
FOUNDATION

Plan

- Deep dive into function features
- Functional programming patterns
- Pure function and functional subset

Val vs Def functions

Functions

Val function (Lambda)

```
val replicate: (Int, String) => String =
  (n: Int, text: String) => ...
```

```
replicate(3, "Hello ")
// res1: String = "Hello Hello Hello "
```

Def function (Method)

```
def replicate(n: Int, text: String): String =
  ...
```

```
replicate(3, "Hello ")
// res3: String = "Hello Hello Hello "
```

Val function (Lambda or anonymous function)

```
(n: Int, text: String) => List.fill(n)(text).mkString
```

Val functions are ordinary objects

```
(n: Int, text: String) => List.fill(n)(text).mkString
```

```
3

"Hello World!"

User("John Doe", 27)
```

Val functions are ordinary objects

```
val replicate = (n: Int, text: String) => List.fill(n)(text).mkString
```

```
val counter = 3

val message = "Hello World!"

val john    = User("John Doe", 27)
```

Val functions are ordinary objects

```
val replicate = (n: Int, text: String) => List.fill(n)(text).mkString
```

```
val counter = 3

val message = "Hello World!"

val john    = User("John Doe", 27)
```

```
val repeat = replicate
```

Val functions are ordinary objects

```
val numbers   = List(1,2,3)
// numbers: List[Int] = List(1, 2, 3)

val functions = List((x: Int) => x + 1, (x: Int) => x - 1, (x: Int) => x * 2)
// functions: List[Int => Int] = List(<function1>, <function1>, <function1>)
```

Val functions are ordinary objects

```
val numbers   = List(1,2,3)
// numbers: List[Int] = List(1, 2, 3)

val functions = List((x: Int) => x + 1, (x: Int) => x - 1, (x: Int) => x * 2)
// functions: List[Int => Int] = List(<function1>, <function1>, <function1>)
```

```
functions(0)(10)
// res12: Int = 11

functions(2)(10)
// res13: Int = 20
```

Val function desugared

```
val replicate: (Int, String) => String = (n: Int, text: String) => List.fill(n)(text).mkString
```

Val function desugared

```
val replicate: (Int, String) => String         = (n: Int, text: String) => List.fill(n)(text).mkString
```

```
val replicate: Function2[Int, String, String] = (n: Int, text: String) => List.fill(n)(text).mkString
```

Val function desugared

```
val replicate: (Int, String) => String        = (n: Int, text: String) => List.fill(n)(text).mkString
```

```
val replicate: Function2[Int, String, String] = new Function2[Int, String, String] {
  def apply(n: Int, text: String): String =
     List.fill(n)(text).mkString
}
```

Val function desugared

```
val replicate: (Int, String) => String        = (n: Int, text: String) => List.fill(n)(text).mkString
```

```
val replicate: Function2[Int, String, String] = new Function2[Int, String, String] {
  def apply(n: Int, text: String): String =
     List.fill(n)(text).mkString
}
```

```
replicate.apply(3, "Hello ")
// res19: String = "Hello Hello Hello "
```

Val function desugared

```
val replicate: (Int, String) => String        = (n: Int, text: String) => List.fill(n)(text).mkString
```

```
val replicate: Function2[Int, String, String] = new Function2[Int, String, String] {
  def apply(n: Int, text: String): String =
     List.fill(n)(text).mkString
}
```

```
replicate.apply(3, "Hello ")
// res19: String = "Hello Hello Hello "
```

```
replicate(3, "Hello ")
// res20: String = "Hello Hello Hello "
```

Def function (Method)

```
import java.time.LocalDate

def createDate(year: Int, month: Int, dayOfMonth: Int): LocalDate =
  ...
```

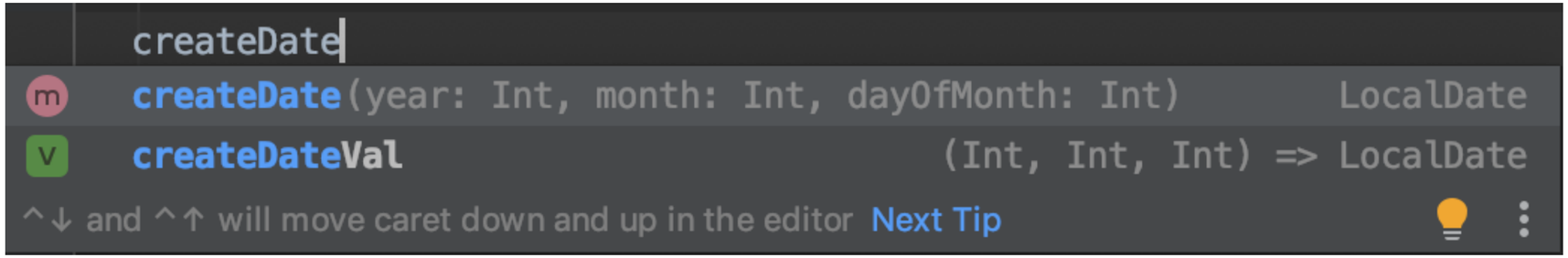
```
createDate(2020, 1, 5)
// res21: LocalDate = 2020-01-05
```

Function arguments

```
import java.time.LocalDate

def createDate(year: Int, month: Int, dayOfMonth: Int): LocalDate =
  ...
```

```
val createDateVal: (Int, Int, Int) => LocalDate =
  (year, month, dayOfMonth) => ...
```

IDE



Javadoc

```
def createDate(year: Int, month: Int, dayOfMonth: Int): LocalDate
```

```
val createDateVal: (Int, Int, Int) => LocalDate
```

Named arguments

```
import java.time.LocalDate

def createDate(year: Int, month: Int, dayOfMonth: Int): LocalDate =
  ...
```

```
createDate(2020, 1, 5)
// res23: LocalDate = 2020-01-05
```

```
createDate(dayOfMonth = 5, month = 1, year = 2020)
// res24: LocalDate = 2020-01-05
```

Def functions are not data

```
def createDate(year: Int, month: Int, dayOfMonth: Int): LocalDate =
  ...
```

```
List(createDate)
// error: missing argument list for method createDate in class App10
// Unapplied methods are only converted to functions when a function type is expected.
// You can make this conversion explicit by writing `createDate _` or `createDate(_,_,_)` instead of `createDate`.
```

Def functions are not data

```
def createDate(year: Int, month: Int, dayOfMonth: Int): LocalDate =
  ...
```

```
List(createDate _)
// res26: List[(Int, Int, Int) => LocalDate] = List(<function3>)
```

Def functions are not data

```
def createDate(year: Int, month: Int, dayOfMonth: Int): LocalDate =
  ...
```

```
List(createDate _)
// res26: List[(Int, Int, Int) => LocalDate] = List(<function3>)
```

```
val createDateVal = createDate _
// createDateVal: (Int, Int, Int) => LocalDate = <function3>
```

Def functions are not data

```
def createDate(year: Int, month: Int, dayOfMonth: Int): LocalDate =
  ...
```

```
List(createDate): List[(Int, Int, Int) => LocalDate]
```

```
val createDateVal: (Int, Int, Int) => LocalDate = createDate
```

Summary

- Val functions are an ordinary objects
- Use def functions for API
- Easy to convert def to val

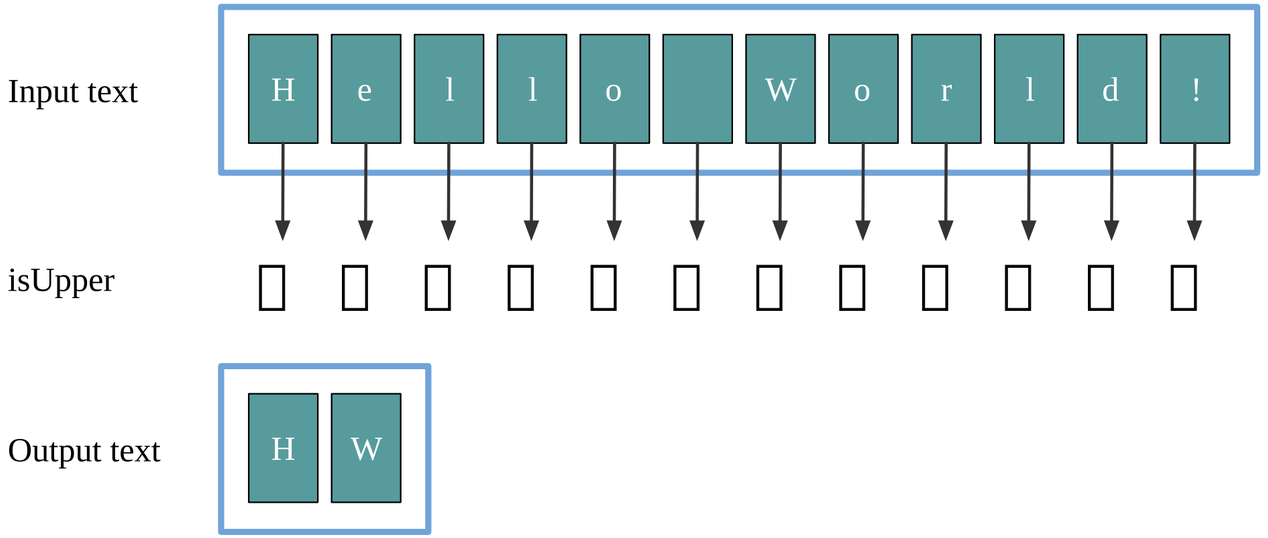
Functions as input

Functions as input

```
def filter(
  text     : String,
  predicate: Char => Boolean
): String = ...
```

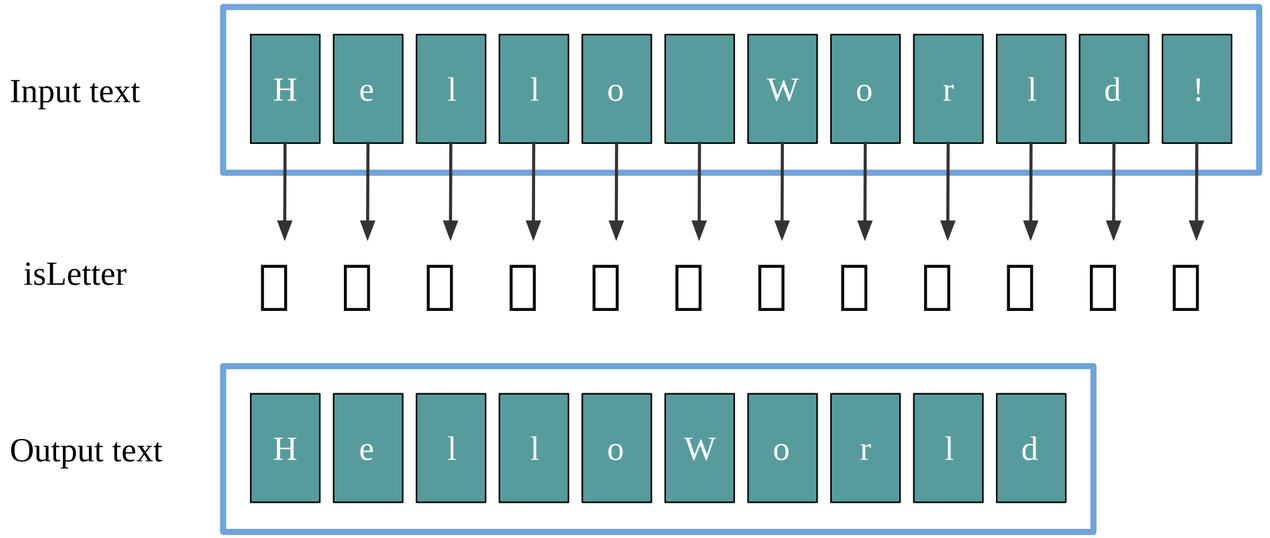
Functions as input

```
filter(
  "Hello World!",
  (c: Char) => c.isUpper
)
// res29: String = "HW"
```



Functions as input

```
filter(
  "Hello World!",
  (c: Char) => c.isLetter
)
// res30: String = "HelloWorld"
```



Reduce code duplication

```
def upperCase(text: String): String = {
  val characters = text.toArray
  for (i <- 0 until text.length) {
    characters(i) = characters(i).toUpper
  }
  new String(characters)
}
```

```
upperCase("Hello")
// res31: String = "HELLO"
```

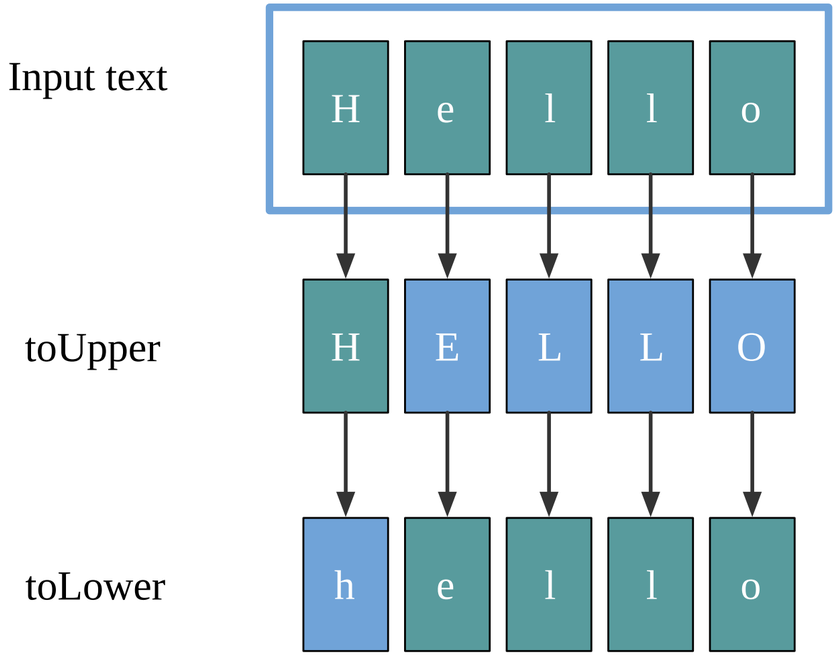
```
def lowerCase(text: String): String = {
  val characters = text.toArray
  for (i <- 0 until text.length) {
    characters(i) = characters(i).toLower
  }
  new String(characters)
}
```

```
lowerCase("Hello")
// res32: String = "hello"
```

Capture pattern

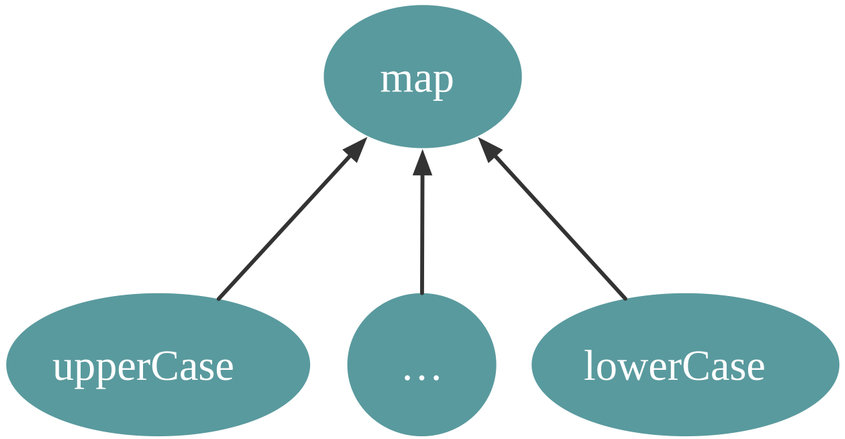
```
def map(text: String, update: Char => Char): String =
  val characters = text.toArray
  for (i <- 0 until text.length) {
    characters(i) = update(characters(i))
  }
  new String(characters)
}
```

```
def upperCase(text: String): String =
  map(text, c => c.toUpper)

def lowerCase(text: String): String =
  map(text, c => c.toLower)
```



Capture pattern

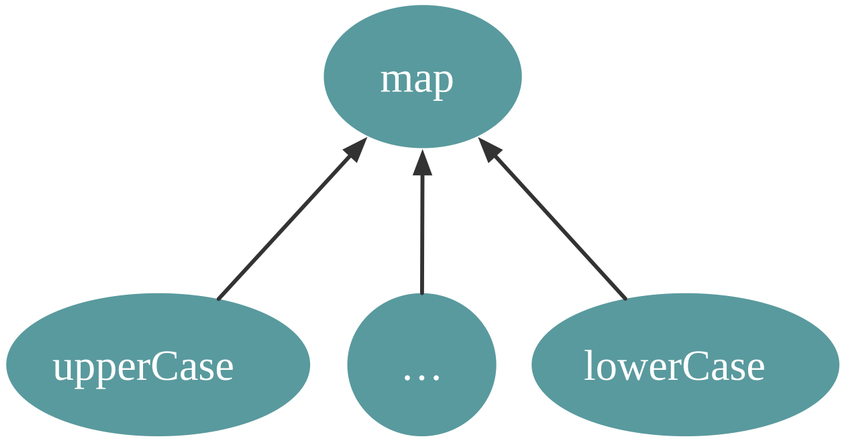
```
def map(text: String, update: Char => Char): String =
  val characters = text.toArray
  for (i <- 0 until text.length) {
    characters(i) = update(characters(i))
  }
  new String(characters)
}
```



