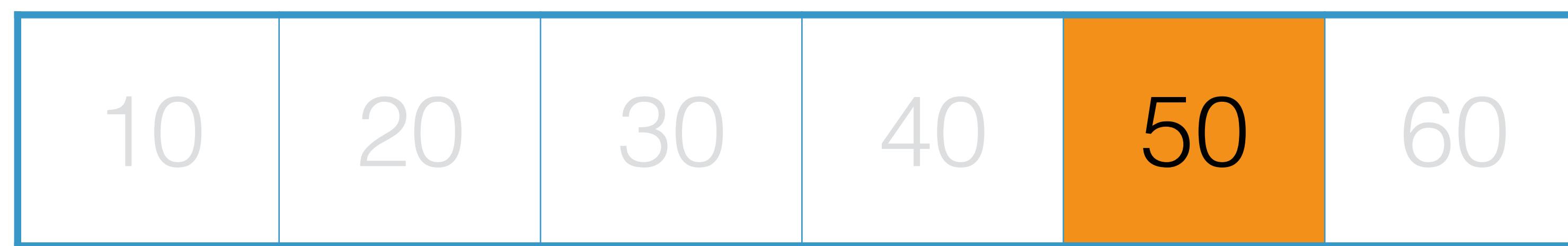
Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



Initial Value

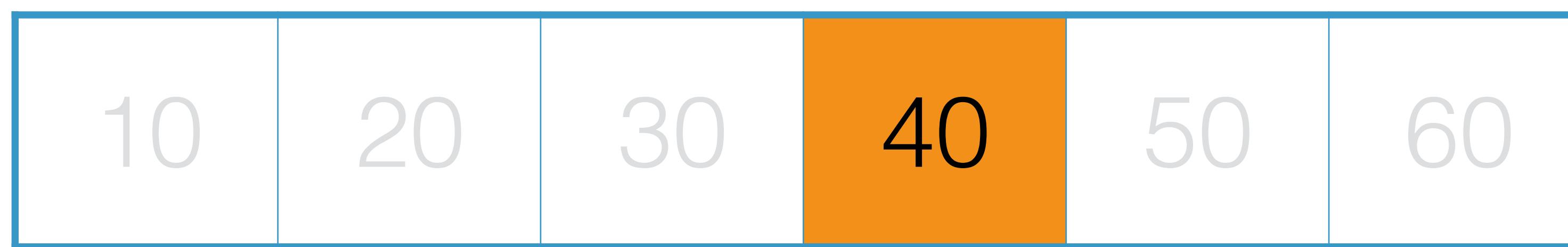
Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



Initial Value

0

$$40 - (-10) = 50$$

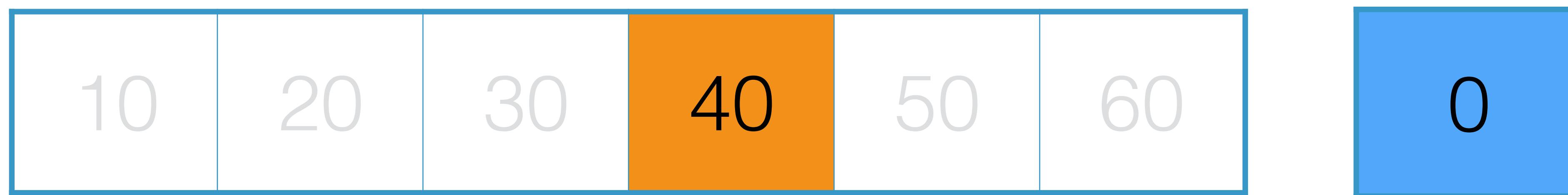
Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



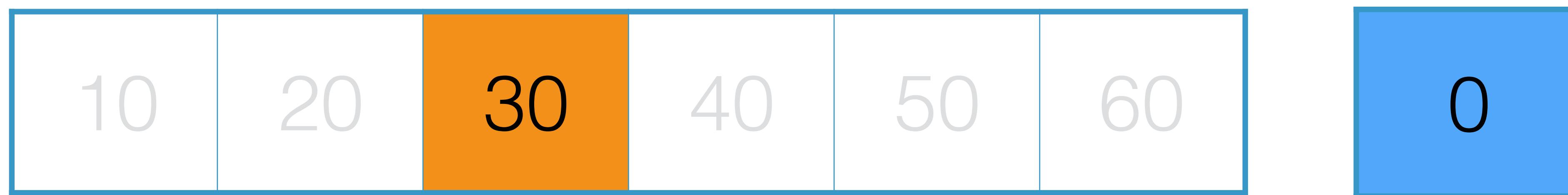
Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



$$30 - 50 = -20$$

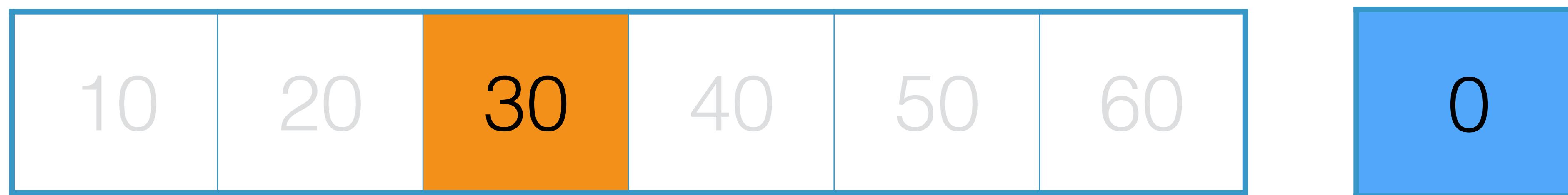
Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



Result

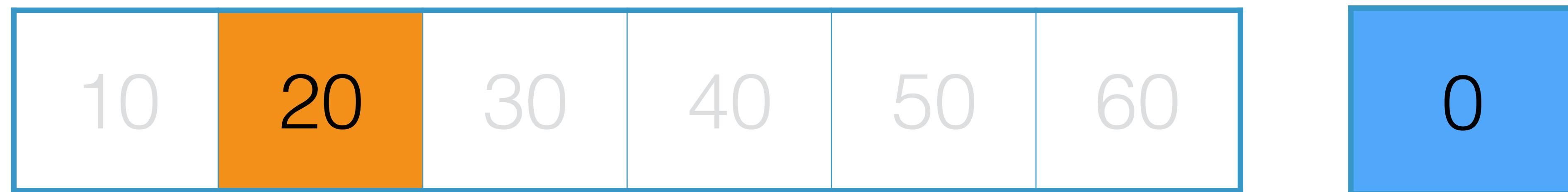


ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

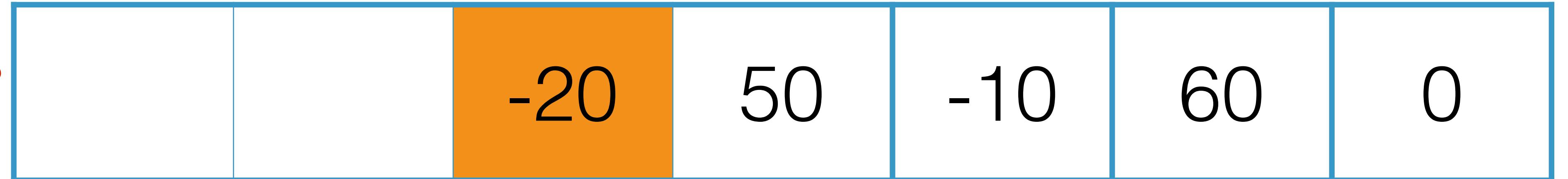
List

Initial Value



$$20 - (-20) = 40$$

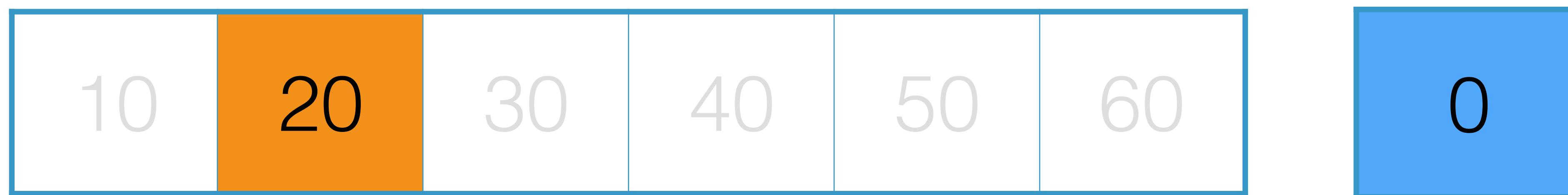
Result



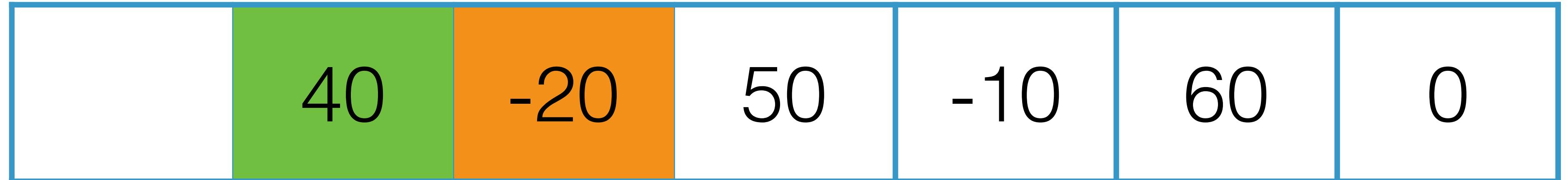
ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



Result

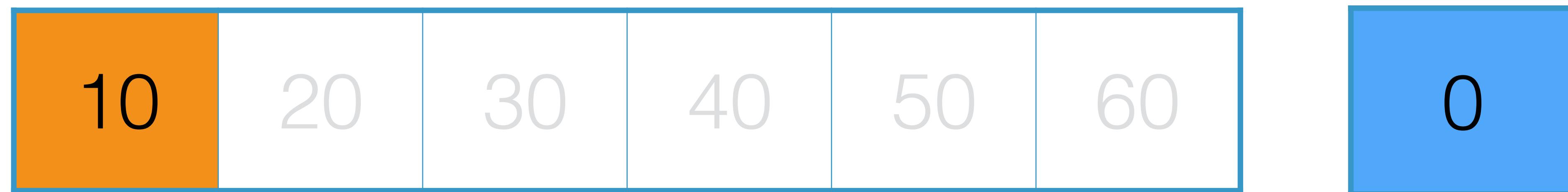


ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

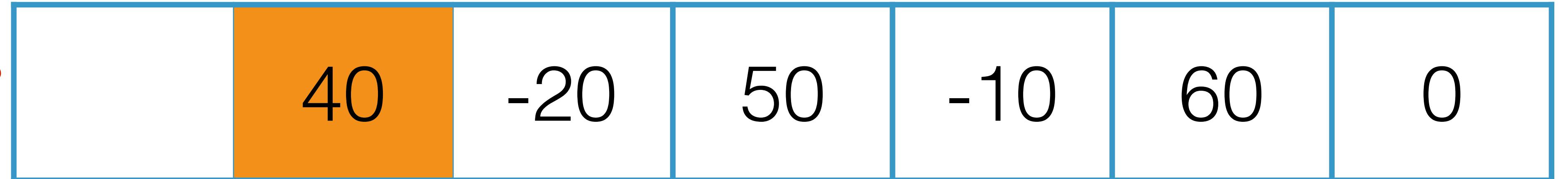
List

Initial Value



$$10 - 40 = -30$$

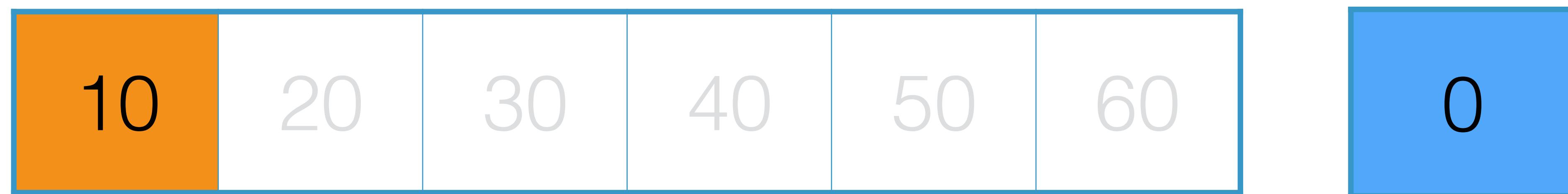
Result



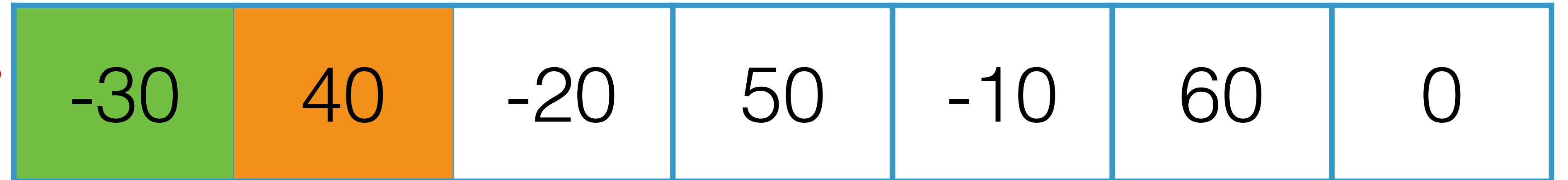
ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



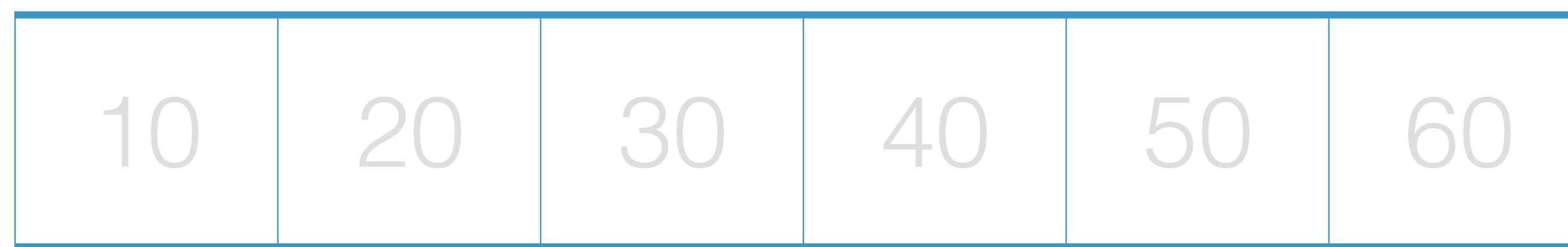
Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

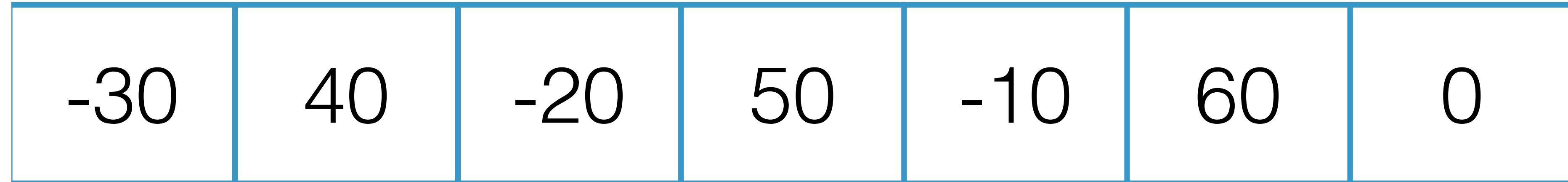
List



Initial Value

0

Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```



```
someNumbers.scanRight(0)(_ - _)
```

```
List[Int] = List(-30, 40, -20, 50, -10, 60, 0)
```

Result

-30	40	-20	50	-10	60	0
-----	----	-----	----	-----	----	---

Scan, ScanLeft, ScanRight

apply that function in rather complicated ways

```
val someNumbers = List(10,20,30,40,50,60)
```



```
someNumbers.scanRight(0)(_ - _)
```

```
List[Int] = List(-30, 40, -20, 50, -10, 60, 0)
```

```
someNumbers.scanLeft(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

```
someNumbers.scan(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

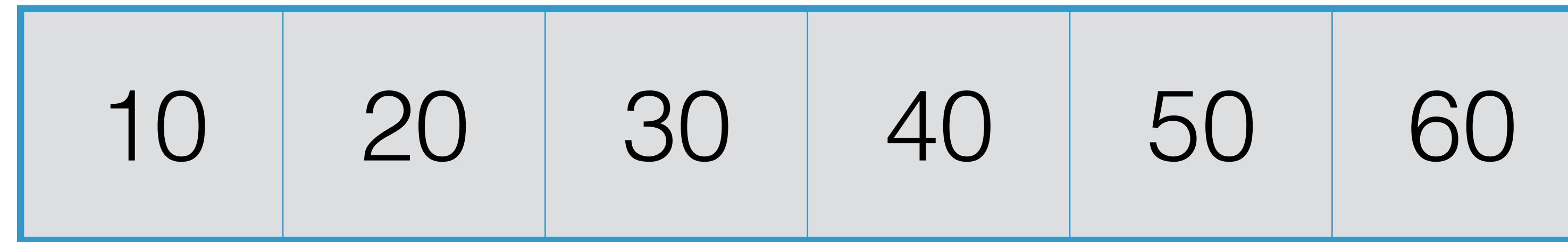
```
someNumbers.scanLeft(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

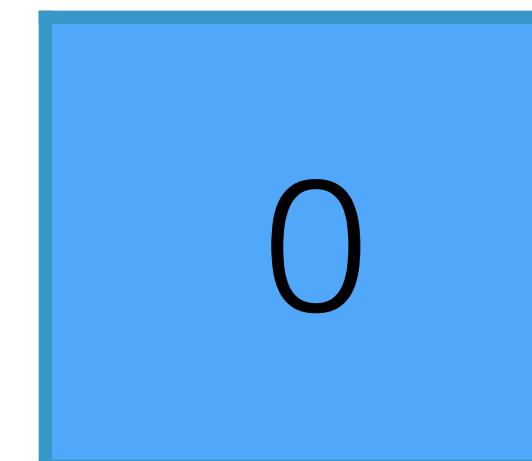
Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

List