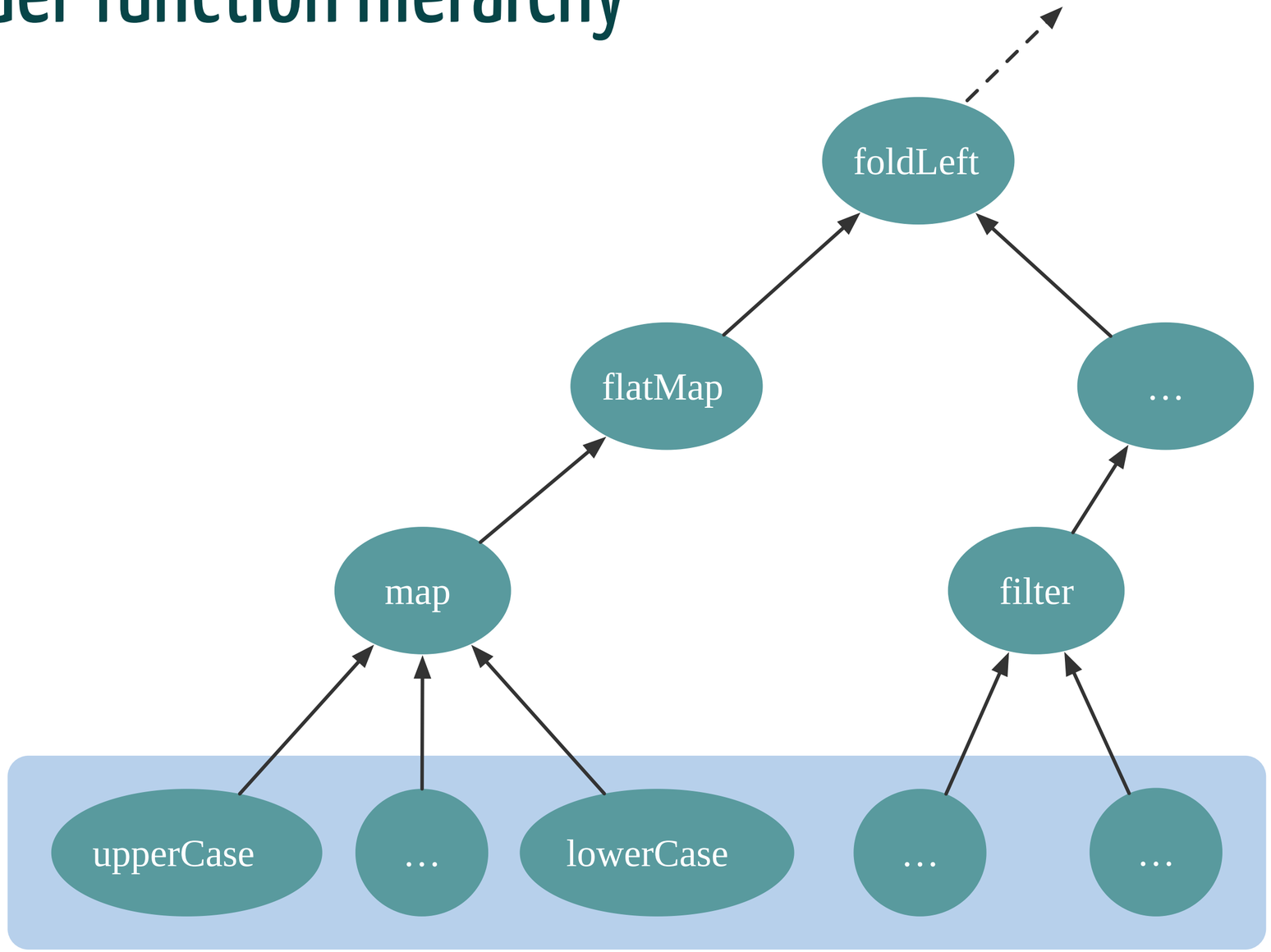

```
def upperCase(text: String): String =
  map(text, c => c.toUpper)

def lowerCase(text: String): String =
  map(text, c => c.toLower)
```

Property based testing

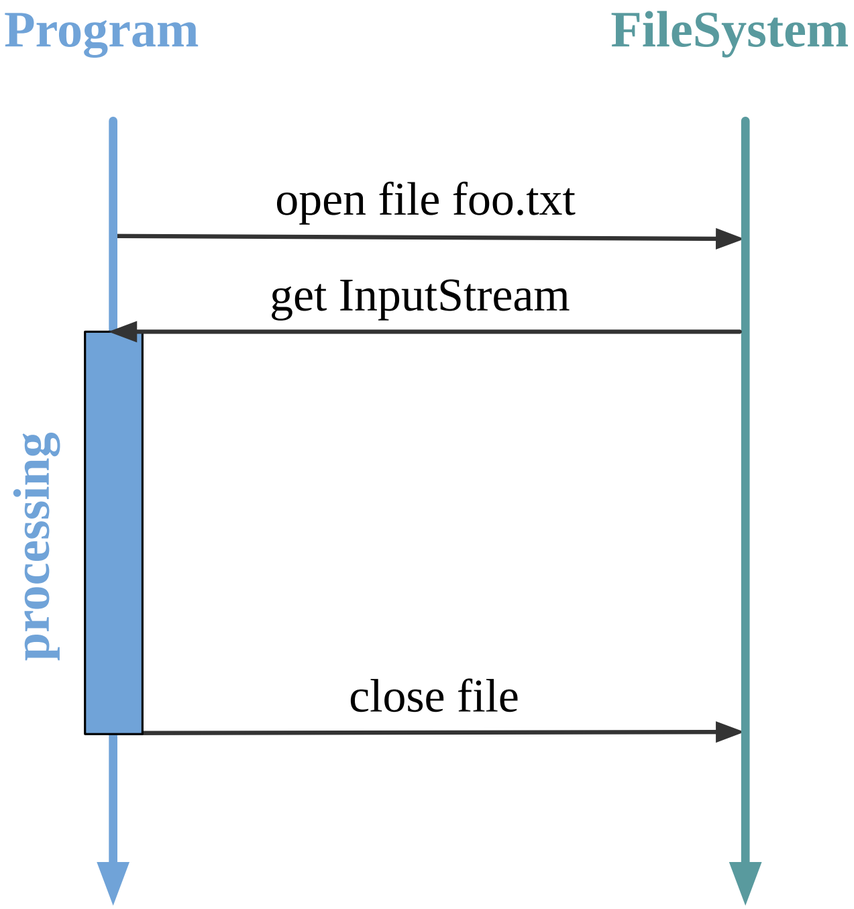
```
test("map does not modify the size of a text") {
  forAll((
    text  : String,
    update: Char => Char
  ) =>
    val outputText = map(text, update)
    outputText.length == text.length
  )
}
```



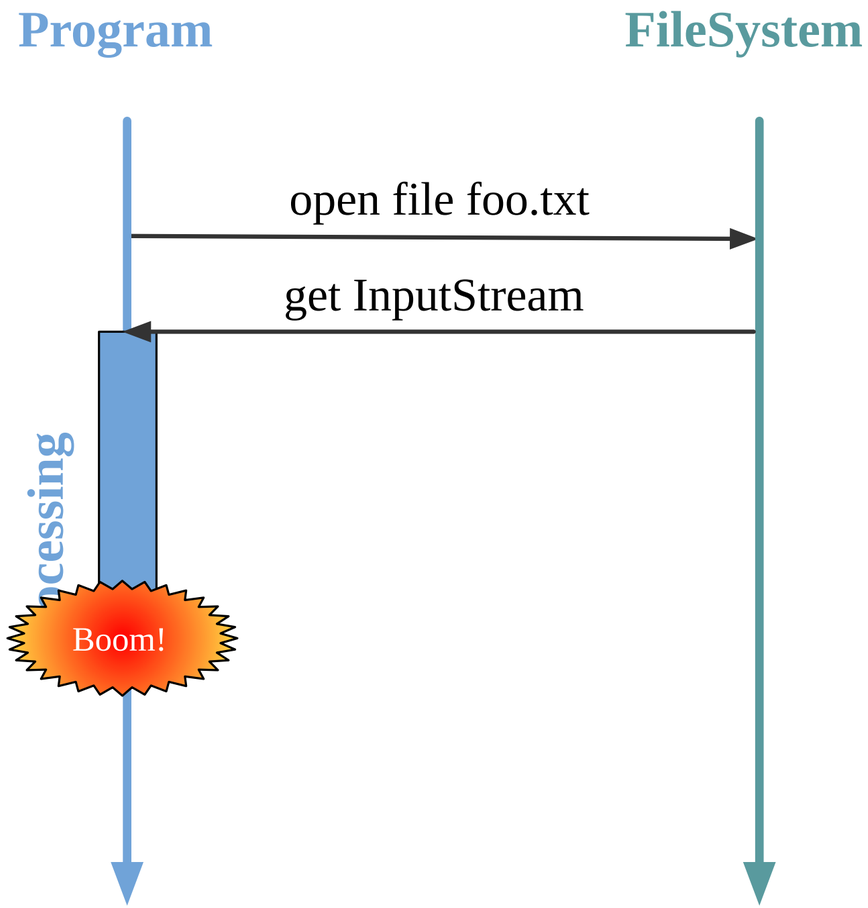
Higher order function hierarchy



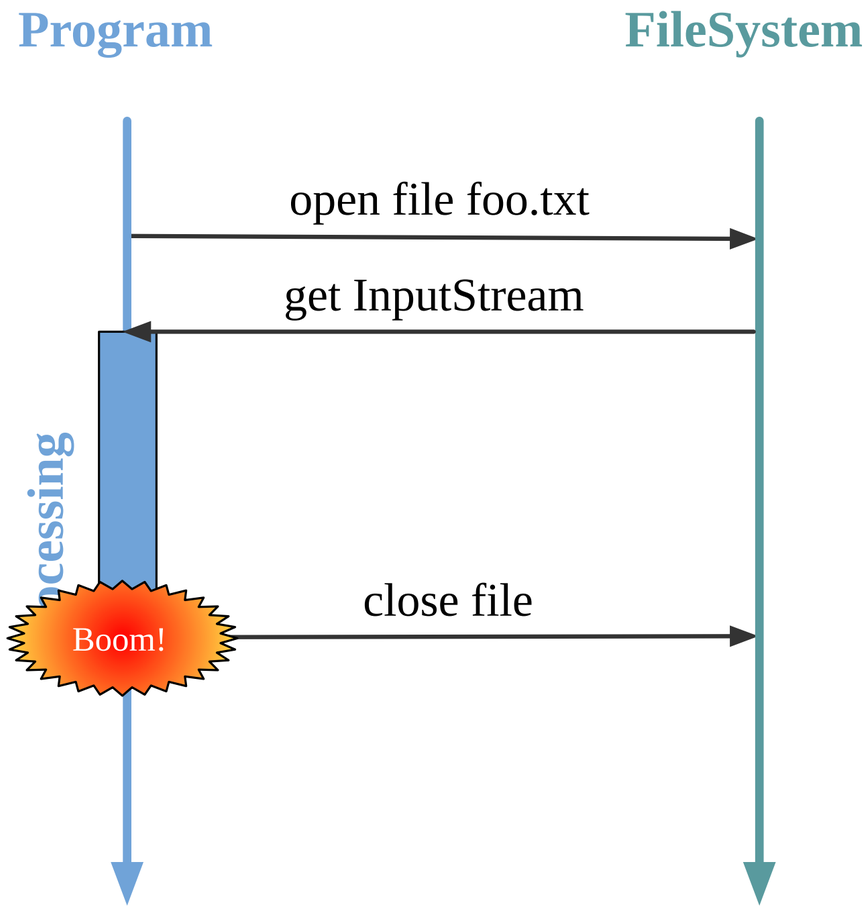
File processing



File processing



File processing



Write tricky code once

```
import scala.io.Source

def usingFile(fileName: String, processing: Iterator[String] => Int): Int = {
  val source = Source.fromResource(fileName)
  try {
    processing(source.getLines())
  } finally {
    source.close()
  }
}
```

Write tricky code once

```
import scala.io.Source

def usingFile(fileName: String, processing: Iterator[String] => Int): Int = {
  val source = Source.fromResource(fileName)
  try {
    processing(source.getLines())
  } finally {
    source.close()
  }
}
```

```
val countLines: Iterator[String] => Int =
  lines => lines.size
```

```
val countWords: Iterator[String] => Int =
  lines => ...
```

Write tricky code once

```
import scala.io.Source

def usingFile(fileName: String, processing: Iterator[String] => Int): Int = {
  val source = Source.fromResource(fileName)
  try {
    processing(source.getLines())
  } finally {
    source.close()
  }
}
```

```
usingFile("50-word-count.txt", countLines)
// res36: Int = 2
```

```
usingFile("50-word-count.txt", countWords)
// res37: Int = 50
```

Summary

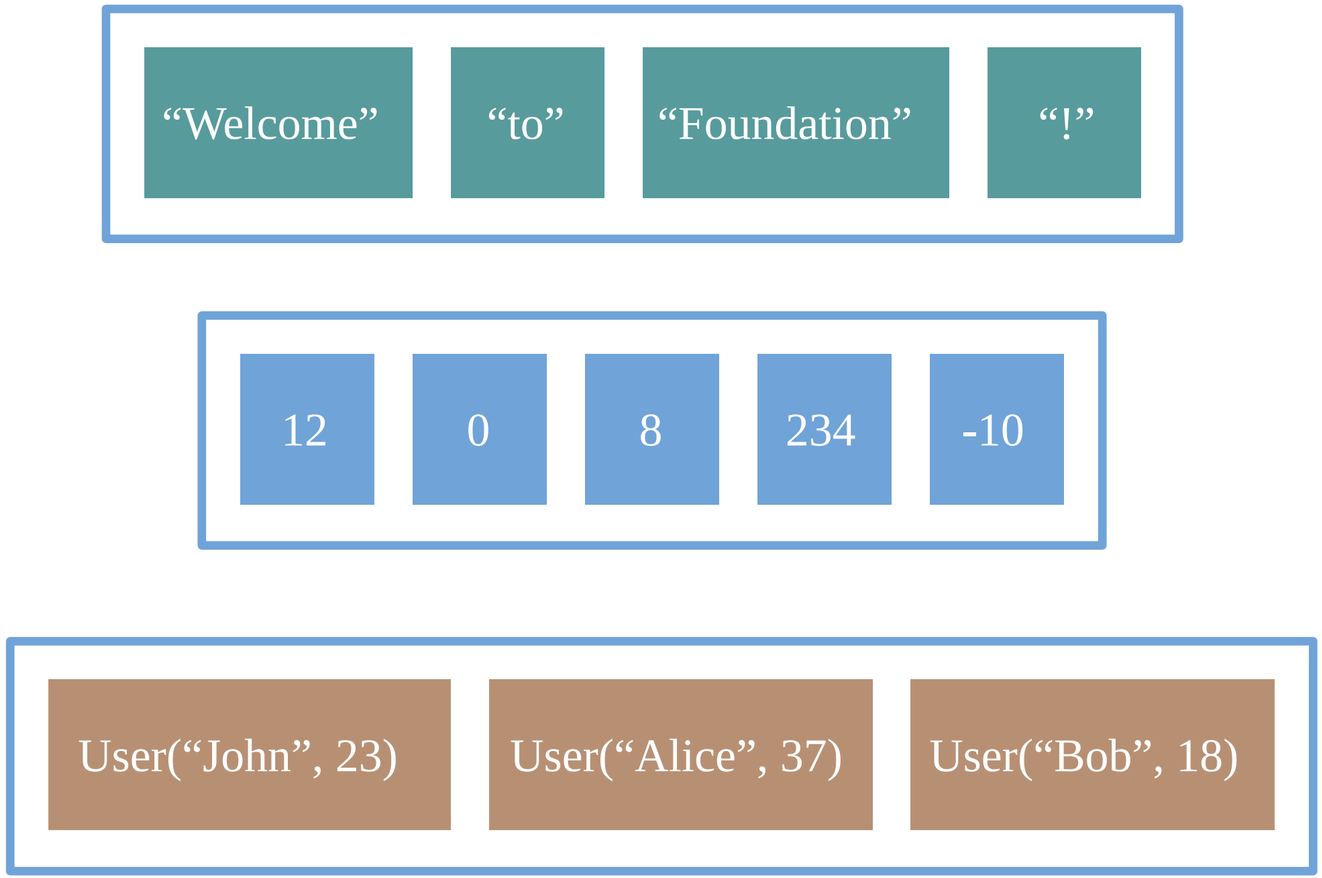
- Higher order function
- Reduce code duplication
- Improve code quality