```
someNumbers.scanLeft(0)(_ - _)
```



Initial Value

Function (a - b)

Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value

0

List

10	20	30	40	50	60
----	----	----	----	----	----



Result

--	--	--	--	--	--

Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value

0

List

10	20	30	40	50	60
----	----	----	----	----	----

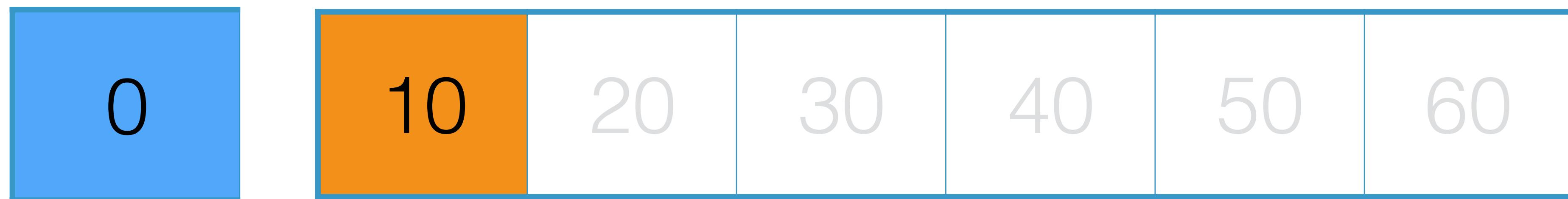
Result

0					
---	--	--	--	--	--

Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value



List

$$0 - 10 = -10$$

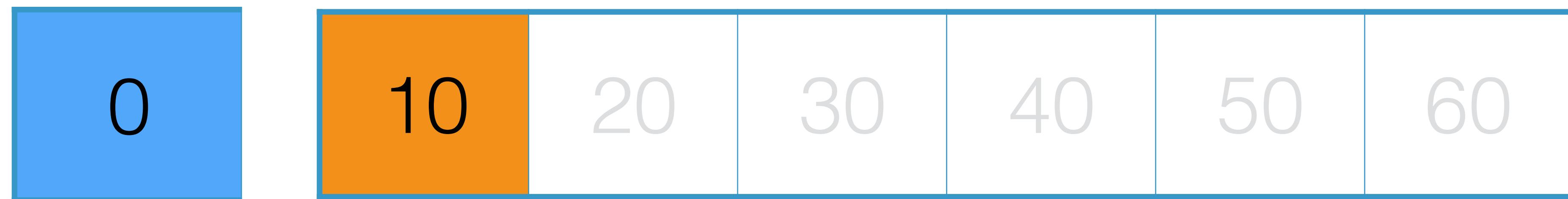
Result



Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value



List

Result

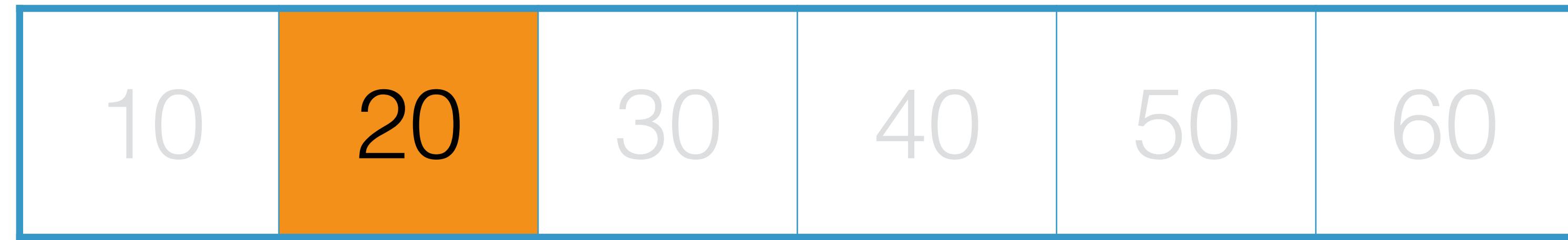


Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value

0



List

$$-10 - 20 = -30$$

Result



Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value

0

10 20 30 40 50 60

List

Result

0 -10 -30

Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value

0

List

10	20	30	40	50	60
----	----	----	----	----	----

$$-30 - 30 = -60$$

Result

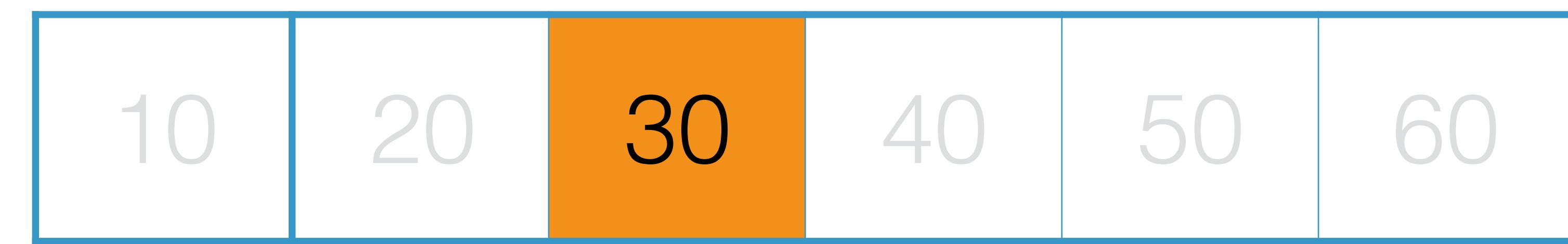
0	-10	-30			
---	-----	-----	--	--	--

Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value

0



List

Result

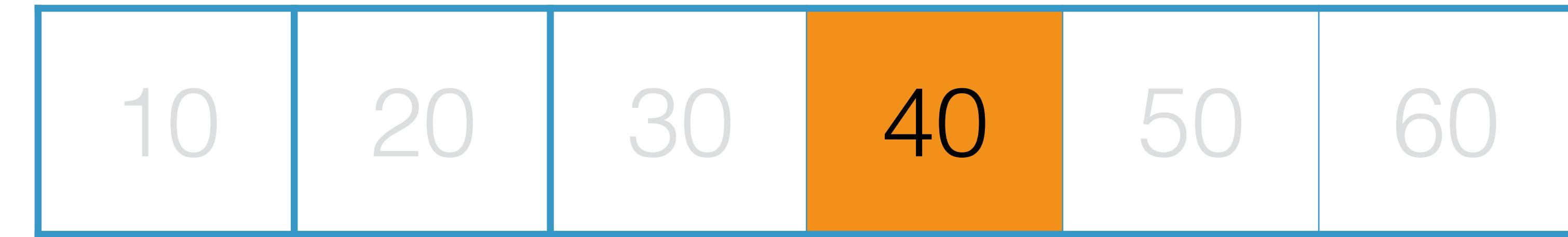


Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value

0



List

$$-60 - 40 = -100$$

Result

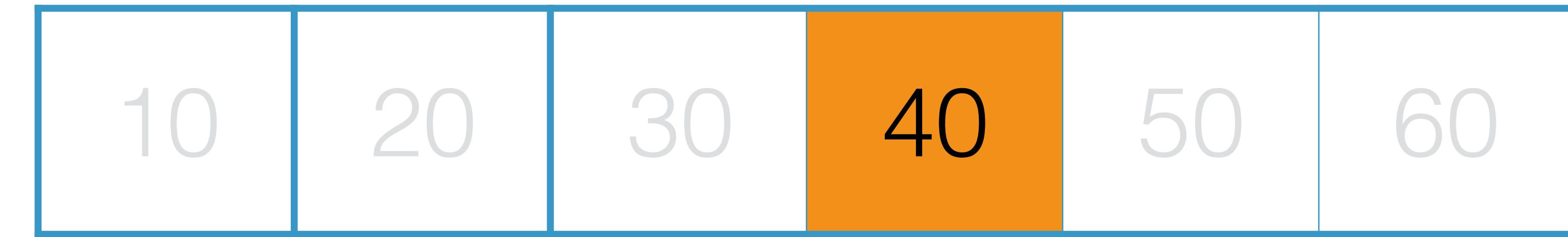


Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value

0



List

Result

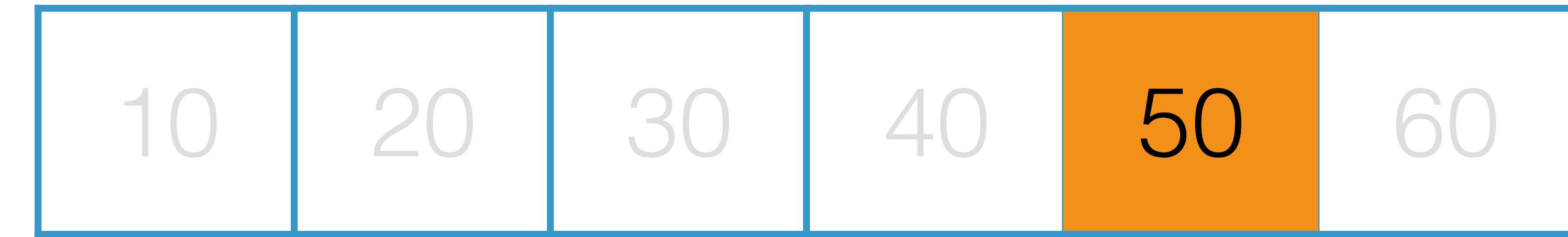


Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

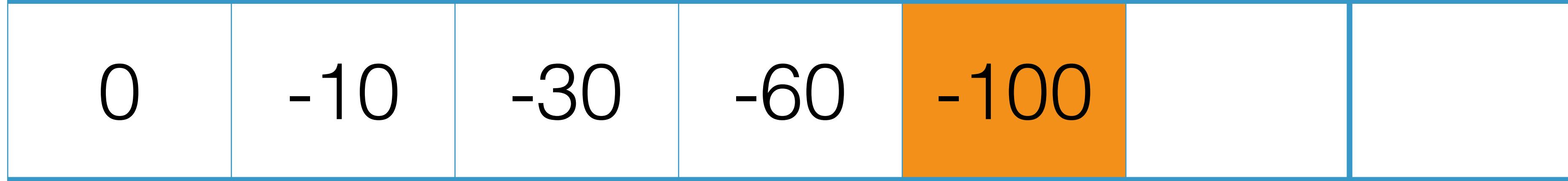
Initial Value

0



$$-100 - 50 = -150$$

Result

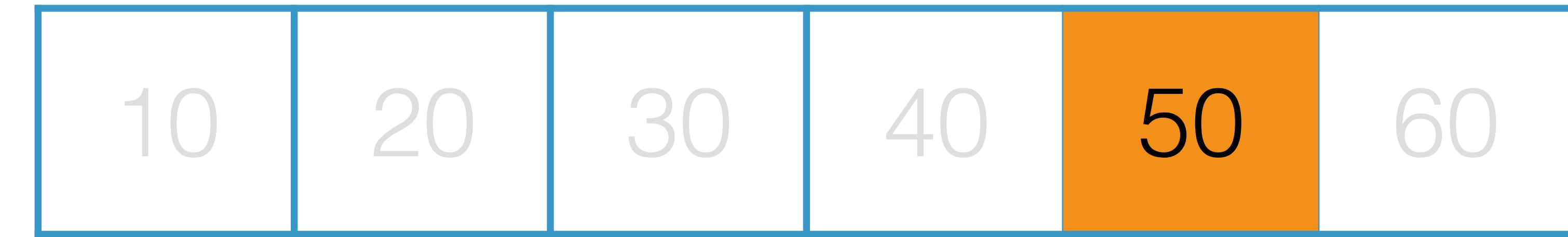


Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

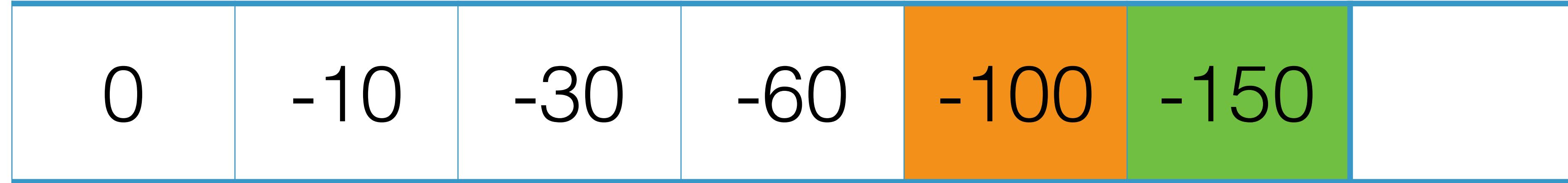
Initial Value

0



List

Result

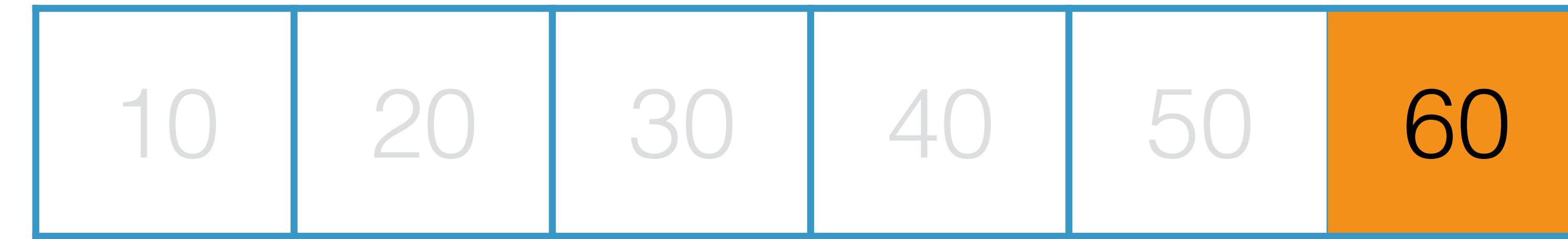


Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value

0



$$-150 - 60 = -210$$

Result

0	-10	-30	-60	-100	-150	
---	-----	-----	-----	------	------	--

Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value

0

List

10	20	30	40	50	60
----	----	----	----	----	----

Result

0	-10	-30	-60	-100	-150	-210
---	-----	-----	-----	------	------	------

Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```

Initial Value

0

List

10	20	30	40	50	60
----	----	----	----	----	----

Result

0	-10	-30	-60	-100	-150	-210
---	-----	-----	-----	------	------	------

Scanleft

```
val someNumbers = List(10,20,30,40,50,60)
```



```
someNumbers.scanLeft(0)(_-_-)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

Result

0	-10	-30	-60	-100	-150	-210
---	-----	-----	-----	------	------	------

Scan, ScanLeft, ScanRight

apply that function in rather complicated ways

```
val someNumbers = List(10,20,30,40,50,60)
```



```
someNumbers.scanRight(0)(_ - _)
```

```
List[Int] = List(-30, 40, -20, 50, -10, 60, 0)
```



```
someNumbers.scanLeft(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

```
someNumbers.scan(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

Scan

```
someNumbers.scan(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

scan makes no guarantees about direction - it is merely coincidence that it matches scanLeft

The output depends on the implementation in the traversable (here List)

Scan, ScanLeft, ScanRight

apply that function in rather complicated ways

```
val someNumbers = List(10,20,30,40,50,60)
```



```
someNumbers.scanRight(0)(_ - _)
```

```
List[Int] = List(-30, 40, -20, 50, -10, 60, 0)
```



```
someNumbers.scanLeft(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```



```
someNumbers.scan(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

Example 44

Fold, FoldLeft, FoldRight

Fold, FoldLeft, FoldRight

are very closely related to

. scan , . scanLeft , . scanRight

Fold, FoldLeft, FoldRight

will return the “last”
individual result from

. scan, . scanLeft, . scanRight

ScanRight, FoldRight

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers.scanRight(0)(_ - _)
```

```
List[Int] = List(-30, 40, -20, 50, -10, 60, 0)
```

```
someNumbers.foldRight(0)(_ - _)
```

```
Int = -30
```

ScanRight, FoldRight

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers.scanRight(0)(_ - _)
```

```
List[Int] = List(-30, 40, -20, 50, -10, 60, 0)
```

```
someNumbers.foldRight(0)(_ - _)
```

```
Int = -30
```

ScanLeft,FoldRight

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers.scanLeft(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

```
someNumbers.foldLeft(0)(_ - _)
```

```
List[Int] = -210
```

ScanLeft,FoldRight

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers.scanLeft(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

```
someNumbers.foldLeft(0)(_ - _)
```

```
List[Int] = -210
```

Scan,Fold

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers.scan(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

```
someNumbers.fold(0)(_ - _)
```

```
List[Int] = -210
```

Scan,Fold

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers. scan(0)(_ - _)
```

```
List[Int] = List(0, -10, -30, -60, -100, -150, -210)
```

```
someNumbers. fold(0)(_ - _)
```

```
List[Int] = -210
```

Fold, FoldLeft, FoldRight

will return the “last”
individual result from

. scan, . scanLeft, . scanRight

Example 45

**Reduce, ReduceLeft,
ReduceRight**

Reduce, ReduceLeft, ReduceRight

are conceptually
similar to fold,...