Exercise 1: Functions as input

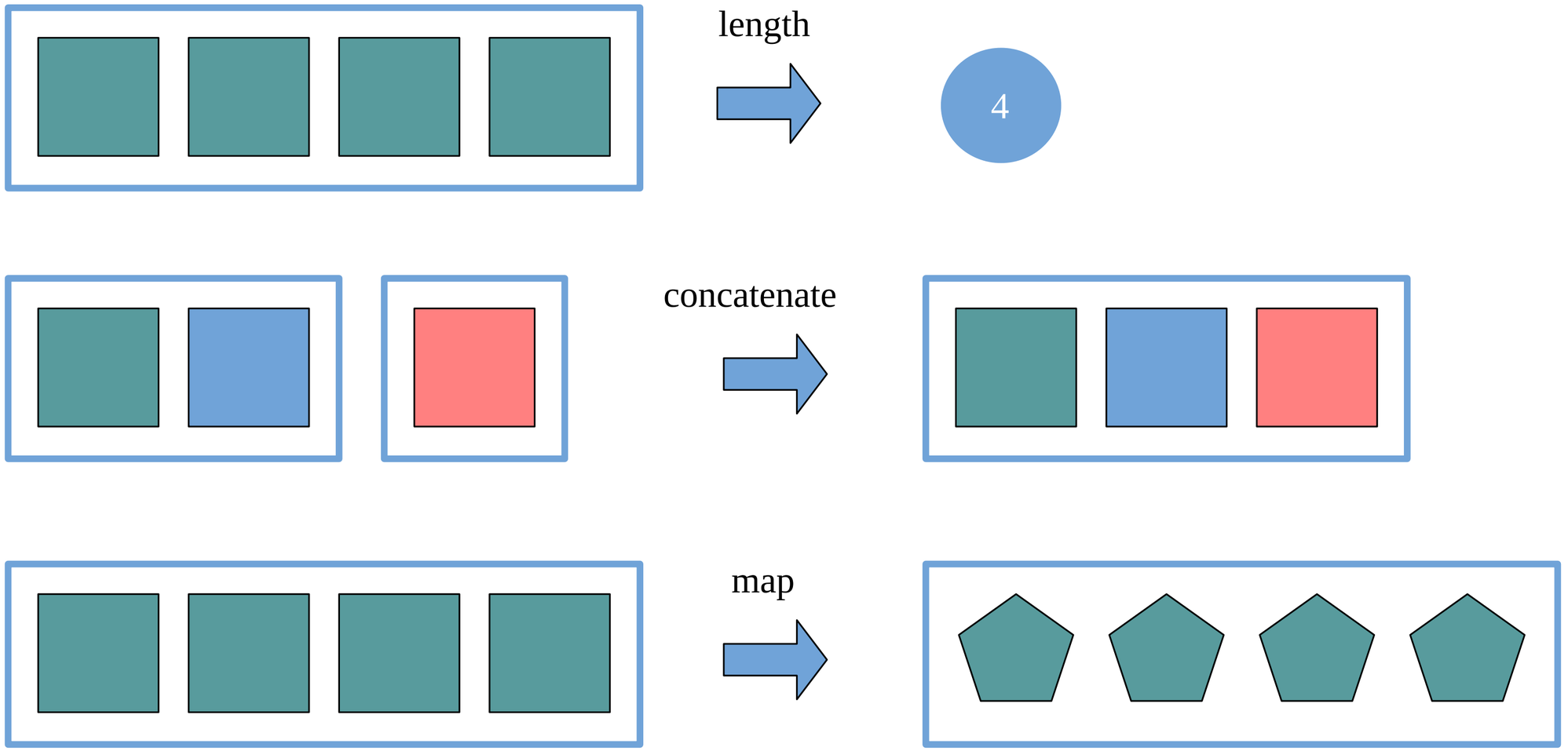
`exercises.function.FunctionExercises.scala`

Parametric functions

Sequence is a generic data structure

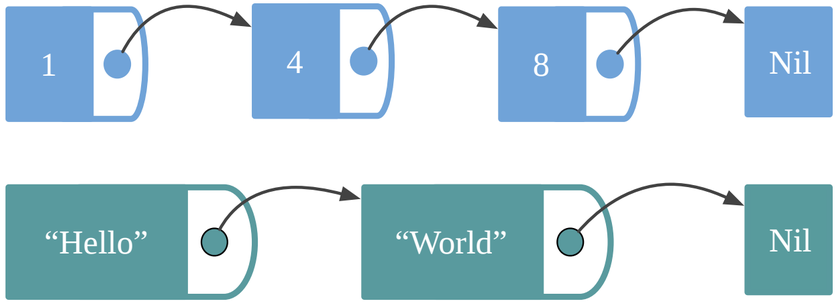


Generic operations



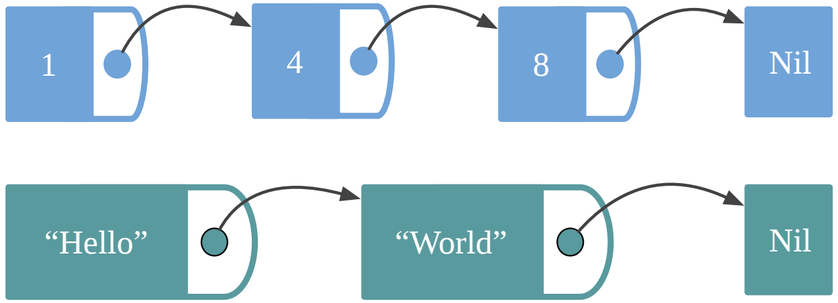
Linked List

```
val numbers: List[Int]    = List(1, 2, 3)

val words  : List[String] = List("Hello", "World")
```



Linked List

```
val numbers: List[Int]    = List(1, 2, 3)

val words  : List[String] = List("Hello", "World")
```




```
val numbers: List = List(1, 2, 3)
// error: type List takes type parameters
// val numbers: List = List(1, 2, 3)
//              ^^^^
```

How to parametrise a function?

```
def map(list: List[Int]   , update: Int    => Int   ): List[Int]    = ...
def map(list: List[String], update: String => String): List[String] = ...
```

How to parametrise a function?

```
def map(list: List[Int]   , update: Int    => Int   ): List[Int]    = ...
def map(list: List[String], update: String => String): List[String] = ...
```

```
def map[A](list: List[A], update: A => A): List[A] = ...
```

How to parametrise a function?

```
def map[A](list: List[A], update: A => A): List[A] = ...
```

```
map(List(1,2,3,4), (x: Int) => x + 1)
// res38: List[Int] = List(2, 3, 4, 5)

map(List("Hello", "World"), (x: String) => x.reverse)
// res39: List[String] = List("olleH", "dlroW")
```

How to parametrise a function?

```
def map[A](list: List[A], update: A => A): List[A] = ...
```

```
val users = List(User("John", 23), User("Alice", 37), User("Bob", 18))
```

```
map(users, (x: User) => x.age)
// error: type mismatch;
//  found   : App14.this.User => Int
//  required: Any => Any
// map(users, (x: User) => x.age)
//            ^^^^^^^^^^^^^^^^^^
```

How to parametrise a function?

```
def map[A](list: List[A], update: A => A): List[A] = ...
```

```
val users = List(User("John", 23), User("Alice", 37), User("Bob", 18))
```

```
map[User](users, (x: User) => x.age)
// error: type mismatch;
//  found   : Int
//  required: App14.this.User
// map[User](users, (x: User) => x.age)
//                               ^^^^^
```

How to parametrise a function?

```
def map[From, To](list: List[From], update: From => To): List[To] = ...
```

How to parametrise a function?

```
def map[From, To](list: List[From], update: From => To): List[To] = ...
```

```
val users = List(User("John", 23), User("Alice", 37), User("Bob", 18))
```

```
map(users, (x: User) => x.age)
// res43: List[Int] = List(23, 37, 18)

map(List(1,2,3,4), (x: Int) => x + 1)
// res44: List[Int] = List(2, 3, 4, 5)
```

How to parametrise a function?

```
def map[From, To](list: List[From], update: From => To): List[To] = ...
```

```
val users = List(User("John", 23), User("Alice", 37), User("Bob", 18))
```

```
map(users, (x: User) => x.age)
// res43: List[Int] = List(23, 37, 18)

map(List(1,2,3,4), (x: Int) => x + 1)
// res44: List[Int] = List(2, 3, 4, 5)
```

#1 Benefit: code reuse

Interpretation

```
def map[From, To](list: List[From], update: From => To): List[To] = ...
```

Interpretation

```
def map[From, To](list: List[From], update: From => To): List[To] = ...
```

The callers of `map` choose `From` and `To`

```
map[String, Int](List("Hello", "World!"), (x: String) => x.length)
// res45: List[Int] = List(5, 6)
```

How can we implement `map`?

```
def map[From, To](list: List[From], update: From => To): List[To] = ...
```

How can we implement `map`?

```
def map[From, To](list: List[From], update: From => To): List[To] = ...
```

- Always return `List.empty` (`Nil`)

How can we implement `map`?

```
def map[From, To](list: List[From], update: From => To): List[To] = ...
```

- Always return `List.empty` (`Nil`)
- Somehow call `f` on the elements of `list`

Does it compile?

```
def map[From, To](list: List[From], update: From => To): List[To] =
  List(1,2,3)
```

Does it compile?

```
def map[From, To](list: List[From], update: From => To): List[To] =
  List(1,2,3)

On line 3: error: type mismatch;
        found   : Int(1)
        required: To
```

Does it compile?

```
def map[From, To](list: List[From], update: From => To): List[To] =
  List(1,2,3)

On line 3: error: type mismatch;
        found   : Int(1)
        required: To
```

```
def map(list: List[Int], update: Int => Int): List[Int] =
  List(1,2,3)
```

Does it compile?

```
def map[From, To](list: List[From], update: From => To): List[To] =
  List(1,2,3)

On line 3: error: type mismatch;
        found   : Int(1)
        required: To
```

```
def map(list: List[Int], update: Int => Int): List[Int] =
  List(1,2,3)
```

#2 Benefit: require less tests and less documentation

Summary

- More reusable
- Caller decides which underlying type to use
- Implementation must be generic
  - more documentation
  - less tests

Exercise 2: Parametric functions

`exercises.function.FunctionExercises.scala`

Two apparently useless functions

```
def identity[A](value: A): A =
  value
```

```
identity(5)
// res47: Int = 5

identity("Hello")
// res48: String = "Hello"
```

```
def constant[A, B](value: A)(discarded: B): A =
  value
```

```
constant(5)("Hello")
// res49: Int = 5

constant("Hello")(5)
// res50: String = "Hello"
```

Two apparently useless functions

```
object Config {
  private var flag: Boolean = true

  def modifyFlag(f: Boolean => Boolean): Boolean = {
    val previousValue = flag
    flag = f(previousValue)
    previousValue
  }
}
```

Two apparently useless functions

```
object Config {
  private var flag: Boolean = true

  def modifyFlag(f: Boolean => Boolean): Boolean = {
    val previousValue = flag
    flag = f(previousValue)
    previousValue
  }
}
```

```
def toggle(): Boolean =
  Config.modifyFlag(x => !x)
```

```
toggle()
// res51: Boolean = true

toggle()
// res52: Boolean = false
```

Two apparently useless functions

```
object Config {
  private var flag: Boolean = true

  def modifyFlag(f: Boolean => Boolean): Boolean = {
    val previousValue = flag
    flag = f(previousValue)
    previousValue
  }
}
```

How would you implement?

```
def disable(): Boolean = ...
```

Two apparently useless functions

```
object Config {
  private var flag: Boolean = true

  def modifyFlag(f: Boolean => Boolean): Boolean = {
    val previousValue = flag
    flag = f(previousValue)
    previousValue
  }
}
```

How would you implement?

```
def disable(): Boolean =
  Config.modifyFlag(_ => false)
```

Two apparently useless functions

```
object Config {
  private var flag: Boolean = true

  def modifyFlag(f: Boolean => Boolean): Boolean = {
    val previousValue = flag
    flag = f(previousValue)
    previousValue
  }
}
```

How would you implement?

```
def disable(): Boolean =
  Config.modifyFlag(_ => false)
```

```
def disable(): Boolean =
  Config.modifyFlag(constant(false))
```

Two apparently useless functions

```
object Config {
  private var flag: Boolean = true

  def modifyFlag(f: Boolean => Boolean): Boolean = {
    val previousValue = flag
    flag = f(previousValue)
    previousValue
  }
}
```

How would you implement?

```
def getFlag: Boolean = ...
```

Two apparently useless functions

```
object Config {
  private var flag: Boolean = true

  def modifyFlag(f: Boolean => Boolean): Boolean = {
    val previousValue = flag
    flag = f(previousValue)
    previousValue
  }
}
```

How would you implement?

```
def getFlag: Boolean =
  Config.modifyFlag(identity)
```

Consistent API

```
trait Config {

  def modifyFlag(f: Boolean => Boolean): Boolean

  def toggle(): Boolean =
    modifyFlag(x => !x)

  def disable(): Boolean =
    modifyFlag(constant(false))

  def enable(): Boolean =
    modifyFlag(constant(true))

  def get: Boolean =
    modifyFlag(identity)
}
```

Consistent API

```
trait Config {

  def modifyFlag(f: Boolean => Boolean): Boolean

  def toggle(): Boolean =
    modifyFlag(x => !x)

  def disable(): Boolean =
    modifyFlag(constant(false))

  def enable(): Boolean =
    modifyFlag(constant(true))

  def get: Boolean =
    modifyFlag(identity)
}
```

```
class FastConfig extends Config {
  def modifyFlag(f: Boolean => Boolean): Boolean =
    ...

  def get: Boolean =
    ...
}
```

```
test("get is consistent"){
  forAll((config: FastConfig) =>
    config.get shouldEqual
      config.modifyFlag(identity)
  )
}
```

What is the type of `identityVal`?

```
def identity[A](value: A): A =
  value
```

```
val identityVal = identity _
```

What is the type of `identityVal`?

```
def identity[A](value: A): A =
  value
```

```
val identityVal: Nothing => Nothing = identity _
```

```
identityVal(4)
// error: type mismatch;
//  found   : Int(4)
//  required: Nothing
// reverse(List(1,2,3,4))
//                   ^
```

What is the type of `identityVal`?

```
def identity[A](value: A): A =
  value
```

```
val identityVal: Int => Int = identity[Int] _
```

```
identityVal(4)
// res58: Int = 4
```

Functions as output

```
def truncate(digits: Int, number: Double): String =
  BigDecimal(number)
    .setScale(digits, BigDecimal.RoundingMode.FLOOR)
    .toDouble
    .toString
```

```
truncate(2, 0.123456789)
// res59: String = "0.12"

truncate(5, 0.123456789)
// res60: String = "0.12345"
```

Functions as output

```
def truncate(digits: Int, number: Double): String =
  BigDecimal(number)
    .setScale(digits, BigDecimal.RoundingMode.FLOOR)
    .toDouble
    .toString

def truncate2D(number: Double): String = truncate(2, number)
def truncate5D(number: Double): String = truncate(5, number)
```

```
truncate2D(0.123456789)
// res62: String = "0.12"

truncate5D(0.123456789)
// res63: String = "0.12345"
```

Functions as output

```
def publishEvent(
  environment: Environment, // DEV, UAT, PRD
  topic: String,
  event: Event,
  parameters: Config
): Unit
```

Functions as output

```
def publishEvent(
  environment: Environment, // DEV, UAT, PRD
  topic: String,
  event: Event,
  parameters: Config
): Unit
```

```
def publishEventToDev(
  topic: String,
  event: Event,
  parameters: Config
): Unit =
  publishEvent(DEV, topic, event, parameters)
```

```
def publishEventToUat(
  topic: String,
  event: Event,
  parameters: Config
): Unit =
  publishEvent(UAT, topic, event, parameters)
```

Functions as output

```
def publishEvent(
  environment: Environment, // DEV, UAT, PRD
  topic: String,
  event: Event,
  parameters: Config
): Unit
```

```
def publishEventToDev = publishEvent(DEV)
def publishEventToUat = publishEvent(UAT)
```

Pseudocode

Functions as output

```
def truncate(digits: Int, number: Double): String
```

```
truncate(2, 0.123456789)
// res65: String = "0.12"
```

```
def truncate(digits: Int): Double => String
```

```
truncate(2)(0.123456789)
// res67: String = "0.12"
```

Functions as output

```
def truncate(digits: Int, number: Double): String
```

```
truncate(2, 0.123456789)
// res65: String = "0.12"
```

```
def truncate(digits: Int): Double => String
```

```
truncate(2)(0.123456789)
// res67: String = "0.12"
```

Currying

```
val function3: (Int ,  Int  ,  Int) => Int

val function3: Int => (Int => (Int  => Int))
```

Functions as output

```
def truncate(digits: Int, number: Double): String
```

```
truncate(2, 0.123456789)
// res69: String = "0.12"
```

```
def truncate(digits: Int): Double => String
```

```
truncate(2)(0.123456789)
// res71: String = "0.12"
```

Currying

```
val function3: (Int ,  Int  ,  Int) => Int

val function3: Int =>  Int =>  Int  => Int
```

Partial function application

```
def truncate(digits: Int): Double => String =
  (number: Int) => ...
```

```
val truncate2D = truncate(2)
val truncate5D = truncate(5)
```

Partial function application

```
def truncate(digits: Int): Double => String =
  (number: Int) => ...
```

```
val truncate2D = truncate(2)
val truncate5D = truncate(5)
```

```
truncate2D(0.123456789)
// res72: String = "0.12"

truncate5D(0.123456789)
// res73: String = "0.12345"
```

Syntax

Uncurried

```
def truncate(digits: Int, number: Double): String
```

Curried

```
def truncate(digits: Int)(number: Double): String

def truncate(digits: Int): Double => String

val truncate: Int => Double => String
```

Conversion (Currying)

```
def truncate(digits: Int, number: Double): String
```

Conversion (Currying)

```
def truncate(digits: Int, number: Double): String
```

```
truncate _
// res75: (Int, Double) => String = <function2>
```

Conversion (Currying)

```
def truncate(digits: Int, number: Double): String
```

```
truncate _
// res75: (Int, Double) => String = <function2>
```

```
(truncate _).curried
// res76: Int => Double => String = scala.Function2$$Lambda$5355/0x0000000101ace840@6e15cec7
```

Type inference

```
case class Pair[A](first: A, second: A) {
  def zipWith[B, C](other: Pair[B], combine: (A, B) => C): Pair[C] = ...
}
```

```
Pair(0, 2).zipWith(Pair(7, 3), (x: Int, y: Int) => x + y)
// res77: Pair[Int] = Pair(7, 5)

Pair(2, 3).zipWith(Pair("Hello ", "World "), replicate)
// res78: Pair[String] = Pair("Hello Hello ", "World World World ")
```

Type inference

```
case class Pair[A](first: A, second: A) {
  def zipWith[B, C](other: Pair[B], combine: (A, B) => C): Pair[C] = ...
}
```

```
Pair(0, 2).zipWith(Pair(7, 3), (x, y) => x + y)
// error: missing parameter type
// Pair(0, 2).zipWith(Pair(7, 3), (x, y) => x + y)
//                                    ^
```

Type inference

```
case class Pair[A](first: A, second: A) {
  def zipWith[B, C](other: Pair[B])(combine: (A, B) => C): Pair[C] = ...
}
```

```
Pair(0, 2).zipWith(Pair(7, 3))((x, y) => x + y)
// res81: Pair[Int] = Pair(7, 5)
```