Reduce does not take an
initial value, however

Reduce does not take an initial value

The first 2 values of the list are used in
the first call to the reduction function

subsequent calls to the reduction
function happen as in fold

Reduce does not take an initial value

The first 2 values of the list are used in
the first call to the reduction function

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers. reduceRight(_ - _)
```

```
someNumbers. reduceLeft(_ - _)
```

```
someNumbers. reduce(_ - _)
```

Reduce does not take an initial value

The first 2 values of the list are used in
the first call to the reduction function

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers.reduceRight(_ - _)
```

Int = -30

```
someNumbers.reduceLeft(_ - _)
```

Int = -190

```
someNumbers.reduce(_ - _)
```

Int = -190

ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers.scanRight(0)(_ - _)
```

```
List[Int] = List(-30, 40, -20, 50, -10, 60, 0)
```

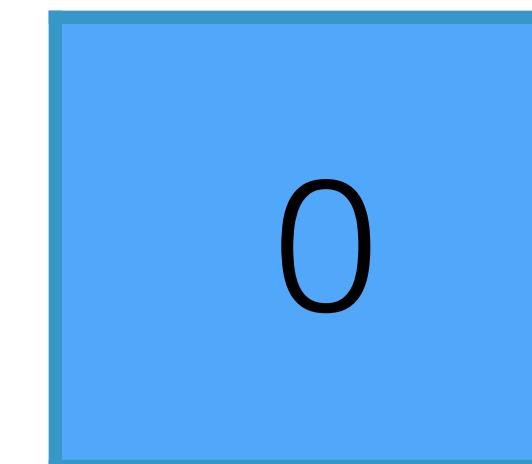
ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

```
someNumbers.scanRight(0)(_ - _)
```



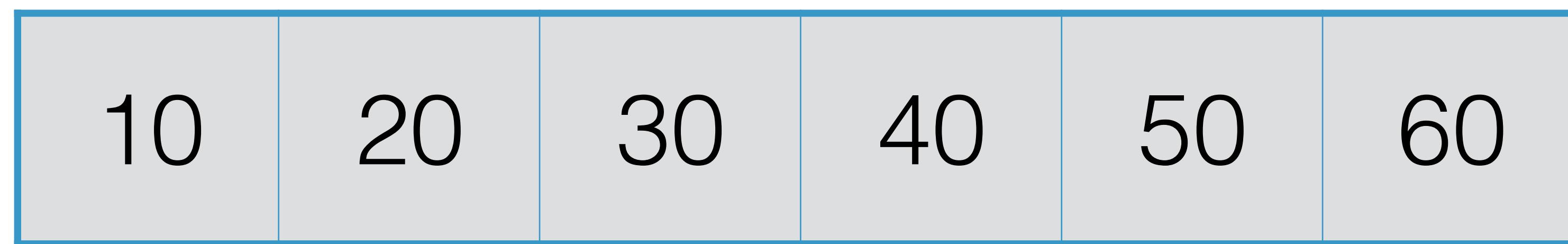
Initial Value

Function (a - b)

ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



Initial Value

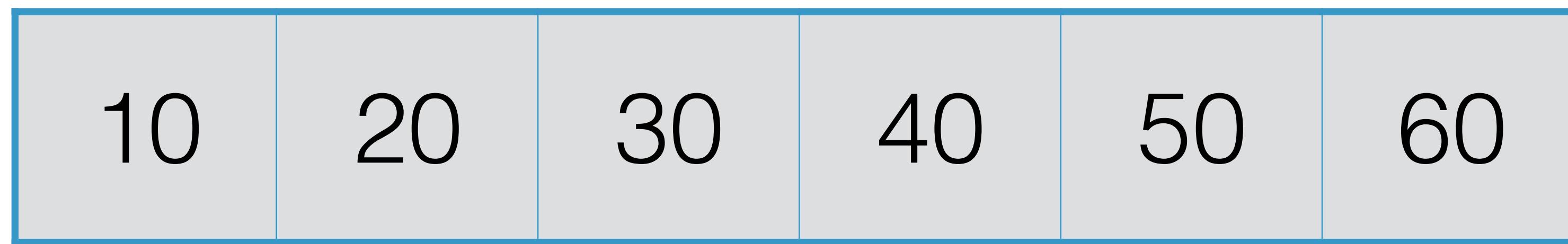
Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



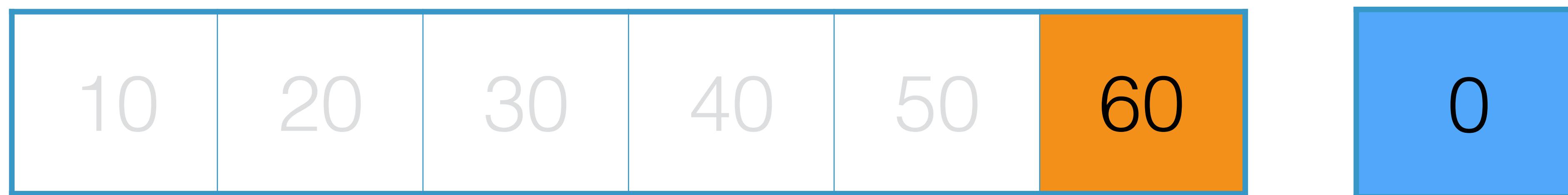
Result



ScanRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List



$$60 - 0 = 60$$

Result



reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers.reduceRight(_ - _)
```

```
Int = -30
```

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

```
someNumbers.scanRight(0)(_ - _)
```

— — — —

Function (a - b)

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

Result

--	--	--	--	--	--

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

Result

--	--	--	--	--	--

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

$$50 - 60 = -10$$

Result

--	--	--	--	--	--

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

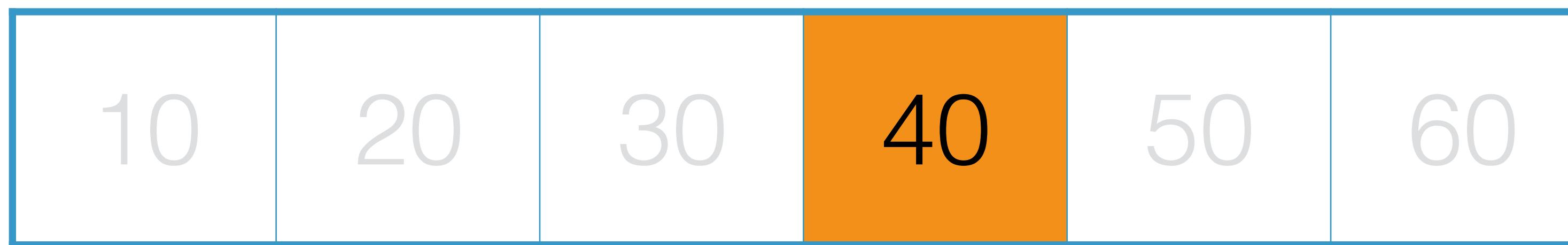
Result

					-10
--	--	--	--	--	-----

reduceRight

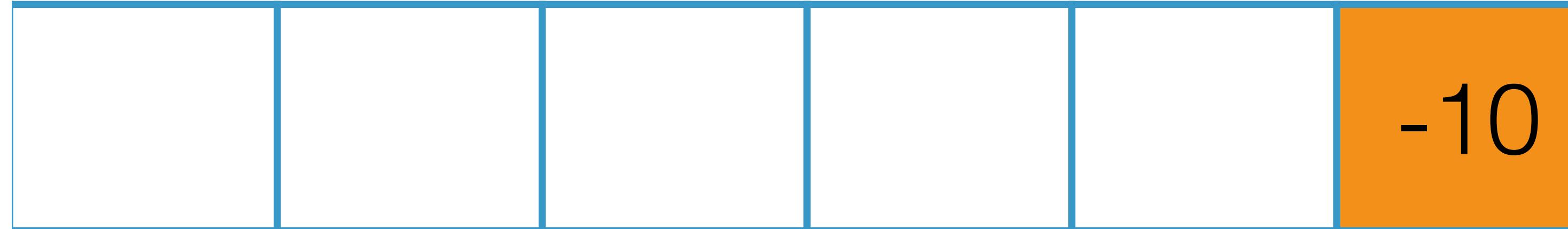
```
val someNumbers = List(10,20,30,40,50,60)
```