Type inference

```
case class Pair[A](first: A, second: A) {
  def zipWith[B, C](other: Pair[B])(combine: (A, B) => C): Pair[C] = ...
}
```

```
Pair(0, 2).zipWith(Pair(7, 3))((x, y) => x + y)
// res81: Pair[Int] = Pair(7, 5)
```

```
Pair(0, 2).zipWith(Pair(7, 3))(_ + _)
// res82: Pair[Int] = Pair(7, 5)
```

Scala API design

```scala
def zip[A, B, C](first: List[A], second: List[B])(f: (A, B) => C): List[C]
```

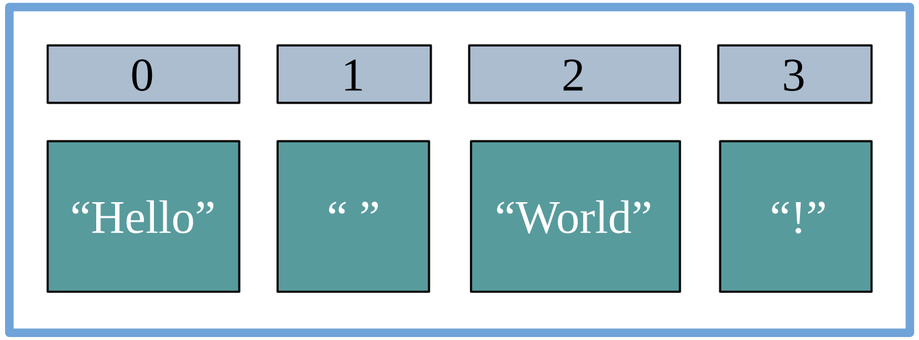
```scala
def mapTwice[A, B, C](values: List[A])(f: A => B)(g: B => C): List[C]
```

Summary

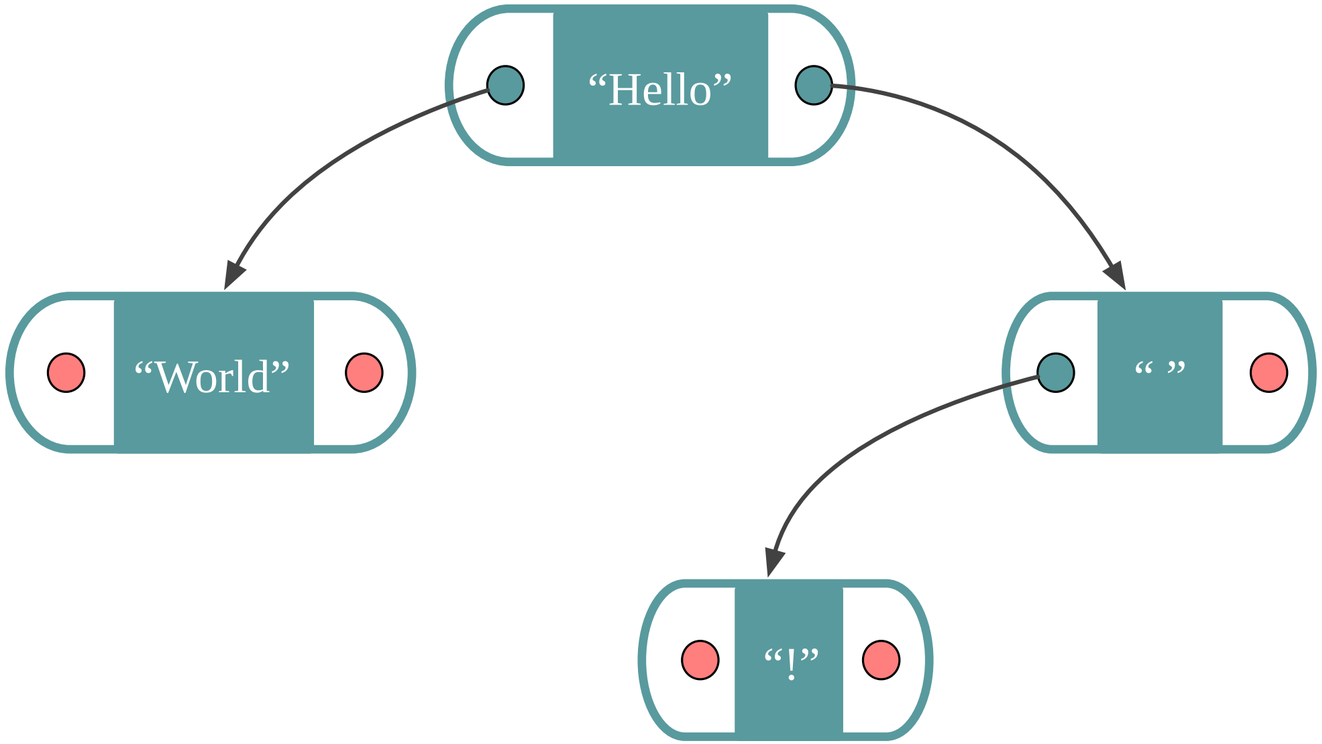
- Currying
- A curried function can be partially applied
- It als help type inference in Scala 2

Iteration

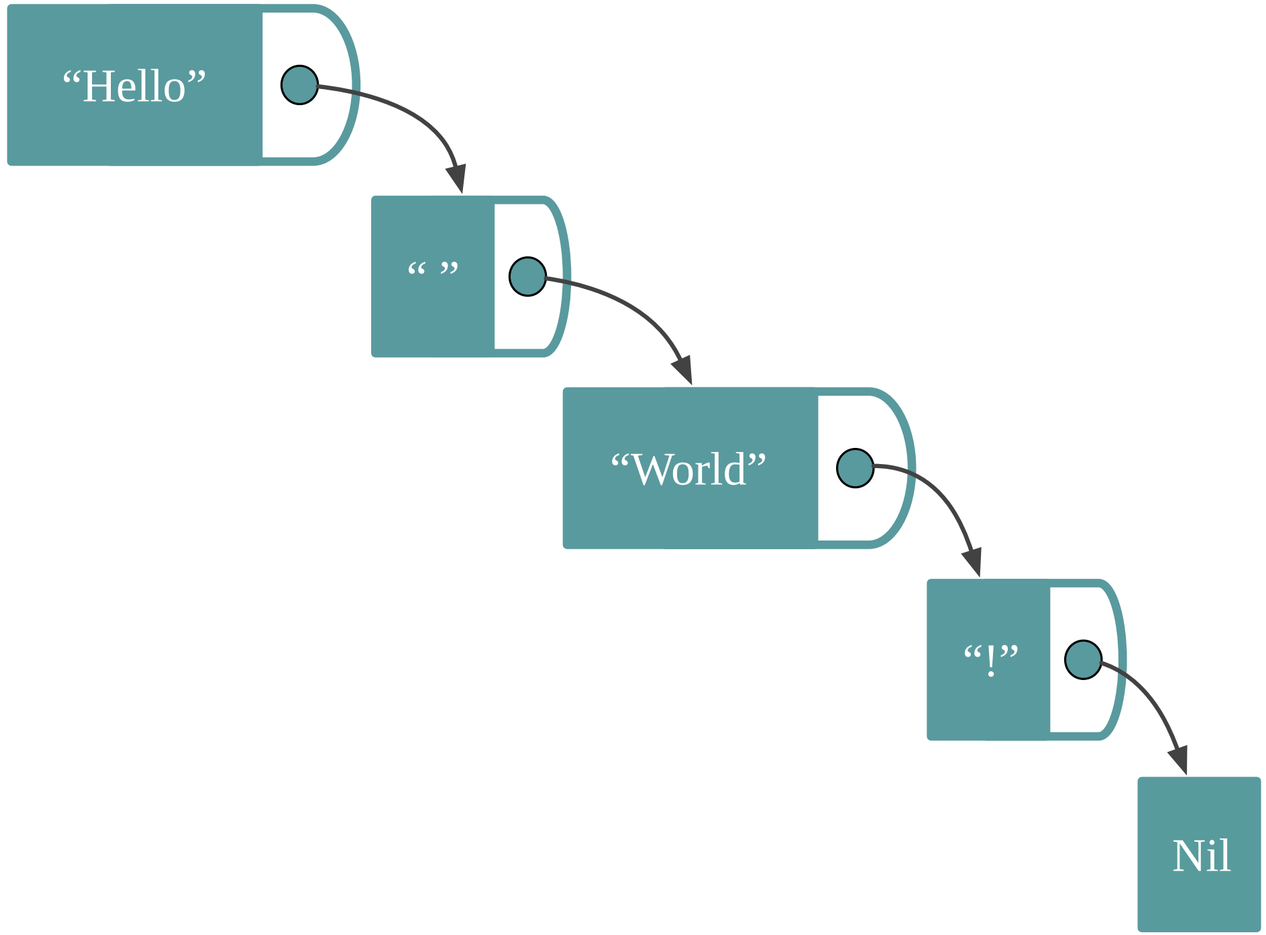
Array



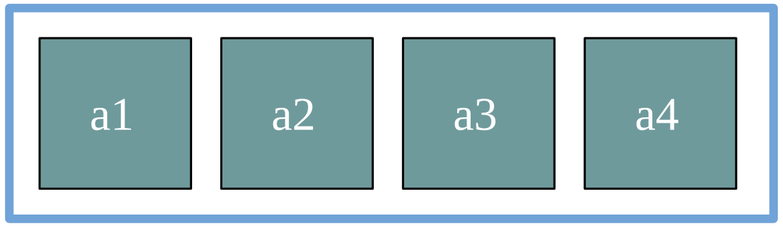
Tree



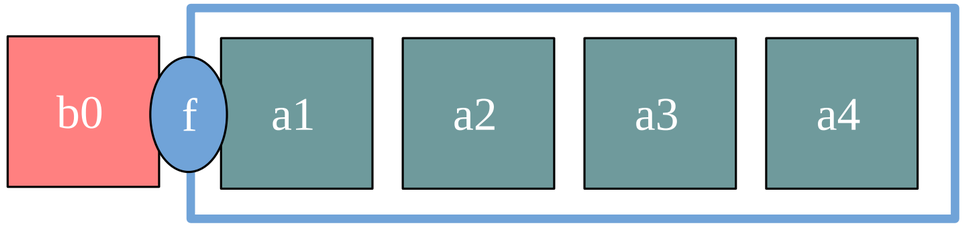
Linked List



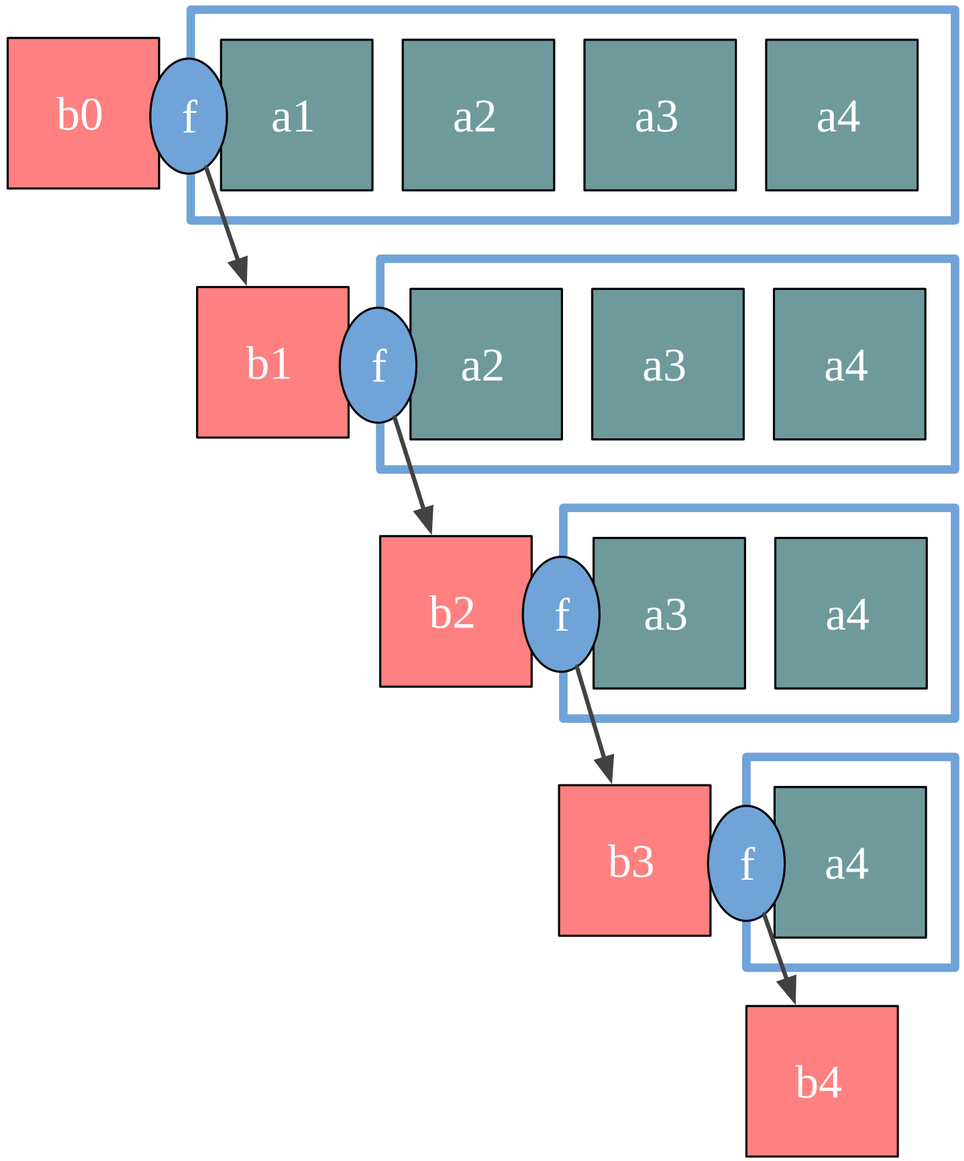
Folding



FoldLeft

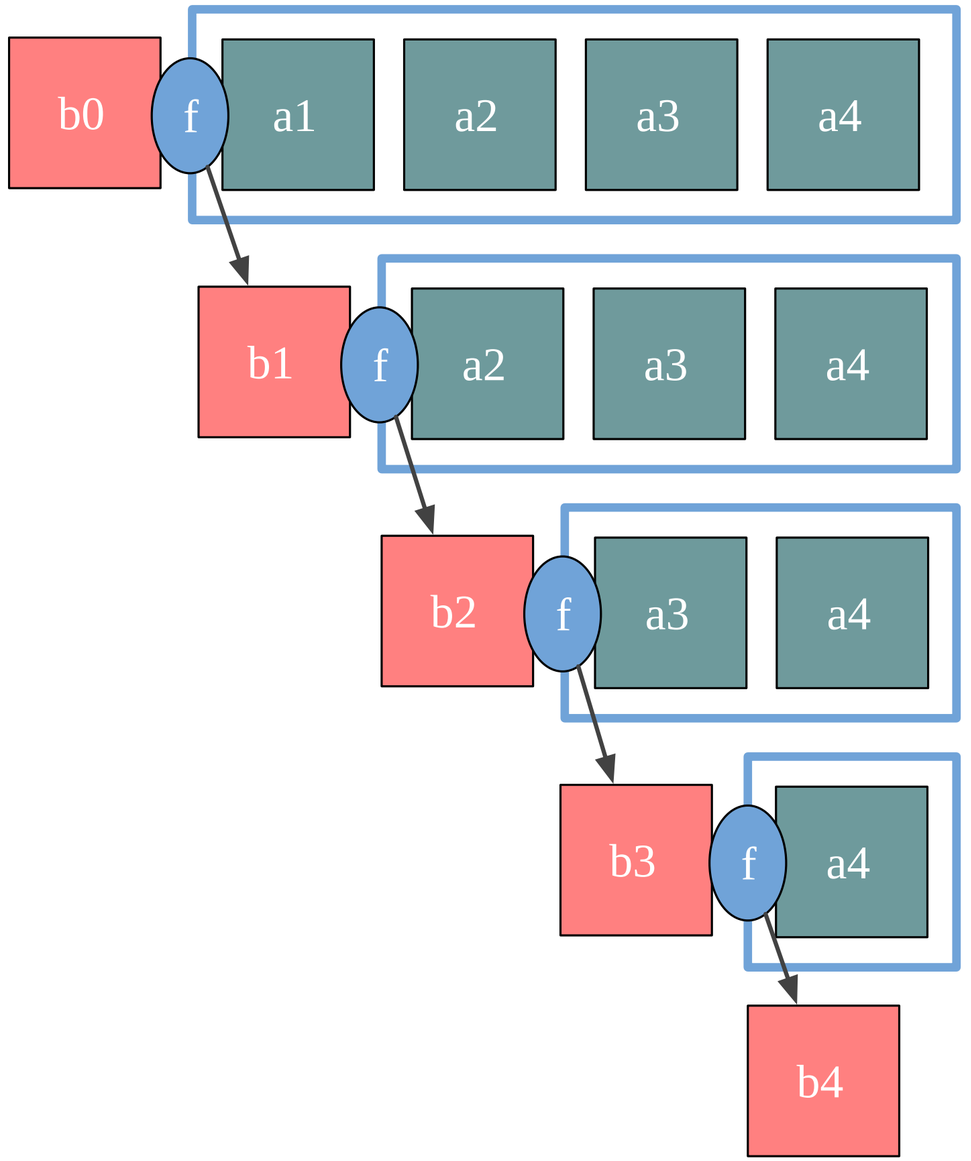


FoldLeft



FoldLeft

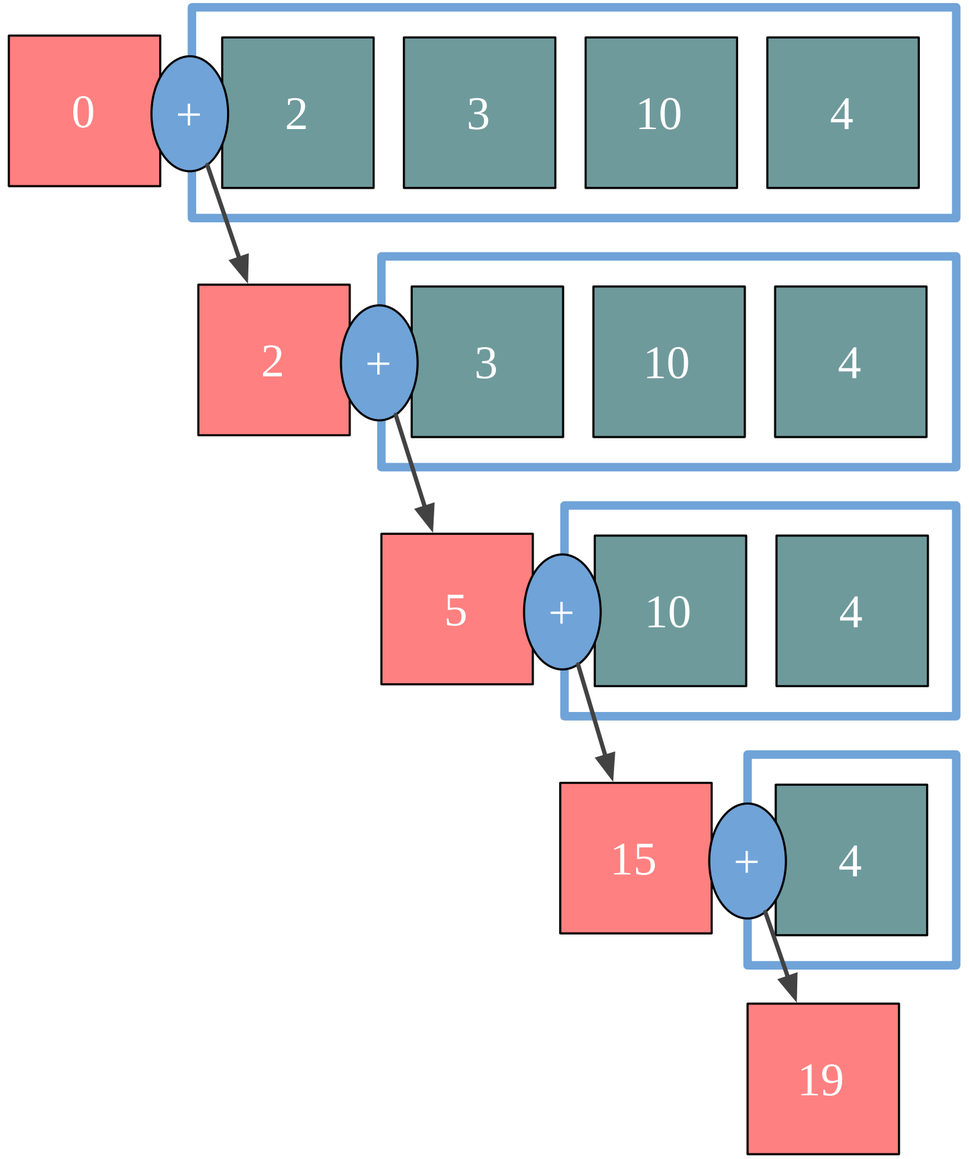
```
def foldLeft[A, B](fa: List[A], b: B)(f: (B, A) => B): B = {
  var acc = b
  for (a <- fa) {
    acc = f(acc, a)
  }
  acc
}
```



FoldLeft

```
def sum(xs: List[Int]): Int =
  foldLeft(xs, 0)(_ + _)
```

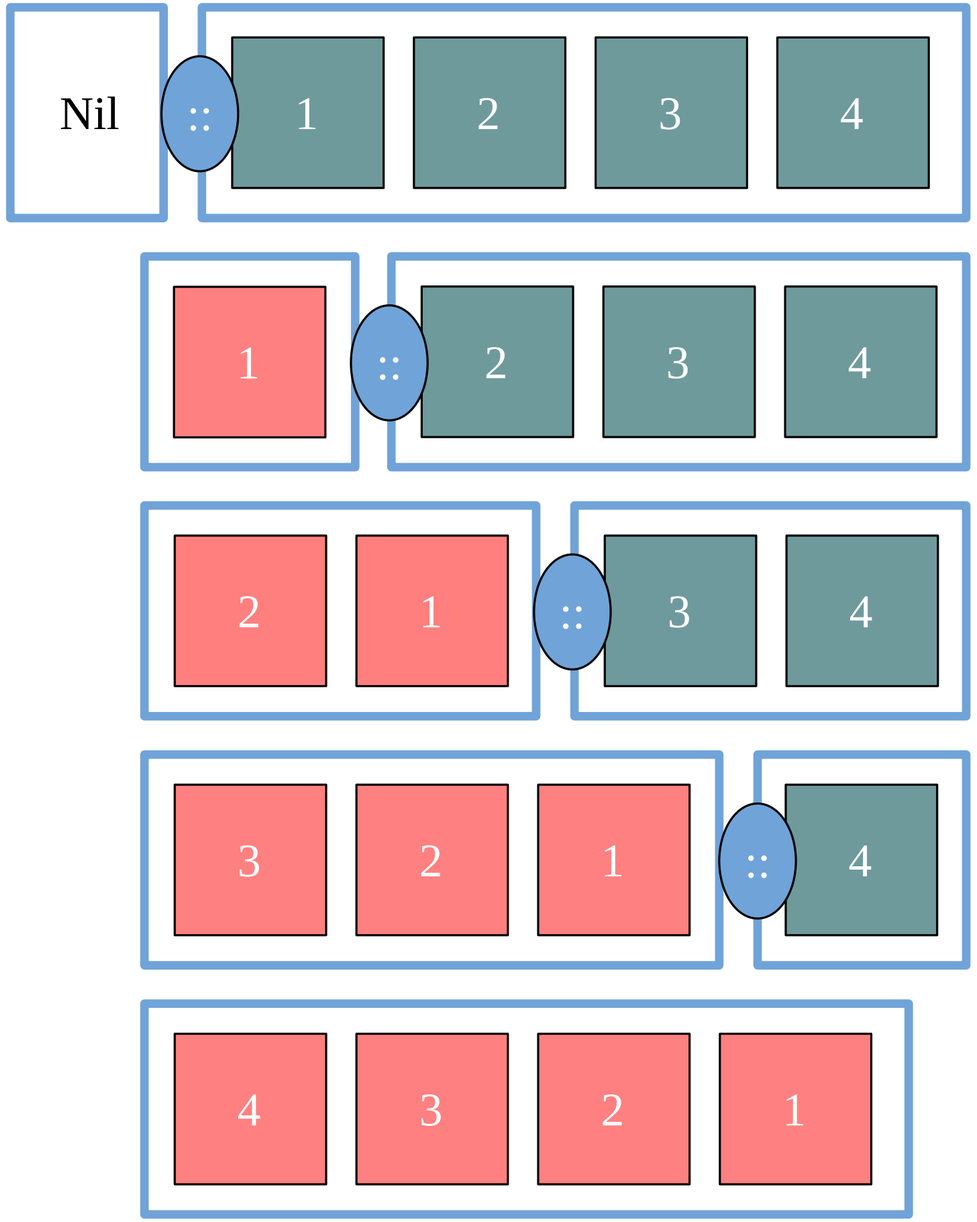
```
sum(List(2,3,10,4))
// res84: Int = 19
```



FoldLeft

```
def reverse[A](xs: List[A]): List[A] =
  foldLeft(xs, List.empty[A])((acc, a) => a :: acc)
```

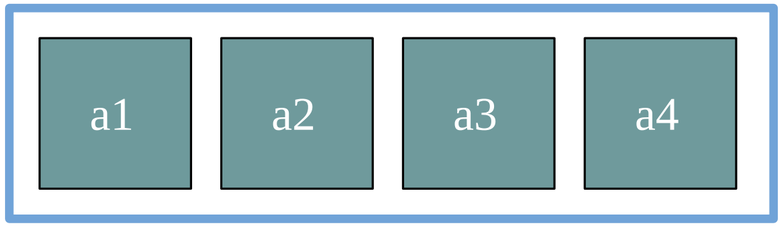
```
reverse(List(1,2,3,4))
// res85: List[Int] = List(4, 3, 2, 1)
```