List



$$40 - (-10) = 50$$

Result



reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

Result

				50	-10
--	--	--	--	----	-----

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

$$30 - 50 = -20$$

Result

				50	-10
--	--	--	--	----	-----

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

Result

			-20	50	-10
--	--	--	-----	----	-----

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

$$20 - (-20) = 40$$

Result

			-20	50	-10
--	--	--	-----	----	-----

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

Result

		40	-20	50	-10
--	--	----	-----	----	-----

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

$$10 - 40 = -30$$

Result

		40	-20	50	-10
--	--	----	-----	----	-----

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

Result

	-30	40	-20	50	-10
--	-----	----	-----	----	-----

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

Result

	-30	40	-20	50	-10
--	-----	----	-----	----	-----

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

List

10	20	30	40	50	60
----	----	----	----	----	----

Result

-30

reduceRight

```
val someNumbers = List(10,20,30,40,50,60)
```

```
someNumbers.reduceRight(_ - _)
```



Int = -30

Result

-30

Example 46

Other, Simpler Reduce Operations

Other, Simpler Reduce Operations

reduce, fold and scan are
higher order methods of collections

The functions passed into these
are called Reduction Functions

Reduction Functions

because the function “reduces” the collection
to a single value, or to a smaller set of values

reduce, fold and scan are
the powerful and complicated

Reduction Functions

reduce, fold and scan are the powerful
and complicated

but there are also some very useful

Other, Simpler Reduce Operations

Say we have 3 lists

A list of weekdays

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
```

A list of weekend days

```
val weekEnds = List("Sat", "Sun")
```

A list of all days

```
val allDays = weekDays ++ weekEnds
```

```
List[String] = List(Mon, Tue, Wed, Thu, Fri, Sat, Sun)
```

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
val weekEnds = List("Sat", "Sun")
val allDays = weekDays ++ weekEnds
```

allDays endsWith weekEnds
Boolean = true

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
val weekEnds = List("Sat", "Sun")
val allDays = weekDays ++ weekEnds
```

allDays startsWith weekDays
Boolean = true

```
val weekDays = List("Mon", "Tue", "Wed", "Thu", "Fri")
val weekEnds = List("Sat", "Sun")
val allDays = weekDays ++ weekEnds
```

allDays **forall** (_ != "monday")
Boolean = true

Example 47

Sets and Maps

Sets and Maps

Lists are just 1 type of Scala Collection (albeit the most important)

Sets and Maps, like lists, are immutable, 'core' Scala collections
(analogous to Java)

Maps

Create a map from
key-value pairs

check for presence
of a given key

convert from a
list

look up the value
for a given key

apply a higher order
function

convert to a
list

Create a map from key-value pairs

```
val stateCodes = Map(  
    "California" -> "CA",  
    "New York" -> "NY",  
    ("Vermont", "VT")  
)
```

A map is a collection of key-value pairs, which are merely tuples!

Create a map from key-value pairs

```
val stateCodes = Map(  
    "California" -> "CA",  
    "New York" -> "NY",  
    ("Vermont", "VT")  
)
```

Method#1 for specifying a key-value pair

Create a map from key-value pairs

```
val stateCodes = Map(  
    "California" -> "CA",  
    "New York" -> "NY",  
    ("Vermont", "VT")  
)
```

Method#2 for specifying a key-value pair

Maps

Create a map from
key-value pairs

check for presence
of a given key

convert from a
list

look up the value
for a given key

apply a higher order
function

convert to a
list

look up the value
for a given key

```
scala> stateCodes("Vermont")
res22: String = VT
```

map(key) will return the value
corresponding to a given key

look up the value
for a given key

```
scala> stateCodes("Georgia")
java.util.NoSuchElementException: key not found: Georgia
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  at scala.collection.AbstractMap.default(Map.scala:59)
  at scala.collection.MapLike$class.apply(MapLike.scala:141)
  at scala.collection.AbstractMap.apply(Map.scala:59)
... 32 elided
```

A non-existent key will yield an
error though

look up the value
for a given key

```
scala> stateCodes("Georgia")
java.util.NoSuchElementException: key not found: Georgia
        at scala.collection.MapLike$class.default(MapLike.scala:228)
        at scala.collection.AbstractMap.default(Map.scala:59)
        at scala.collection.MapLike$class.apply(MapLike.scala:141)
        at scala.collection.AbstractMap.apply(Map.scala:59)
... 32 elided
```

A non-existent key will yield an
error though

Maps

Create a map from
key-value pairs

check for presence
of a given key

convert from a
list

look up the value
for a given key

apply a higher order
function

convert to a
list

check for presence of a given key

```
scala> stateCodes.contains("California")  
res24: Boolean = true
```

```
scala> stateCodes.contains("Georgia")  
res25: Boolean = false
```

contains(key) will return true or
false

check for presence of a given key

```
scala> stateCodes.contains("California")
res24: Boolean = true
```

```
scala> stateCodes.contains("Georgia")
res25: Boolean = false
```

contains(key) will return true or
false

check for presence of a given key

```
scala> stateCodes.contains("California")
res24: Boolean = true
```

```
scala> stateCodes.contains("Georgia")
res25: Boolean = false
```

contains(key) will return true or
false

check for presence of a given key

```
scala> stateCodes.contains("California")
res24: Boolean = true
```

```
scala> stateCodes.contains("Georgia")
res25: Boolean = false
```

contains(key) will return true or
false

Maps

Create a map from
key-value pairs

check for presence
of a given key

convert from a
list

look up the value
for a given key

apply a higher order
function

convert to a
list

Apply a higher order function

**foreach, map, reduce will all work with
Maps, just as they will with Lists**

**but the function you specify must operate
on key-value pairs, i.e. 2-element tuples**

Apply a higher order function

the function you specify must operate
on key-value pairs, i.e. 2-element tuples

```
scala> stateCodes.foreach((p: (String, String)) => println(p._1 + "=" + p._2))
California=CA
New York=NY
Vermont=VT
```

Apply a higher order function

the function you specify must operate
on key-value pairs, i.e. 2-element tuples

```
stateCodes.foreach((p: (String, String))  
=> println(p._1 + "=" + p._2))
```

2-element tuples

```
stateCodes.foreach((p: (String, String))  
=> println(p._1 + " = " + p._2))
```

the function must operate on
key-value pairs, i.e. 2-element tuples

```
stateCodes.foreach((p: (String, String))  
=> println(p._1 + "=" + p._2))
```

the function must operate on
key-value pairs, i.e. 2-element tuples

```
stateCodes.foreach((p: (String, String))  
=> println(p._1 + " = " + p._2))
```

the function must operate on
key-value pairs, i.e. 2-element tuples

```
stateCodes.foreach((p: (String, String))  
=> println(p._1 + "=" + p._2))
```

Apply a higher order function

**foreach, map, reduce will all work with
Maps, just as they will with Lists**

**but the function you specify must operate
on key-value pairs, i.e. 2-element tuples**

Maps

Create a map from
key-value pairs

check for presence
of a given key

convert from a
list

look up the value
for a given key

apply a higher order
function

convert to a
list

Convert from a list

Say we have a list of keys

```
val states = List("California", "New  
York", "Vermont")
```

And another list of values

```
val codes = List("CA", "NY", "VT")
```

Create a map using zip and toMap

```
stateCodes2 = (states zip codes).toMap
```

```
stateCodes2 = (states zip codes).toMap
```

Zip will create a list of 2-tuples
from 2 lists

```
List[(String, String)] = List((California,CA),  
(New York,NY), (Vermont,VT))
```

```
stateCodes2 = (states zip codes).toMap
```

**toMap will create Map from that
list of 2-tuples**

```
res29: scala.collection.immutable.Map[String, String] = Map(California -> CA, New York -> NY, Vermont -> VT)
```

Convert from a list

```
val states = List("California", "New York", "Vermont")
```

```
val codes = List("CA", "NY", "VT")
```

Create a map using zip and toMap

```
stateCodes2 = (states zip codes).toMap
```

You can check that its identical to our original map

```
scala> stateCodes2 == stateCodes
res27: Boolean = true
```

Maps

Create a map from
key-value pairs

check for presence
of a given key

convert from a
list

look up the value
for a given key

apply a higher order
function

convert to a
list

Convert to a list

To get a list of keys from a map:

```
val states = stateCodes.keySet.toList
```

And to get a list of values from a map:

```
val codes = stateCodes.values.toList
```

Convert to a list

To get a list of keys from a map:

```
val states = stateCodes.keySet.toList
```

And to get a list of values from a map:

```
val codes = stateCodes.values.toList
```

Convert to a list

To get a list of keys from a map:

```
val states = stateCodes.keySet.toList
```

And to get a list of values from a map:

```
val codes = stateCodes.values.toList
```

Maps

Create a map from
key-value pairs

check for presence
of a given key

convert from a
list

look up the value
for a given key

apply a higher order
function

convert to a
list

Sets and Maps

Lists are just 1 type of Scala Collection (albeit the most important)

Sets and Maps, like lists, are immutable, 'core' Scala collections
(analogous to Java)

Sets

Sets are really similar to lists, except that they
are unordered, and have a uniqueness constraint

So we won't spend a whole lot of
time on them :-)

Example 48

Mutable Collections, and Arrays

Mutable Collections, and Arrays

Lists, Sets and Maps are all immutable collections

There are corresponding mutable collections..

..and then there are Arrays

Lists, Sets and Maps are all immutable collections

They can't be grown/shrunk, neither can their elements be modified

There are corresponding mutable collections..

Rarely used, these can be grown/shrunk, and the elements can be modified too

..and then there are Arrays

these are fixed length, but elements can be modified

Lists, Sets and Maps are all immutable collections
They can't be grown/shrunk, neither can their elements be modified

Lists, Sets and Maps are all immutable collections

They can't be grown/shrunk, neither can their elements be modified

There are corresponding mutable collections..

Rarely used, these can be grown/shrunk, and the elements can be modified too

..and then there are Arrays

these are fixed length, but elements can be modified

There are corresponding mutable collections..

Rarely used, these can be grown/shrunk, and
the elements can be modified too

Lists, Sets and Maps are all immutable collections

They can't be grown/shrunk, neither can their elements be modified

There are corresponding mutable collections..

Rarely used, these can be grown/shrunk, and the elements can be modified too

..and then there are Arrays

these are fixed length, but elements can be modified

..and then there are Arrays
these are fixed length, but elements
can be modified

Mutable Collections

Creating mutable equivalents of lists, maps, sets

Converting mutable collections to immutable ones

Converting immutable collections to mutable ones

Creating lists, maps, sets

```
val someNumbers = List(10,20,30,40,50,60)
val stateCodes = Map("California" -> "CA", ("Vermont", "VT"))
val stateSet = Set("California", "Vermont")
```

Creating lists, maps, sets

```
val someNumbers = collection.immutable.List(10,20,30,40,50,60)
val stateCodes = collection.immutable.Map("California" -> "CA",
  ("Vermont","VT"))
val stateSet = collection.immutable.Set("California","Vermont")
```

Creating lists, maps, sets

```
val someNumbers = collection.immutable.List(10,20,30,40,50,60)  
val stateCodes = collection.immutable.Map("California" -> "CA",  
("Vermont","VT"))  
val stateSet = collection.immutable.Set("California","Vermont")
```

Creating mutable equivalents of lists, maps, sets

```
val someNumbers = collection.mutable.Buf.list(100, 200, 300, 400, 500, 600)
val stateCodes = collection.mutable.HashMap(("California", "CA", "CA"),
("Vermont", "VT", "VT")))
val statesSet = collection.mutable.Set("California", "Vermont")
```

Creating mutable equivalents of lists, maps, sets

```
val someNumbers = collection.mutable.Buffer(10,20,30,40,50)
val stateCodes = collection.mutable.Map("California" -> "CA",
("Vermont","VT"))
val stateSet = collection.mutable.Set("California","Vermont")
```

Mutable Collections

 Creating mutable equivalents of lists, maps, sets

Converting mutable collections to immutable ones

Converting immutable collections to mutable ones

Converting mutable collections to immutable ones

Simply use the `.toList`, `.toMap`
and `.toSet` methods!

Mutable Collections

 Creating mutable equivalents of lists, maps, sets

 Converting mutable collections to immutable ones

Converting immutable collections to mutable ones

Converting immutable collections from mutable ones

Step 1: Use the `<type>.toBuilder` method to get a builder object, basically a mutable collection

Step 2: Add all the values you want using `foreach` on the original immutable object

Step 3: Use the `.result` on the builder object to go back to the immutable

Step 1: Use the <type>.toBuilder method to get a builder object, basically a mutable collection

```
scala> val listBuilder = List.newBuilder[Int]
listBuilder: scala.collection.mutable.Builder[Int, List[Int]] = ListBuffer()
```

Step 1: Use the <type>.toBuilder method to get a builder object, basically a mutable collection

```
scala> val listBuilder = List.newBuilder[Int]
listBuilder: scala.collection.mutable.Builder[Int, List[Int]] = ListBuffer()
```

Converting immutable collections from mutable ones

Step 1: Use the `<type>.toBuilder` method to get a builder object, basically a mutable collection

Step 2: Add all the values you want using `foreach` on the original immutable object

Step 3: Use the `.result` on the builder object to go back to the immutable

**Step 2: Add all the values you want using
foreach on the original immutable object**

```
someNumbers.foreach(listBuilder+=_)
```

**Step 2: Add all the values you want using
foreach on the original immutable object**

someNumbers.foreach(**listBuilder+=_**)

Converting immutable collections from mutable ones

Step 1: Use the `<type>.toBuilder` method to get a builder object, basically a mutable collection

Step 2: Add all the values you want using `foreach` on the original immutable object

Step 3: Use the `.result` on the builder object to go back to the immutable

Step 3: Use the `.result` on the builder object to go back to the immutable

```
scala> listBuilder.result  
res45: List[Int] = List(10, 20, 30, 40, 50, 60)
```

Step 3: Use the `.result` on the builder object to go back to the immutable

```
scala> listBuilder.result  
res45: List[Int] = List(10, 20, 30, 40, 50, 60)
```

Step 3: Use the `.result` on the builder object to go back to the immutable

```
scala> listBuilder.result  
res45: List[Int] = List(10, 20, 30, 40, 50, 60)
```

Converting immutable collections from mutable ones

Step 1: Use the `<type>.toBuilder` method to get a builder object, basically a mutable collection

Step 2: Add all the values you want using `foreach` on the original immutable object

Step 3: Use the `.result` on the builder object to go back to the immutable

Mutable Collections

- ✓ Creating mutable equivalents of lists, maps, sets
- ✓ Converting mutable collections to immutable ones
- ✓ Converting immutable collections to mutable ones

Lists, Sets and Maps are all immutable collections

They can't be grown/shrunk, neither can their elements be modified

There are corresponding mutable collections..

Rarely used, these can be grown/shrunk, and the elements can be modified too

..and then there are Arrays

these are fixed length, but elements can be modified

Arrays

these are fixed length, but elements
can be modified

Arrays are technically not collections,
but practically are just like Lists

Array[String],
Array[Int] etc

Collections

Array

Arrays are normally used for collections whose size is known

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)
```

Array[String],
Array[Int] etc

```
scala> val fiveToOne = Array(5, 4, 3, 2, 1)
fiveToOne: Array[Int] = Array(5, 4, 3, 2, 1)
```

Collections

Array

Arrays have a zero based index and can be indexed using the () operator

Array[String],
Array[Int] etc

fiveToOne(2)

```
scala> val fiveToOne = Array(5, 4, 3, 2, 1)
fiveToOne: Array[Int] = Array(5, 4, 3, 2, 1)
```

Array

Collections

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)
```

```
scala> fiveInts(0) = fiveToOne(2)
```

```
scala> fiveInts
res16: Array[Int] = Array(3, 0, 0, 0, 0)
```

Array[String],
Array[Int] etc

Elements of an array
can be updated

Array

Collections

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)
```

Array[String],
Array[Int] etc

While this variable is immutable ie. it
cannot be reassigned to a new Array

It's contents are mutable

Collections

Array

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)
```

Array[String],
Array[Int] etc

Lists are very similar to Arrays

except that their contents are immutable

Example 49

Option Collections

Emptiness in Scala

None

None is a special value
associated with an Option

Option is a collection used to
capture presence or absence of a value

None is a special value associated with an Option

```
scala> def fraction(numer:Double, denom:Double): Option[Double]
| = {
|   if (denom == 0) None
|   else Option(numer/denom)
| }
```

fraction: (numer: Double, denom: Double)Option[Double]

Option is a collection used to capture presence or absence of a value

None is a special value associated with an Option

```
scala> def fraction(numer:Double, denom:Double): Option[Double]
|   = {
|     if (denom == 0) None
|     else Option(numer/denom)
|   }
fraction: (numer: Double, denom: Double)Option[Double]
```

Option is a collection used to capture presence or absence of a value

Recap

None is a special value associated with an Option

```
scala> def fraction(numer:Double, denom:Double): Option[Double]
| = {
|   if (denom == 0) None
|   else Option(numer/denom)
| }
fraction: (numer: Double, denom: Double)Option[Double]
```

Option is a collection used to capture presence or absence of a value

None is a special value associated with an Option

```
scala> def fraction(numer:Double, denom:Double): Option[Double]
| = {
|   if (denom == 0) None
|   else Option(numer/denom)
| }
fraction: (numer: Double, denom: Double)Option[Double]
```

```
scala> fraction(100,0)
res5: Option[Double] = None
```

Option is a collection used to capture presence or absence of a value

None is a special value associated with an Option

```
scala> def fraction(numer:Double, denom:Double): Option[Double]
| = {
|   if (denom == 0) None
|   else Option(numer/denom)
| }
```

fraction: (numer: Double, denom: Double)Option[Double]

Option is a collection used to capture presence or absence of a value

None is a special value associated with an Option

```
scala> def fraction(numer:Double, denom:Double): Option[Double]
|   = {
|     if (denom == 0) None
|     else Option(numer/denom)
|   }
```

fraction: (numer: Double, denom: Double)Option[Double]

```
[scala> fraction(22,7)
res4: Option[Double] = Some(3.142857142857143)
```

Option is a collection used to capture presence or absence of a value

None is a special value
associated with an Option

Some is the opposite of
None

```
[scala] > fraction(22,7)
res4: Option[Double] = Some(3.142857142857143)
```

Option is a collection used to
capture presence or absence of a value

Option is a collection used to capture presence or absence of a value

Using Option is a type-safe alternative to using null in Java

Option forces the programmer to explicitly deal with the possibility that nothing was returned

Option forces the programmer to explicitly deal
with the possibility that nothing was returned

Err...how?

In order to extract a value from the result of an
Option, the programmer must specify a plan B

In order to extract a value from the result of an Option, the programmer must specify a plan B

```
scala> val piOption = fraction(22,7)
piOption: Option[Double] = Some(3.142857142857143)
```

```
scala> val pi = piOption getOrElse "Oops, pi should be 3.14"
pi: Any = 3.142857142857143
```

call the function as usual

In order to extract a value from the result of an Option, the programmer must specify a plan B

```
scala> val piOption = fraction(22,7)
piOption: Option[Double] = Some(3.142857142857143)
```

```
scala> val pi = piOption getOrElse "Oops, pi should be 3.14"
pi: Any = 3.142857142857143
```

but to extract the value, use getOrElse

In order to extract a value from the result of an Option, the programmer must specify a plan B

```
scala> val piOption = fraction(22,7)
piOption: Option[Double] = Some(3.142857142857143)
```

```
scala> val pi = piOption getOrElse "Oops, pi should be 3.14"
pi: Any = 3.142857142857143
```

but to extract the value, use getOrElse

In order to extract a value from the result of an Option, the programmer must specify a plan B

```
scala> val piOption = fraction(22,7)
piOption: Option[Double] = Some(3.142857142857143)
```

```
scala> val pi = piOption getOrElse "Oops, pi should be 3.14"
pi: Any = 3.142857142857143
```

which forces you to specify the value returned if the Option contained None

In order to extract a value from the result of an Option, the programmer must specify a plan B

```
scala> val piOption = fraction(22,7)
piOption: Option[Double] = Some(3.142857142857143)
```

```
scala> val pi = piOption getOrElse "Oops, pi should be 3.14"
pi: Any = 3.142857142857143
```

this value was not needed in this particular instance..

In order to extract a value from the result of an Option, the programmer must specify a plan B

```
scala> val piOption = fraction(22,0)  
piOption: Option[Double] = None
```

```
scala> val pi = piOption getOrElse "Oops, pi should be 3.14"  
pi: Any = Oops, pi should be 3.14
```

but had we mistyped the denominator..

In order to extract a value from the result of an Option, the programmer must specify a plan B

```
scala> val piOption = fraction(22,0)
piOption: Option[Double] = None
```

```
scala> val pi = piOption getOrElse "Oops, pi should be 3.14"
pi: Any = Oops, pi should be 3.14
```

but had we mistyped the denominator..

In order to extract a value from the result of an Option, the programmer must specify a plan B

```
scala> val piOption = fraction(22,0)  
piOption: Option[Double] = None
```

```
scala> val pi = piOption getOrElse "Oops, pi should be 3.14"  
pi: Any = Oops, pi should be 3.14
```

our plan B would have come in handy

In order to extract a value from the result of an Option, the programmer must specify a plan B

```
scala> val piOption = fraction(22,0)
piOption: Option[Double] = None
```

```
scala> val pi = piOption getOrElse "Oops, pi should be 3.14"
pi: Any = Oops, pi should be 3.14
```

our plan B would have come in handy

In order to extract a value from the result of an Option, the programmer must specify a plan B

Another way of extracting the value (again with a plan B) is via a match

```
piOption match
{
    case Some(pi) => pi
    case None => "Something bad happened"
}
```

match our option variable

In order to extract a value from the result of an Option, the programmer must specify a plan B

Another way of extracting the value (again with a plan B) is via a match

```
piOption match
{
    case Some(pi) => pi
    case None => "Something bad happened"
}
```

match our option variable

In order to extract a value from the result of an Option, the programmer must specify a plan B

Another way of extracting the value (again with a plan B) is via a match

```
piOption match
{
    case Some(pi) => pi
    case None => "Something bad happened"
}
```

Check for Some, i.e. plan A!

In order to extract a value from the result of an Option, the programmer must specify a plan B

Another way of extracting the value (again with a plan B) is via a match

```
piOption match
{
    case Some(pi) => pi
    case None => "Something bad happened"
}
```

Check for None, i.e. plan B!

Example 50

Error handling with util.Try

Error handling with util.Try

Almost exactly analogous to the use of Options instead of null

is the use of util.Try to wrap around expressions that might throw exceptions

Error handling with util.Try searching a map for a non-existent key throws an exception

```
val stateCodes = Map("California" -> "CA", "Vermont" -> "VT")
val stateCode = stateCodes("NoSuchState")
```

```
java.util.NoSuchElementException: key not found: NoSuchState
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  at scala.collection.AbstractMap.default(Map.scala:59)
  at scala.collection.MapLike$class.apply(MapLike.scala:141)
  at scala.collection.AbstractMap.apply(Map.scala:59)
  ... 32 elided
```

Error handling with `util.Try`

wrap the lookup in a `util.Try`

```
val stateCodes = Map("California" -> "CA", "Vermont" -> "VT")
val stateCode = stateToStateCode("NoSuchState"))
```

Error handling with util.Try

wrap the lookup in a util.Try

```
val stateCodes = Map("California" -> "CA", "Vermont" -> "VT")
val stateCode = util.Try(stateCodes("NoSuchState"))
```

```
stateCode: scala.util.Try[String] = Failure(java.util.NoSuchElementException: key not found: NoSuchState)
```

failure is graceful..the return type is
Try[String]

Error handling with `util.Try`

wrap the lookup in a `util.Try`

```
scala> val stateCode = util.Try(stateCodes("California"))
stateCode: scala.util.Try[String] = Success(CA)
```

Success is pretty graceful too..

Error handling with `util.Try`

the return value can be obtained either
with `getOrElse`, or with a `match`

Of course both of these require the
programmer to specify a plan B as usual

Error handling with util.Try

```
stateCode match {  
    case util.Success(code) => code  
    case util.Failure(error) => "something terrible happened!"  
}
```

Error handling with util.Try

```
stateCode match {  
    case util.Success(code) => code  
    case util.Failure(error) => "something terrible happened!"  
}
```

Error handling with util.Try

```
val stateCodes = Map("California" -> "CA", "Vermont" -> "VT")
val stateCode = util.try(stateCodes("California"))
```

```
stateCode match {
  case util.Success(code) => code
  case util.Failure(error) => "something terrible happened!"
}
```

String = CA

Error handling with `util.Try`

```
val stateCodes = Map("California" -> "CA", "Vermont" -> "VT")  
val stateCode = util.try(stateCodes("California"))
```

```
stateCode match {  
  case util.Success(code) => code  
  case util.Failure(error) => "something terrible happened!"  
}
```

String = CA

Error handling with util.Try

```
stateCode match {  
    case util.Success(code) => code  
    case util.Failure(error) => "something terrible happened!"  
}
```

Error handling with util.Try

```
val stateCodes = Map("California" -> "CA", "Vermont" -> "VT")
val stateCode = util.try(stateCodes("NoSuchState"))
```

```
stateCode match {
  case util.Success(code) => code
  case util.Failure(error) => "something terrible happened!"
}
```

String = something terrible happened!

Error handling with util.Try

```
val stateCodes = Map("California" -> "CA", "Vermont" -> "VT")  
val stateCode = util.try(stateCodes("NoSuchState"))
```

```
stateCode match {  
  case util.Success(code) => code  
  case util.Failure(error) => "something terrible happened!"  
}
```

String = something terrible happened!

Error handling with util.Try

```
stateCode match {  
    case util.Success(code) => code  
    case util.Failure(error) => "something terrible happened!"  
}
```

Error handling with util.Try

Almost exactly analogous to the use of Options instead of null

is the use of util.Try to wrap around expressions that might throw exceptions