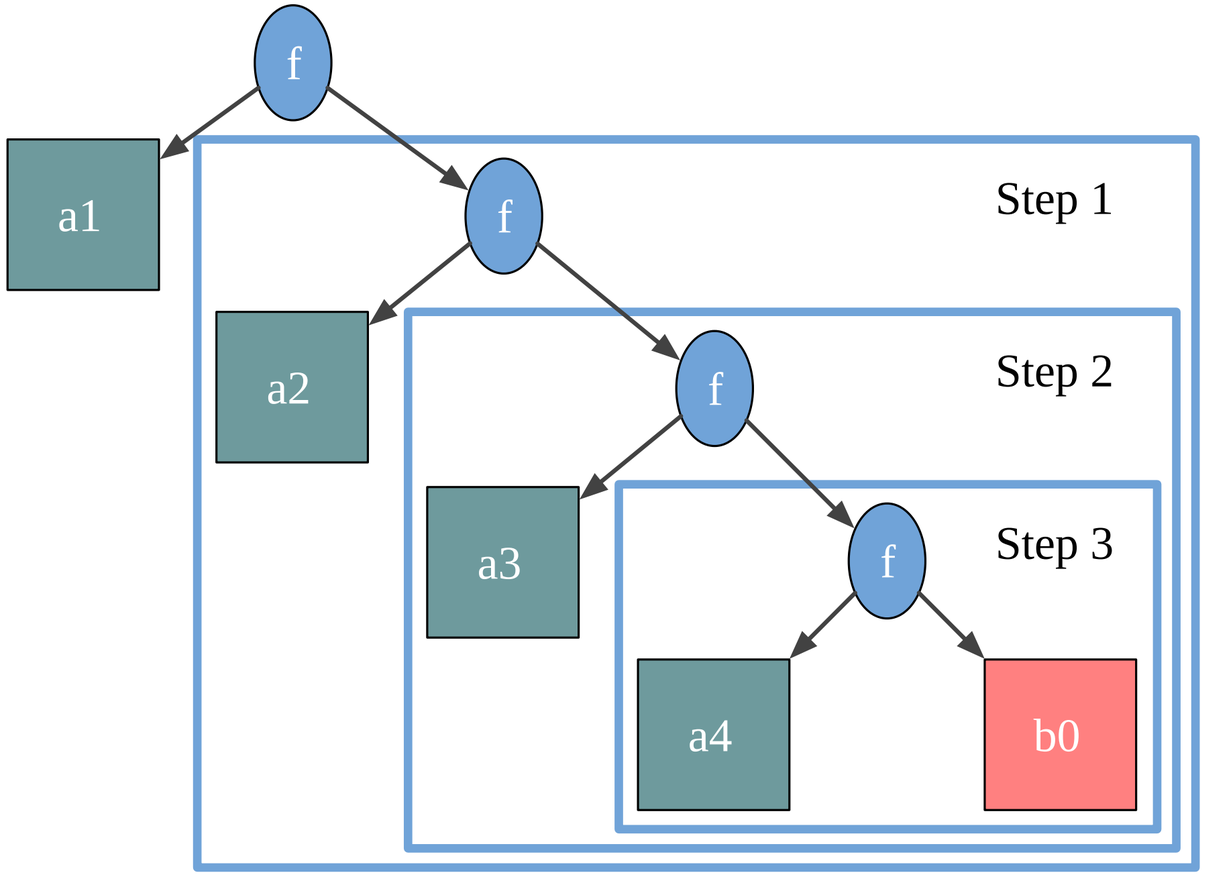
Exercise 3c-f

`exercises.function.FunctionExercises.scala`

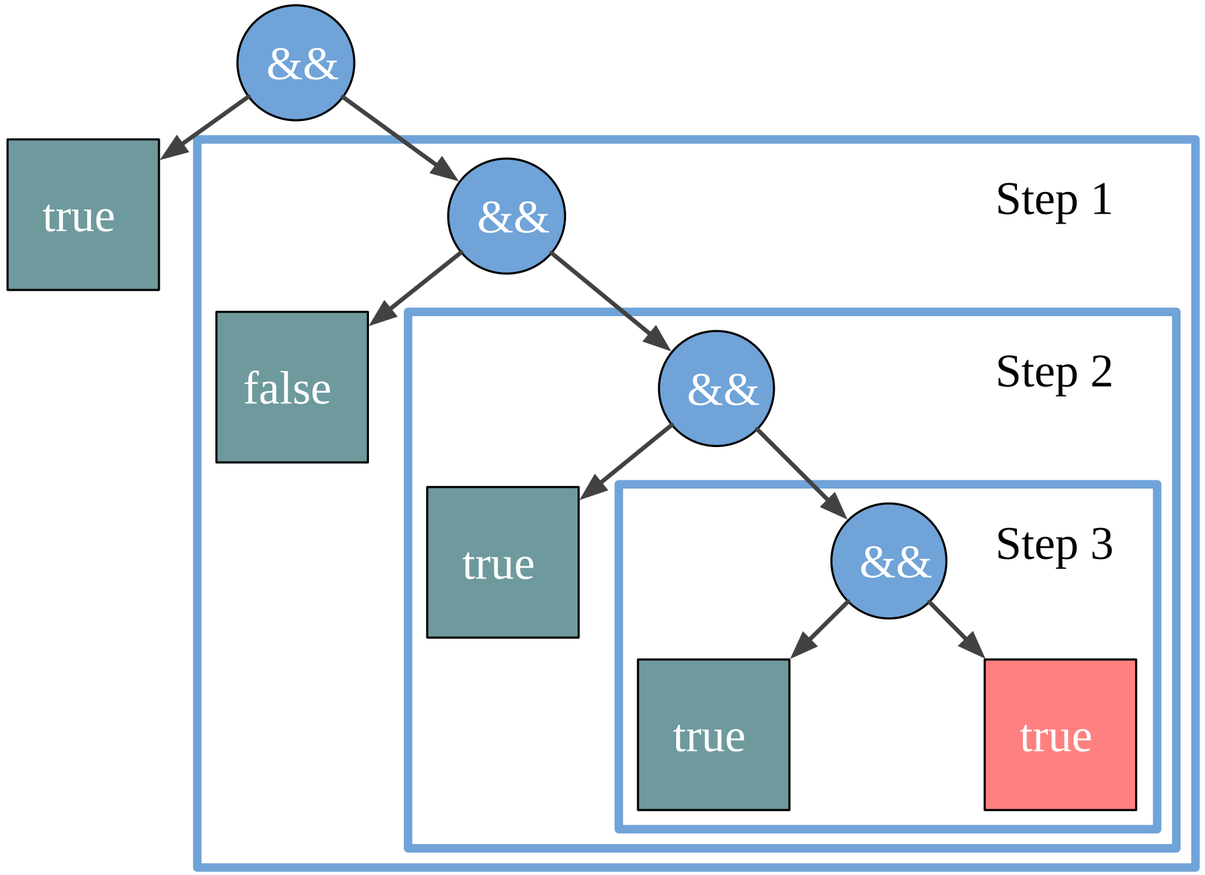
Folding



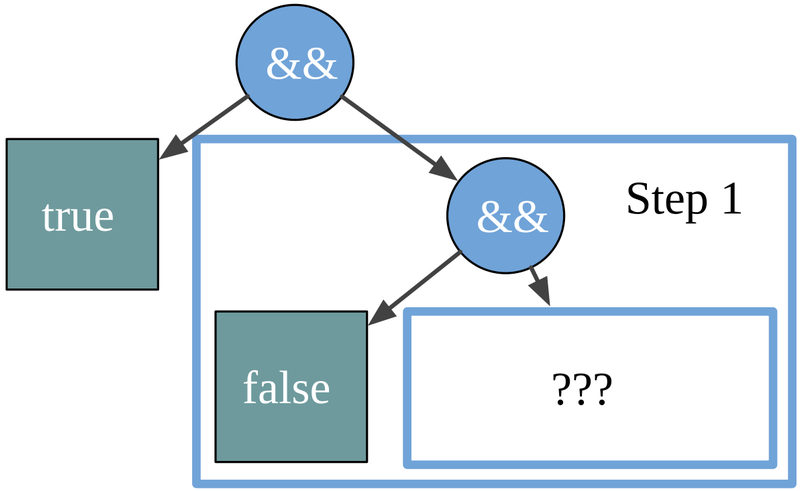
FoldRight



FoldRight

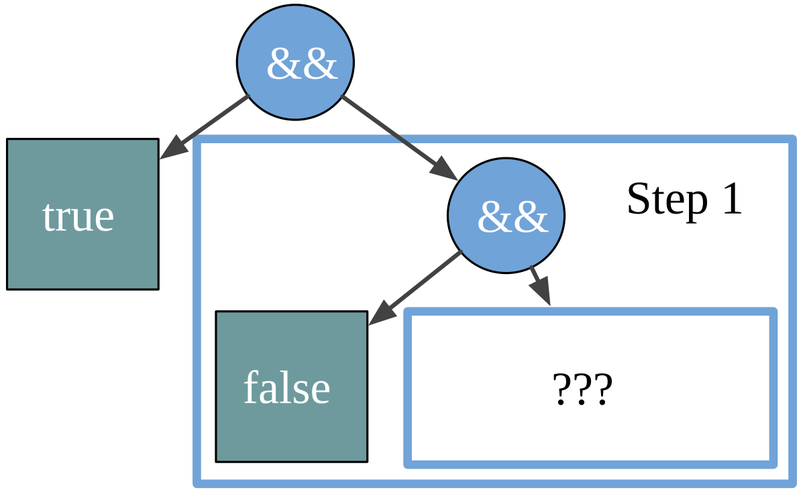


FoldRight is lazy

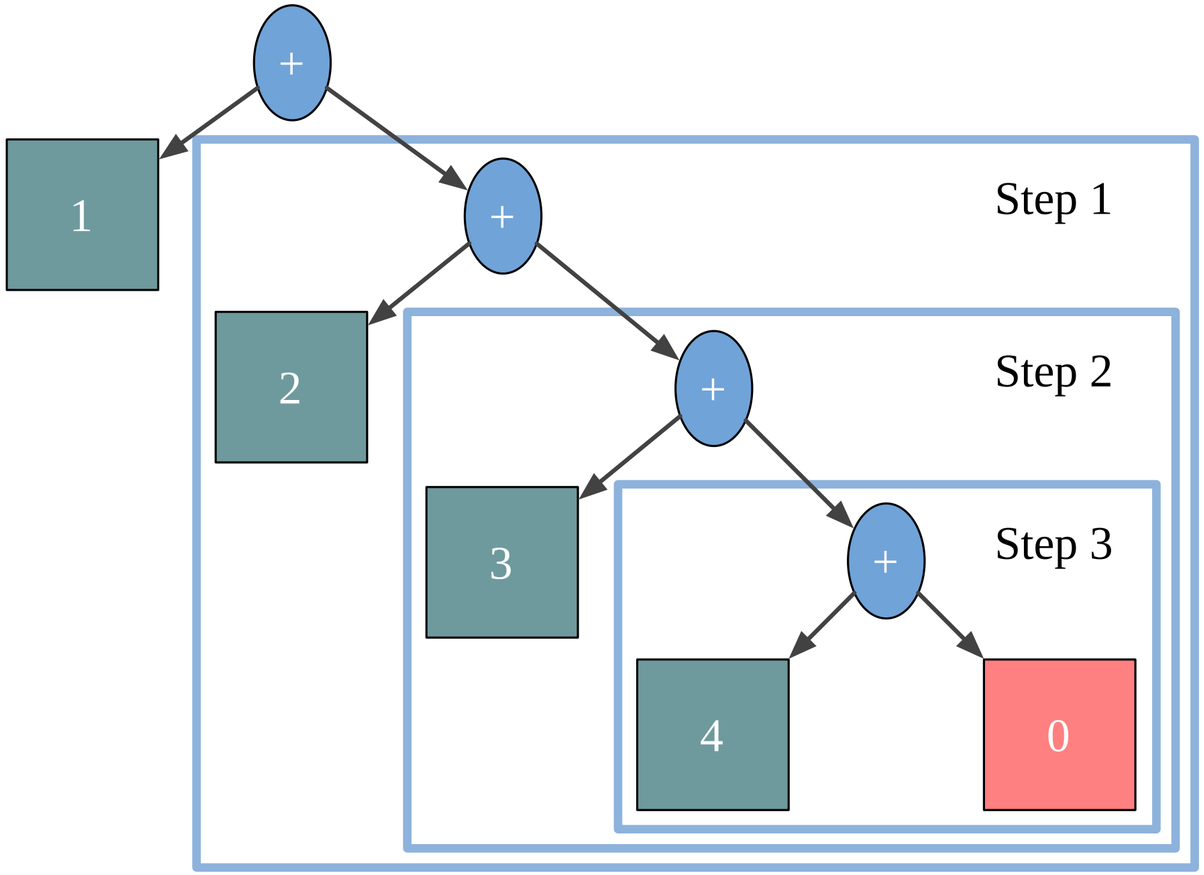


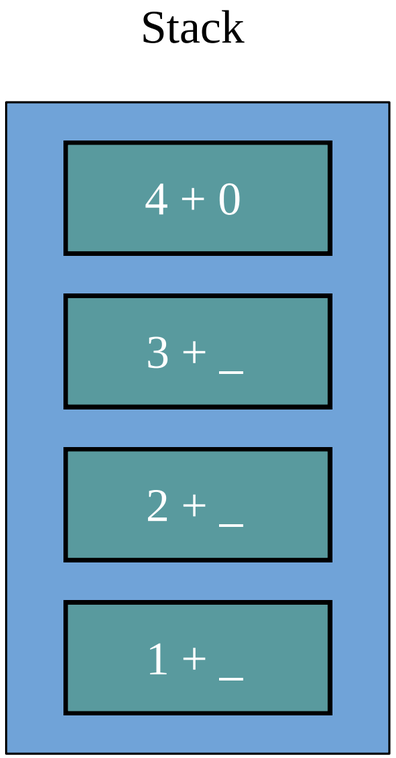
FoldRight is lazy

```
def foldRight[A, B](xs: List[A], b: B)(f: (A, `=> B`) => B): B =
  xs match {
    case Nil    => b
    case h :: t => f(h, `foldRight(t, b)(f)`)
  }
```



FoldRight is NOT always stack safe





FoldRight replaces constructors

```
sealed trait List[A]

case class Nil[A]() extends List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
```

```
val xs: List[Int] = Cons(1, Cons(2, Cons(3, Nil())))
```

FoldRight replaces constructors

```
sealed trait List[A]

case class Nil[A]() extends List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
```

```
val xs: List[Int] = Cons(1, Cons(2, Cons(3, Nil())))
```

```
def foldRight[A, B](list: List[A], `b: B`)(f: `(A, => B) => B`): B

foldRight(xs, b)(f) == foldRight(Cons(1, Cons(2, Cons(3, Nil()))), b)(f)
                    ==           f   (1, f   (2, f   (3, b    )))
```

FoldRight replaces constructors

```
sealed trait List[A]

case class Nil[A]() extends List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
```

```
val xs: List[Int] = Cons(1, Cons(2, Cons(3, Nil())))
```

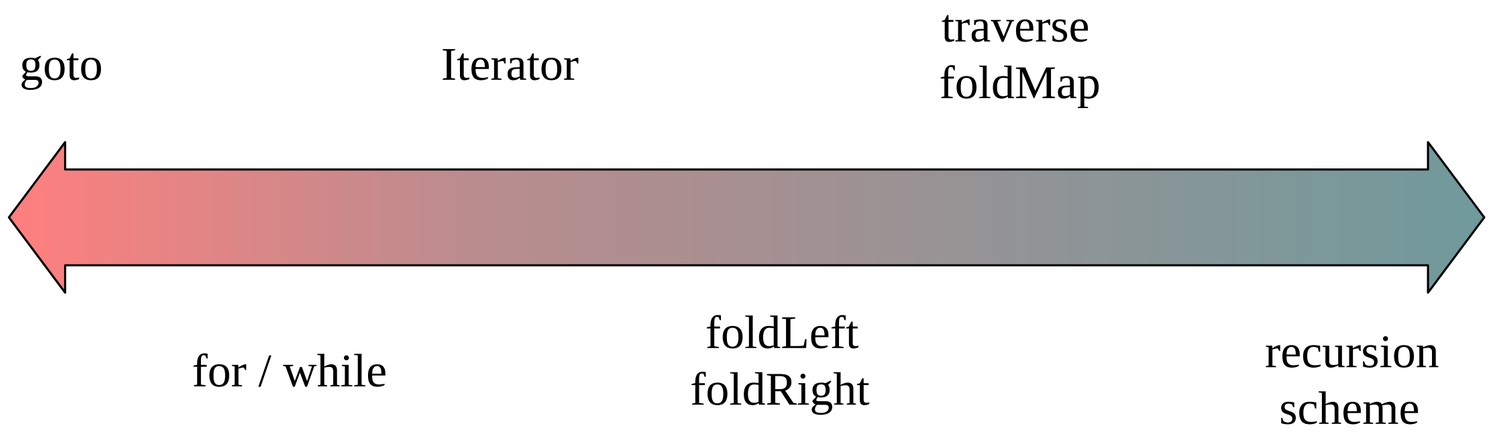
```
def foldRight[A, B](list: List[A], `b: B`)(f: `(A, => B) => B`): B

foldRight(xs, b)(f) == foldRight(Cons(1, Cons(2, Cons(3, Nil()))), b)(f)
                    ==           f   (1, f   (2, f   (3, b    )))
```

Home exercise: How would you "replace constructors" for an Option or a Binary Tree?

Finish Exercise 3

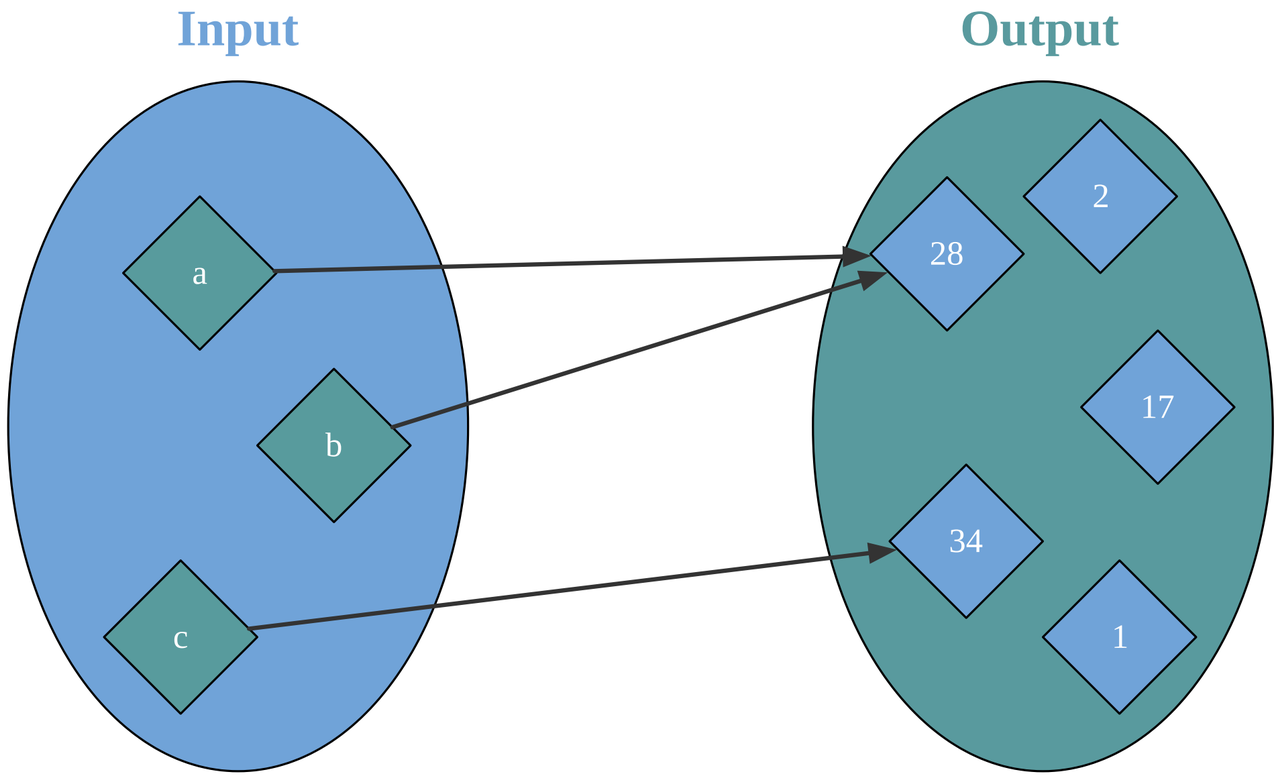
`exercises.function.FunctionExercises.scala`

Different level of abstractions



Pure function

Pure functions are mappings between two sets

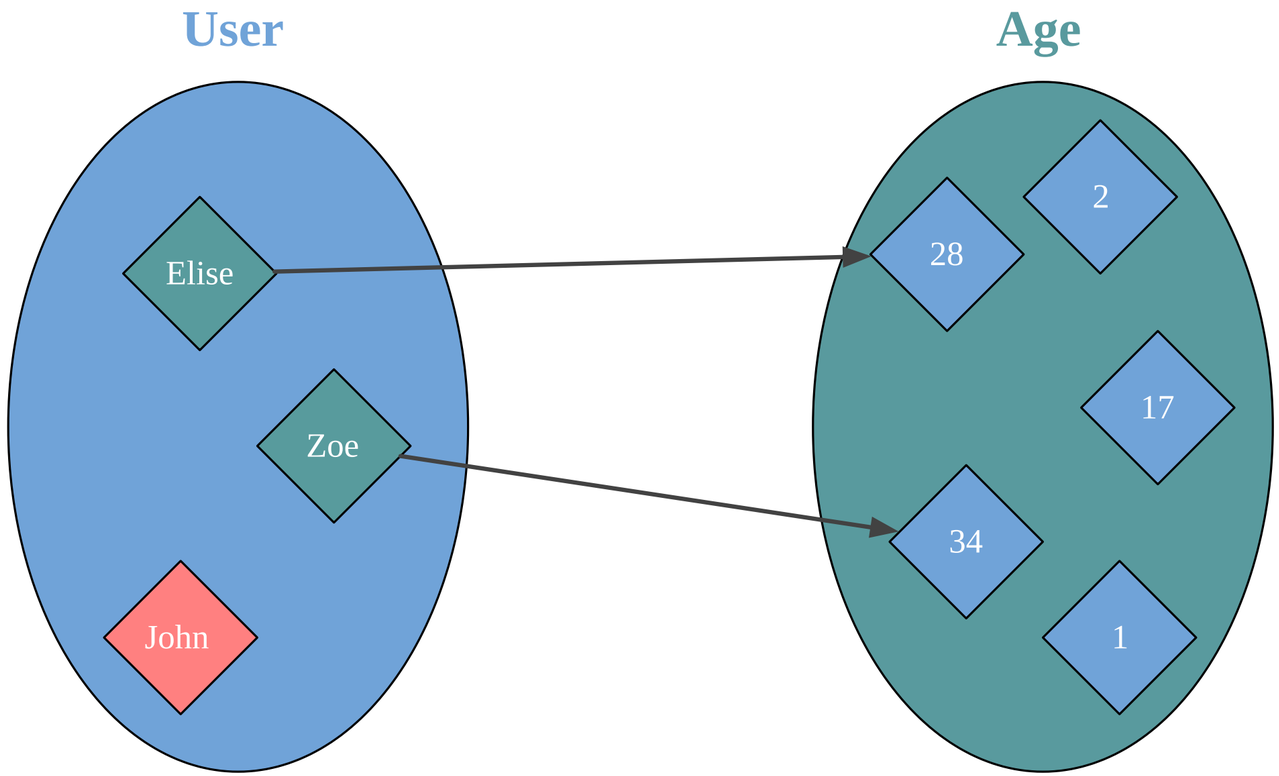


Programming function

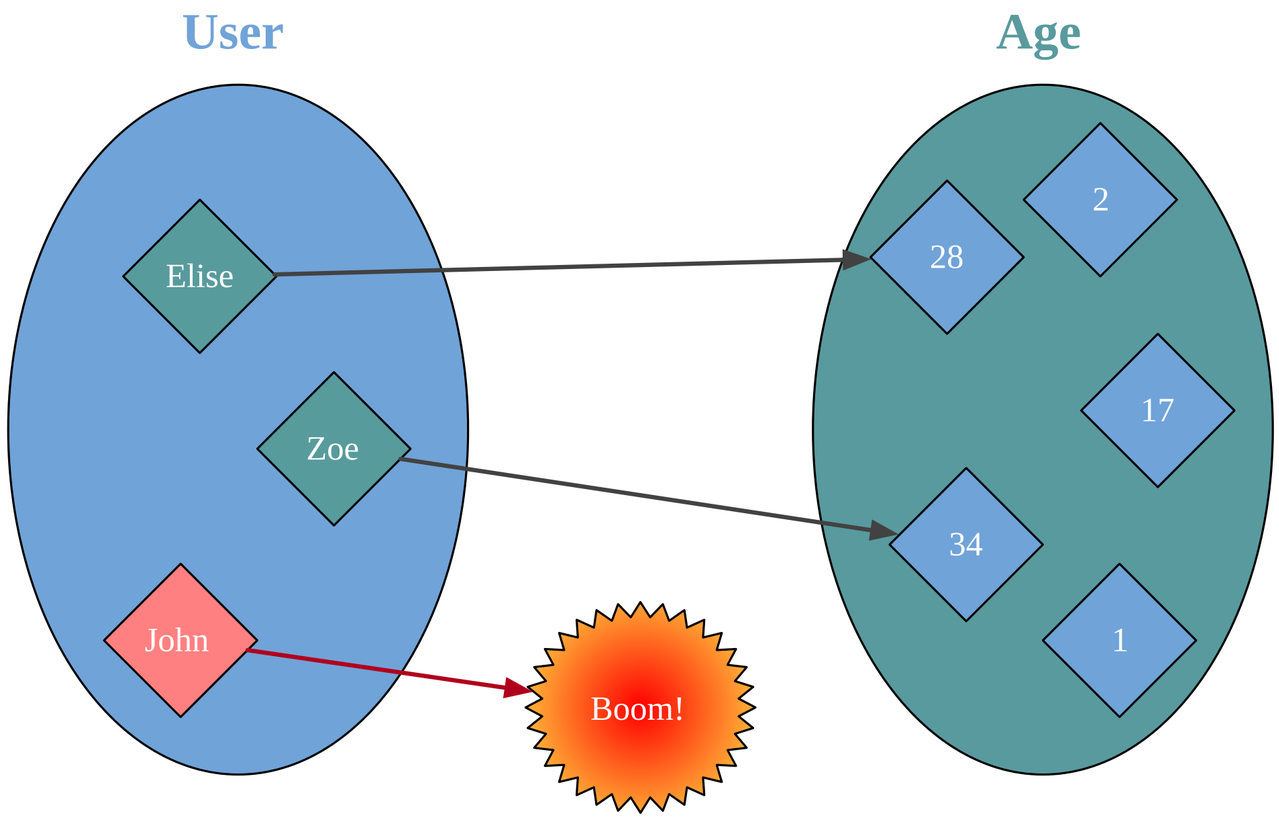
!=

Pure function

Partial function



Partial function



Partial function

```
def head(list: List[Int]): Int =
  list match {
    case x :: xs => x
  }
```

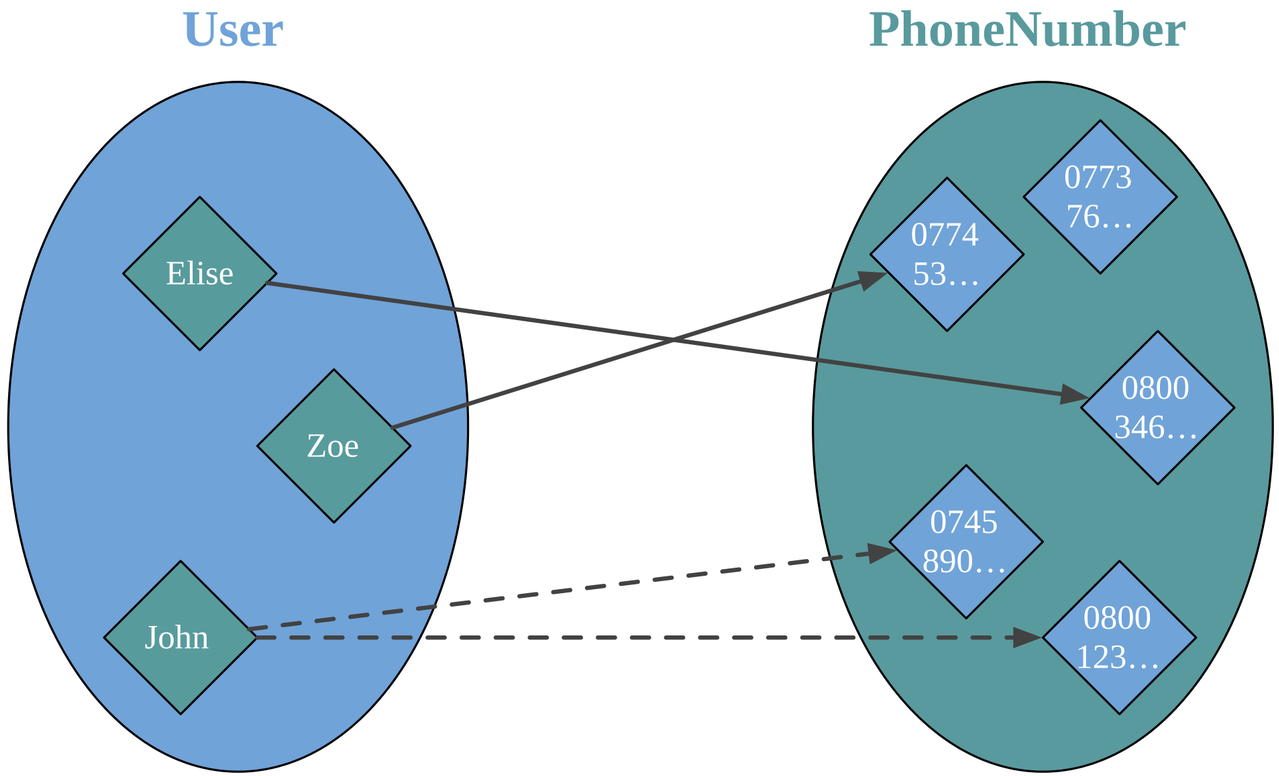
```
head(Nil)
// scala.MatchError: List() (of class scala.collection.immutable.Nil$)
//     at repl.Session$App86.head(1-Function.html:991)
//     at repl.Session$App86$$anonfun$164.apply$mcI$sp(1-Function.html:1000)
//     at repl.Session$App86$$anonfun$164.apply(1-Function.html:1000)
//     at repl.Session$App86$$anonfun$164.apply(1-Function.html:1000)
```

Exception

```
case class Item(id: Long, unitPrice: Double, quantity: Int)

case class Order(status: String, basket: List[Item])

def submit(order: Order): Order =
  order.status match {
    case "Draft" if order.basket.nonEmpty =>
      order.copy(status = "Submitted")
    case other =>
      throw new Exception("Invalid Command")
  }
```

```
submit(Order("Delivered", Nil))
// java.lang.Exception: Invalid Command
//     at repl.Session$App86.submit(1-Function.html:1018)
//     at repl.Session$App86$$anonfun$165.apply(1-Function.html:1026)
//     at repl.Session$App86$$anonfun$165.apply(1-Function.html:1026)
```

Nondeterministic



Nondeterministic

```
import java.util.UUID
import java.time.Instant
```

```
UUID.randomUUID()
// res87: UUID = b7f4a839-44a9-4ea7-83a9-d567655d83f1

UUID.randomUUID()
// res88: UUID = 4ce900f7-f7ac-4510-ba8d-ce9b12d46a82
```

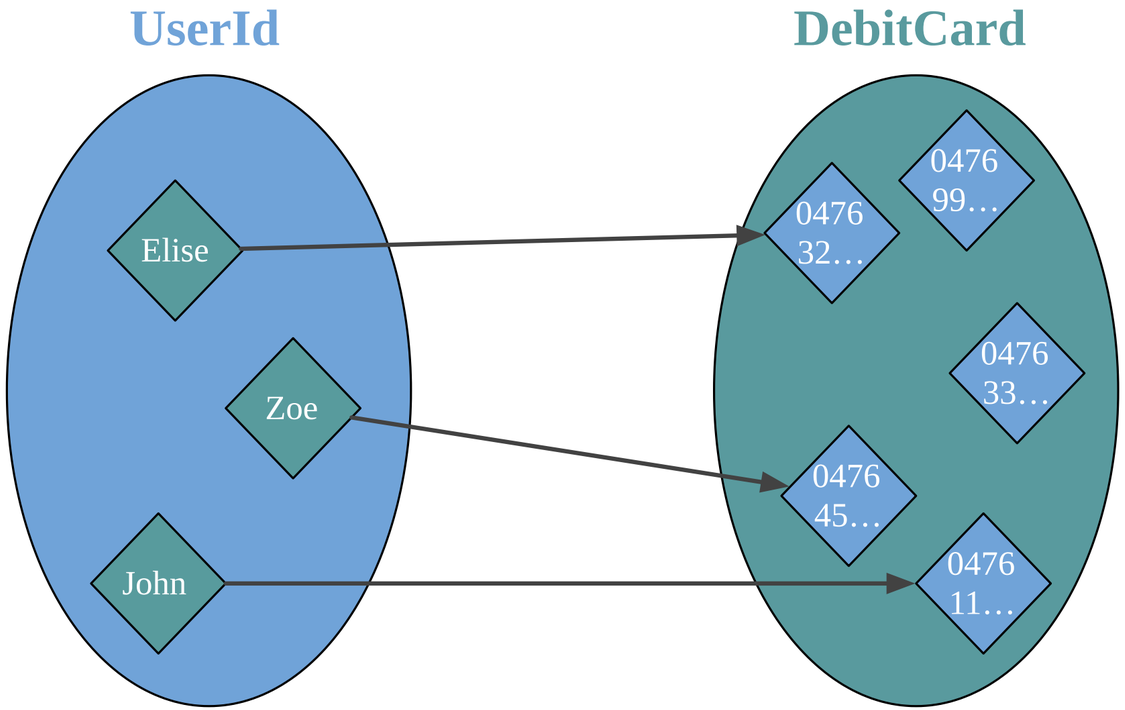
```
Instant.now()
// res89: Instant = 2020-04-15T11:10:56.947132Z

Instant.now()
// res90: Instant = 2020-04-15T11:10:56.948218Z
```

Mutation

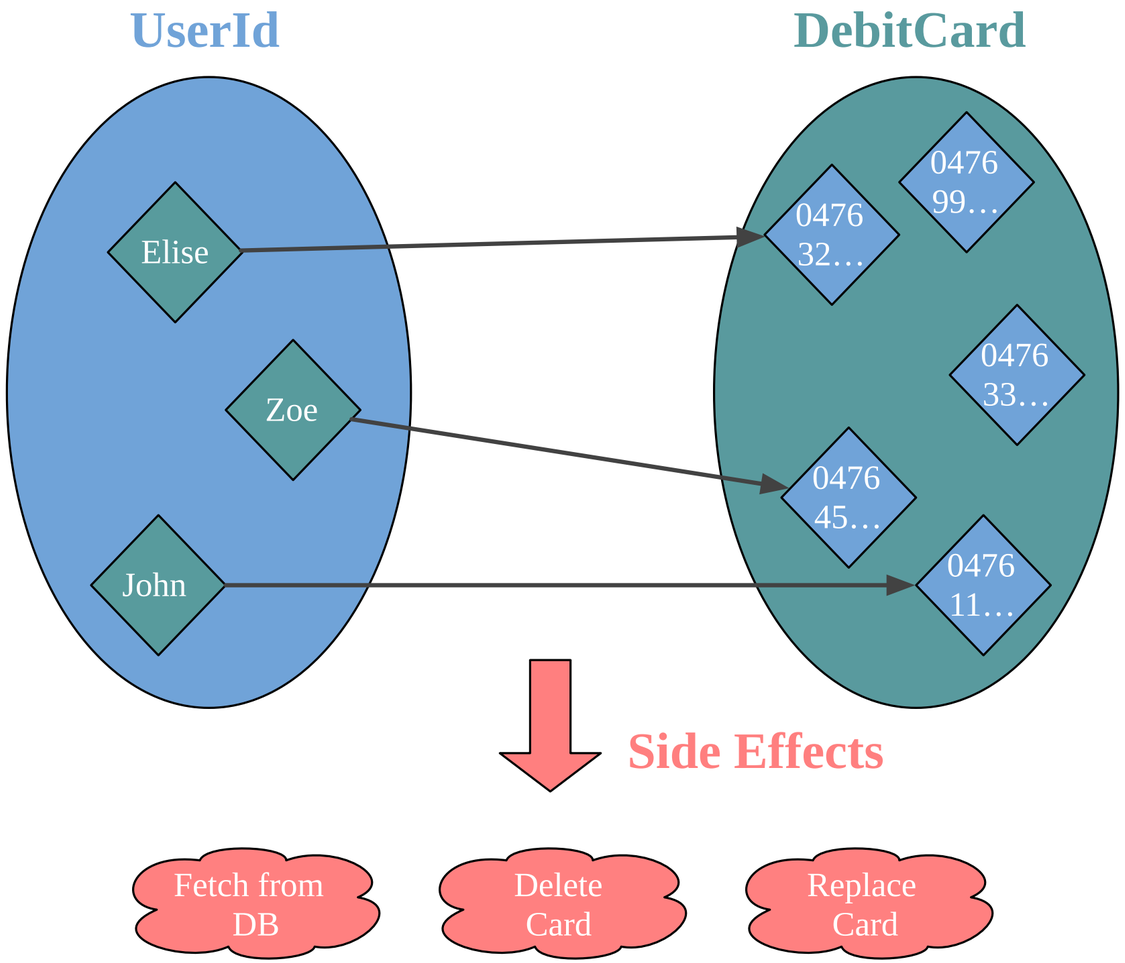
```
class User(initialAge: Int) {
  var age: Int = initialAge

  def getAge: Int = age

  def setAge(newAge: Int): Unit =
    age = newAge
}

val john = new User(24)
```

```
john.getAge
// res91: Int = 24

john.setAge(32)

john.getAge
// res93: Int = 32
```

Side effect


UserId
DebitCard
Elise
Zoe
John
0476 32…
0476 99…
0476 33…
0476 45…
0476 11…

Side effect


UserId
DebitCard
Elise
Zoe
John
0476 32…
0476 99…
0476 33…
0476 45…
0476 11…
Side Effects
Fetch from DB
Delete Card
Replace Card

Side effect

```
def println(message: String): Unit = ...
```

```
val x = println("Hello")
// Hello
```

Side effect

```
def println(message: String): Unit = ...
```

```
val x = println("Hello")
// Hello
```

```
scala> scala.io.Source.fromURL("http://google.com")("ISO-8859-1").take(100).mkString
res21: String = <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="fr"><head>
```

Side effect

```
def println(message: String): Unit = ...
```

```
val x = println("Hello")
// Hello
```

```
scala> scala.io.Source.fromURL("http://google.com")("ISO-8859-1").take(100).mkString
res21: String = <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="fr"><head>
```

```
var x: Int = 0

def count(): Int = {
  x = x + 1
  x
}
```

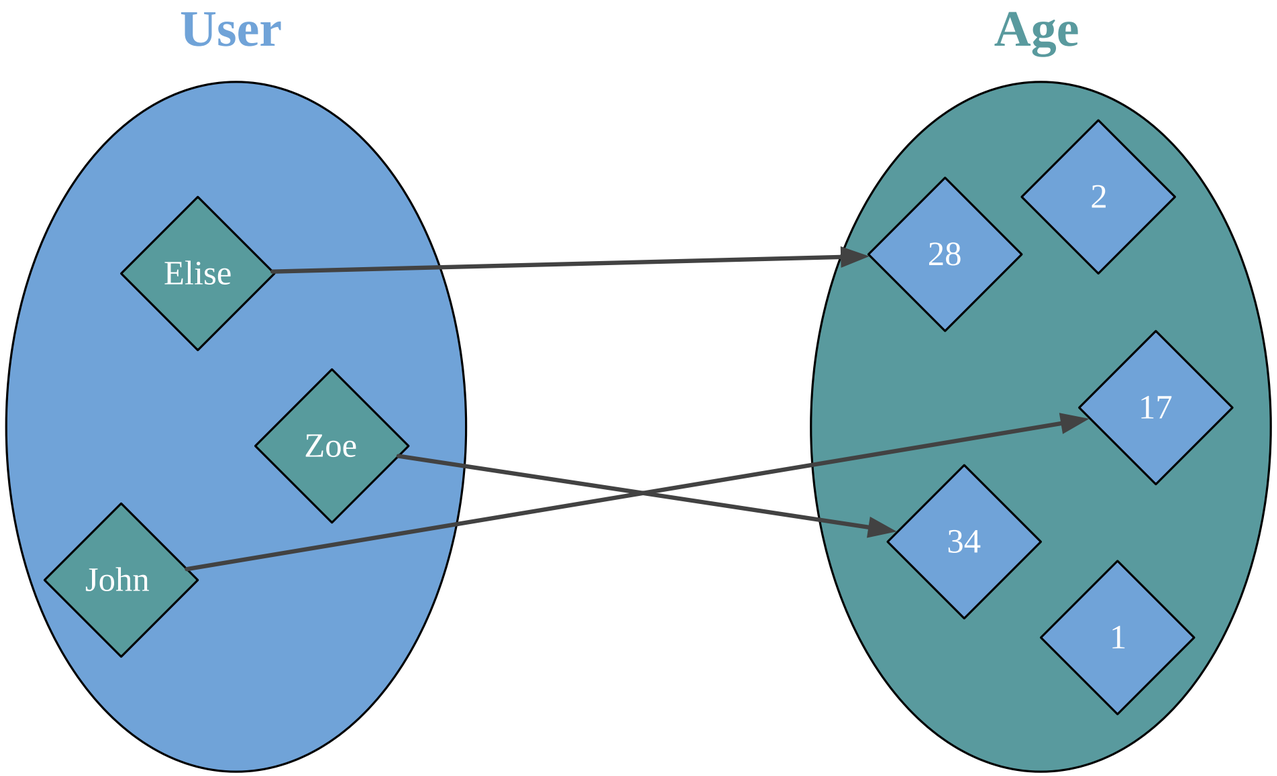
A function without side effects only returns a value

Pure function

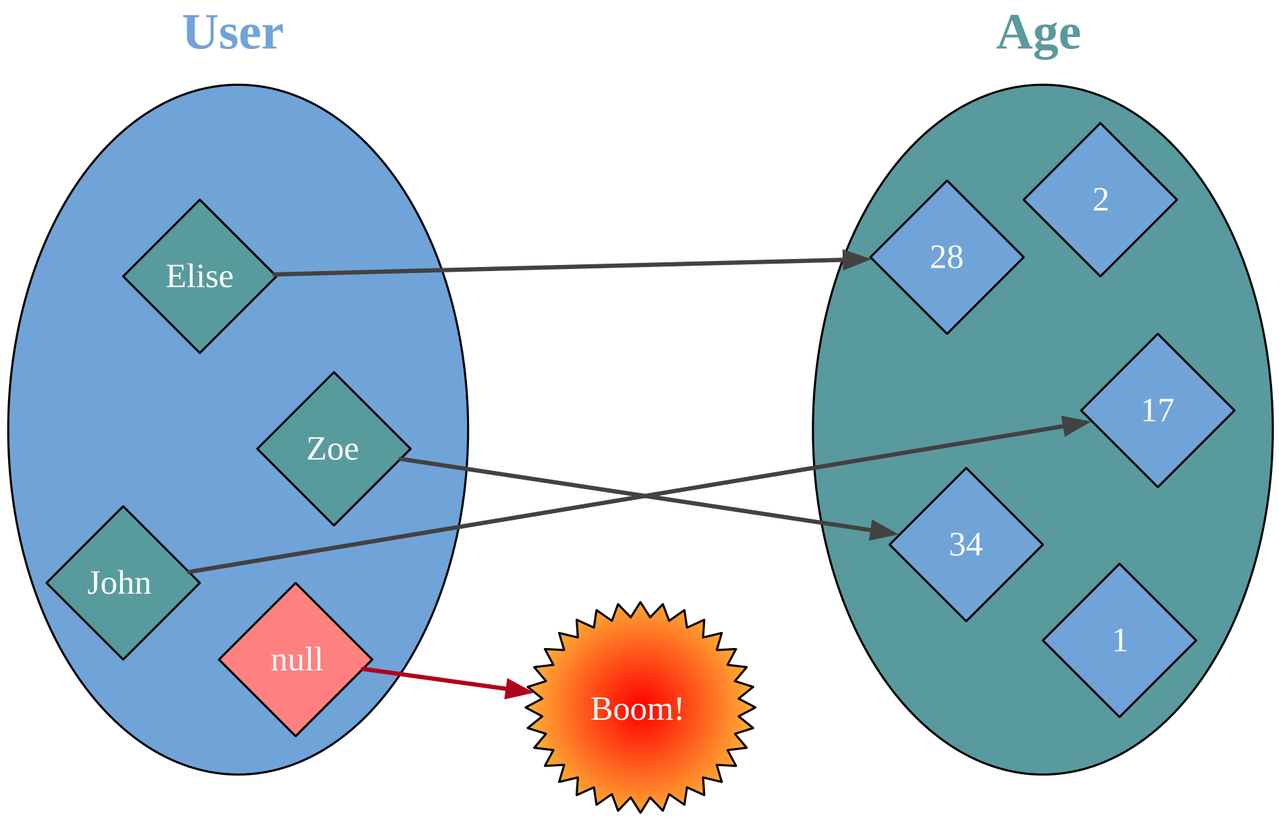
- total (not partial)
- no exception
- deterministic (not nondeterministic)
- no mutation
- no side effect

Functional subset = pure function + ...

Null



Null



Null

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

def getAge(user: User): Int = {
  if(user == null) -1
  else user.age
}
```

Null

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

def getAge(user: User): Int = {
  if(user == null) -1
  else user.age
}
```

`null` causes NullPointerException

We cannot remove `null` from the language (maybe in Scala 3)

So we ignore null: don't return it, don't handle it

Reflection

```
trait OrderApi {
  def insertOrder(order: Order): Future[Unit]
  def getOrder(orderId: OrderId): Future[Order]
}
```

```
class DbOrderApi(db: DB) extends OrderApi { ... }

class OrderApiWithAuth(api: OrderApi, auth: AuthService) extends OrderApi { ... }
```

```
def getAll(api: OrderApi)(orderIds: List[OrderId]): Future[List[Order]] =
  api match {
    case x: DbOrderApi       => ... □
    case x: OrderApiWithAuth => ... □
    case _                   => ... □
  }
```

Reflection

```
trait OrderApi {
  def insertOrder(order: Order): Future[Unit]
  def getOrder(orderId: OrderId): Future[Order]
}
```

```
class DbOrderApi(db: DB) extends OrderApi { ... }

class OrderApiWithAuth(api: OrderApi, auth: AuthService) extends OrderApi { ... }
```

```
def getAll(api: OrderApi)(orderIds: List[OrderId]): Future[List[Order]] = {
  if     (api.`isInstanceOf[DbOrderApi]`) ...
  else if(api.`isInstanceOf[OrderApiWithAuth]`) ...
  else ...
}
```

An OPEN trait/class is equivalent to a record of functions

```
trait OrderApi {
  def insertOrder(order: Order): Future[Unit]
  def getOrder(orderId: OrderId): Future[Order]
}

case class OrderApi(
  insertOrder: Order   => Future[Unit],
  getOrder   : OrderId => Future[Order]
)
```

An OrderApi is any pair of functions `(insertOrder, getOrder)`

A SEALED trait/class is equivalent to an enumeration

```
sealed trait ConfigValue

object ConfigValue {
  case class AnInt(value: Int)      extends ConfigValue
  case class AString(value: String) extends ConfigValue
  case object Empty                 extends ConfigValue
}
```

A ConfigValue is either an Int, a String or Empty

Any, AnyRef, AnyVal are all OPEN trait

```
def getTag(any: Any): Int = any match {
  case x: Int         =>  1 
  case x: String      =>  2 
  case x: ConfigValue =>  3 
  case _              => -1 
}
```

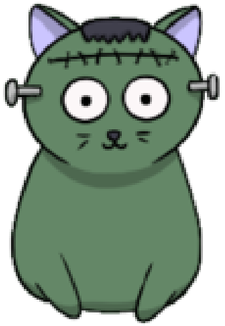
Functional subset (aka Scalazzi subset)

- total
- no exception
- deterministic
- no mutation
- no side effect
- no null
- no reflection

@impurepics

FUNCTIONS

TOTAL
(NOT PARTIAL)



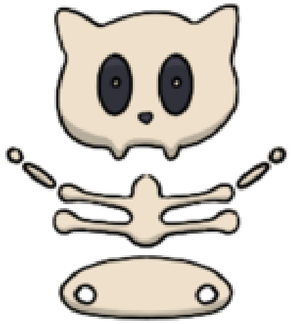
DETERMINISTIC
(NO RANDOMNESS)

PURE
(NO SIDE EFFECT)

NO MUTATION

NO NULL

NO REFLECTION



NO EXCEPTION

Exercise 4

`exercises.function.FunctionExercises.scala`

Why should we use the functional subset?

1. Refactoring: remove unused code

```
def hello_1(foo: Foo, bar: Bar) = {
  `val x = f(foo)`
  val y = g(bar)
  `h(y)`
  y
}
```

```
def hello_2(foo: Foo, bar: Bar) =
  g(bar)
```

1. Refactoring: remove unused code

```
def hello_1(foo: Foo, bar: Bar) = {
  `val x = f(foo)`
  val y = g(bar)
  `h(y)`
  y
}
```

```
def hello_2(foo: Foo, bar: Bar) =
  g(bar)
```

Counter example

```
def f(foo: Foo): Unit = upsertToDb(foo)

def h(id: Int): Unit = globalVar += 1
```

1. Refactoring: reorder variables

```
def hello_1(foo: Foo, bar: Bar) = {
  `val x = f(foo)`
  val y = g(bar)
  h(x, y)
}
```

```
def hello_2(foo: Foo, bar: Bar): Int = {
  val y = g(bar)
  `val x = f(foo)`
  h(x, y)
}
```

1. Refactoring: reorder variables

```
def hello_1(foo: Foo, bar: Bar) = {
  `val x = f(foo)`
  val y = g(bar)
  h(x, y)
}
```

```
def hello_2(foo: Foo, bar: Bar): Int = {
  val y = g(bar)
  `val x = f(foo)`
  h(x, y)
}
```

Counter example

```
def f(foo: Foo): Unit = print("foo")
def g(bar: Bar): Unit = print("bar")

hello_1(foo, bar) // print foobar
hello_2(foo, bar) // print barfoo
```

1. Refactoring: extract - inline

```
def hello_extract(foo: Foo, bar: Bar) = {
  val x = f(foo)
  val y = g(bar)
  h(x, y)
}
```

```
def hello_inline(foo: Foo, bar: Bar) = {
  h(f(foo), g(bar))
}
```

1. Refactoring: extract - inline

```
def hello_extract(foo: Foo, bar: Bar) = {
  val x = f(foo)
  val y = g(bar)
  h(x, y)
}
```

```
def hello_inline(foo: Foo, bar: Bar) = {
  h(f(foo), g(bar))
}
```

Counter example

```
def f(foo: Foo): Boolean = false

def g(bar: Bar): Boolean = throw new Exception("Boom!")

def h(b1: Boolean, b2:` =>` Boolean): Boolean = b1 && b2

hello_extract(foo, bar) // throw Exception
hello_inline (foo, bar) // false
```

1. Refactoring: extract - inline

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def doSomethingExpensive(x: Int): Future[Int] =
  Future { ??? }

for {
  x <- doSomethingExpensive(5)
  y <- doSomethingExpensive(8) // sequential, 2nd Future starts when 1st Future is complete
} yield x + y
```

```
val fx = doSomethingExpensive(5)
val fy = doSomethingExpensive(8) // both Futures start in parallel

for {
  x <- fx
  y <- fy
} yield x + y
```

1. Refactoring: de-duplicate

```
def hello_duplicate(foo: Foo) = {
  val x = f(foo)
  val y = f(foo)
  h(x, y)
}
```

```
def hello_simplified(foo: Foo) = {
  val x = f(foo)
  h(x, x)
}
```

1. Refactoring: de-duplicate

```
def hello_duplicate(foo: Foo) = {
  val x = f(foo)
  val y = f(foo)
  h(x, y)
}
```

```
def hello_simplified(foo: Foo) = {
  val x = f(foo)
  h(x, x)
}
```

Counter example

```
def f(foo: Foo): Unit = print("foo")

hello_duplicate(foo)  // print foofoo
hello_simplified(foo) // print foo
```

Pure function

means

fearless refactoring

2. Local reasoning

```
def hello(foo: Foo, bar: Bar): Int = {

  ??? // only depends on foo, bar
}
```

2. Local reasoning

```
class HelloWorld(fizz: Fizz) {

  val const = 12.3

  def hello(foo: Foo, bar: Bar): Int = {

    ??? // only depends on foo, bar, const and fizz
  }

}
```

2. Local reasoning

```
class HelloWorld(fizz: Fizz) {

  var secret = null // 

  def hello(foo: Foo, bar: Bar): Int = {
    FarAwayObject.mutableMap += "foo" -> foo // 
    publishMessage(Hello(foo, bar)) // 
    ???
  }

}

object FarAwayObject {
  val mutableMap = ??? // 
}
```

3. Easier to test

```
test("submit") {
  val item  = Item("xxx", 2, 12.34)
  val now   = Instant.now()
  val order = Order("123", "Checkout", List(item), submittedAt = None)

  submit(order, `now`) shouldEqual order.`copy`(status = "Submitted", submittedAt = Some(`now`))
}
```

Dependency injection is given by local reasoning

No mutation, no randomness, no side effect

4. Better documentation

```
def getAge(user: User): `Int` = ???

def getOrElse[A](fa: Option[A])(orElse: `=> A`): A = ???

def parseJson(x: String): `Either[ParsingError`, Json] = ???

def mapOption[`A`, `B`](fa: Option[`A`])(f: `A` => `B`): Option[`B`] = ???

def none: Option[`Nothing`] = ???
```

5. Potential compiler optimisations

Fusion

```
val largeList = List.range(0, 10000)

largeList.map(f).map(g) == largeList.map(f andThen g)
```

Caching

```
def memoize[A, B](f: A => B): A => B = ???

val cacheFunc = memoize(f)
```

What's the catch?

With pure function, you cannot **DO** anything

Resources and further study

- Explain List Folds to Yourself
- Constraints Liberate, Liberties Constrain

Module 2: Side Effect

Parametric types

```
case class Point(x: Int, y: Int)
```

```
Point(3, 4)
// res99: Point = Point(3, 4)
```

```
case class Pair[A](first: A, second: A)
```

```
Pair(3, 4)
// res100: Pair[Int] = Pair(3, 4)

Pair("John", "Doe")
// res101: Pair[String] = Pair("John", "Doe")
```

Parametric functions

```
def swap[A](pair: Pair[A]): Pair[A] =
  Pair(pair.second, pair.first)
```

```
swap(Pair(1, 5))
// res102: Pair[Int] = Pair(5, 1)
swap(Pair("John", "Doe"))
// res103: Pair[String] = Pair("Doe", "John")
```

Pattern match

```
def swap[A](pair: Pair[A]): Pair[A] =
  pair match {
    case x: Pair[Int]    => Pair(x.first + 1, x.second - 1)
    case x: Pair[String] => Pair(x.first    , x.second.reverse)
    case other           => Pair(pair.second, pair.first)
  }
```

Pattern match

```
def swap[A](pair: Pair[A]): Pair[A] =
  pair match {
    case x: Pair[Int]    => Pair(x.first + 1, x.second - 1)
    case x: Pair[String] => Pair(x.first    , x.second.reverse)
    case other           => Pair(pair.second, pair.first)
  }
```

```
swap(Pair(1, 5))
// res105: Pair[Int] = Pair(2, 4)
```

```
swap(Pair("John", "Doe"))
// java.lang.ClassCastException: class java.lang.String cannot be cast to class java.lang.Integer (java.lang.String
//     at scala.runtime.BoxesRunTime.unboxToInt(BoxesRunTime.java:99)
//     at repl.Session$App104.swap(1-Function.html:1270)
//     at repl.Session$App104$$anonfun$188.apply(1-Function.html:1286)
//     at repl.Session$App104$$anonfun$188.apply(1-Function.html:1286)
```

Type erasure

```
def swap(pair: Pair[Any]): Pair[Any] =
  pair match {
    case x     => Pair(x.first + 1, x.second - 1)
    case x     => Pair(x.first    , x.second.reverse)
    case other => Pair(pair.second, pair.first)
  }
```

Type erasure is a good thing ™

```
def swap[A](pair: Pair[A]): Pair[A] = ???
```

For all type A, `swap` takes a `Pair` of `A` and returns a `Pair` of `A`.

1. Type parameters must be defined before we use them

```
case class Pair[`A`](first: A, second: A)

def swap[`A`](pair: Pair[A]): Pair[A] =
  Pair(pair.second, pair.first)
```

```
def swap(pair: Pair[A]): Pair[A] =
  Pair(pair.second, pair.first)

On line 2: error: not found: type A
swap(pair: Pair[A]): Pair[A] =
               ^
```

2. Type parameters should not be introspected

```
def showPair[A](pair: Pair[A]): String =
  pair match {
    case p: Pair[Int]    => s"(${p.first}, ${p.second})"
    case p: Pair[Double] => s"(${truncate2(p.first)} , ${truncate2(p.second)})"
    case _               => "N/A"
  }
```

```
showPair(Pair(10, 99))
showPair(Pair(1.12345, 0.000001))
showPair(Pair("John", "Doe"))
```

2. Type parameters should not be introspected

```
def show[A](value: A): String =
  value match {
    case x: Int       => x.toString
    case x: Double    => truncate2(x)
    case _            => "N/A"
  }
```

```
show(1)
show(2.3)
show("Foo")
```

A type parameter is a form of encapsulation

Types vs Type constructors

```
Int
String
Direction
```

```
List
Map
Ordering
```

```
val counter: Int = 5

val message: String = "Welcome!"
```

```
val elems: List = List(1, 2, 3)
// error: type List takes type parameters
// val elems: List = List(1, 2, 3)
//                   ^^^^
```

Def vs Val functions details

Def function (Method)

In Scala

```
def replicate(n: Int, text: String): String
```

In Java

```
String replicate(int n, String text)
```

Conciseness

```
def plus(x: Int, y: Int): Int =
  x + y
```

```
val plus: (Int, Int) => Int =
  (x: Int, y: Int) => x + y
```

Conciseness

```scala
def plus(x: Int, y: Int): Int =
  x + y
```

```scala
val plus: (Int, Int) => Int =
  (x: Int, y: Int) => x + y
```

```scala
def plus(x: Int, y: Int) =
  x + y
```

```scala
val plus =
  (x: Int, y: Int) => x + y
```

Conciseness

```
def plus(x: Int, y: Int): Int =
  x + y
```

```
val plus: (Int, Int) => Int =
  (x: Int, y: Int) => x + y
```

```
def plus(x: Int, y: Int) =
  x + y
```

```
val plus =
  (x: Int, y: Int) => x + y
```

```
val plus: (Int, Int) => Int =
  (x, y) => x + y
```

Conciseness

```scala
def plus(x: Int, y: Int): Int =
  x + y
```

```scala
val plus: (Int, Int) => Int =
  (x: Int, y: Int) => x + y
```

```scala
def plus(x: Int, y: Int) =
  x + y
```

```scala
val plus =
  (x: Int, y: Int) => x + y
```

```scala
val plus: (Int, Int) => Int =
  _ + _
```

Definition order

```
val repeat = replicate

val replicate: (Int, String) => String =
  (n, text) => List.fill(n)(text).mkString
```

Definition order

```
val repeat = replicate

val replicate: (Int, String) => String =
  (n, text) => List.fill(n)(text).mkString
```

```
// warning: Reference to uninitialized value replicate
// val repeat = replicate
```

```
repeat(3, "Hello ")
// java.lang.NullPointerException
//     at repl.Session$App121$$anonfun$235.apply(1-Function.html:1476)
//     at repl.Session$App121$$anonfun$235.apply(1-Function.html:1476)
```

Definition order

```
def repeat(n: Int, text: String): String =
  replicate(n, text)

def replicate(n: Int, text: String): String =
  List.fill(n)(text).mkString
```

```
repeat(3, "Hello ")
// res123: String = "Hello Hello Hello "
```

Definition order

```
lazy val repeat = replicate

lazy val replicate: (Int, String) => String =
  (n, text) => List.fill(n)(text).mkString
```

```
repeat(3, "Hello ")
// res125: String = "Hello Hello Hello "
```

Unimplemented functions

```
def repeat(n: Int, text: String): String =
  ???
```

Unimplemented functions

```
def repeat(n: Int, text: String): String =
  ???
```

```
def ??? : Nothing = throw new NotImplementedError
```

Unimplemented functions

```
def repeat(n: Int, text: String): String =
  ???
```

```
val replicate: (Int, String) => String =
  ???
// scala.NotImplementedError: an implementation is missing
//     at scala.Predef$.$qmark$qmark$qmark(Predef.scala:347)
//     at repl.Session$App127$$anonfun$238.apply$mcV$sp(1-Function.html:1544)
//     at repl.Session$App127$$anonfun$238.apply(1-Function.html:1542)
//     at repl.Session$App127$$anonfun$238.apply(1-Function.html:1542)
```

Unimplemented functions

```
def repeat(n: Int, text: String): String =
  ???
```

```
lazy val replicate: (Int, String) => String =
  ???
```