FOUNDATION

Function

First class function

```scala
val counter  : Int        = 3
val message  : String     = "Hello World!"
val increment: Int => Int = (x: Int) => x + 1
```

First class function

```scala
val counter  : Int        = 3
val message  : String     = "Hello World!"
val increment: Int => Int = (x: Int) => x + 1
```

```scala
def increment(x: Int): Int = x + 1
```

First class function

```scala
val counter  : Int        = 3
val message  : String     = "Hello World!"
val increment: Int => Int = (x: Int) => x + 1
```

```scala
val increment: Function1[Int, Int] = (x: Int) => x + 1
val plus: Function2[Int, Int, Int] = (x: Int, y: Int) => x + y
```

First class function

```
val inc   : Int => Int = (x: Int) => x + 1
val dec   : Int => Int = (x: Int) => x - 1
val double: Int => Int = (x: Int) => x * 2

val list = List(inc, dec, double)

val dictionary = Map(
  "increment"  -> inc,
  "decrement"  -> dec,
  "double"     -> double,
)
```

```
scala> dictionary("double")(10)
res0: Int = 20
```

Higher order function

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

```scala
scala> upperCase("Hello")
res1: String = HELLO
```

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

```scala
scala> lowerCase("Hello")
res2: String = hello
```

Higher order function

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

Higher order function

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

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

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

def  password(s: String): String = map(s, c => '*')
```

```
scala> password("123456")
res3: String = ******
```

Partial function application

```scala
def formatDouble(scale: Int)(value: Double): String =
  BigDecimal(value)
    .setScale(scale, BigDecimal.RoundingMode.HALF_DOWN)
    .toDouble
    .toString
```

```
scala> formatDouble(2)(1.123456789)
res4: String = 1.12

scala> formatDouble(5)(1.123456789)
res5: String = 1.12346
```

Partial function application

```scala
def formatDouble(scale: Int)(value: Double): String =
  BigDecimal(value)
    .setScale(scale, BigDecimal.RoundingMode.HALF_DOWN)
    .toDouble
    .toString

val format2D = formatDouble(2)
val format5D = formatDouble(5)
```

```
scala> format2D(1.123456789)
res6: String = 1.12

scala> format5D(1.123456789)
res7: String = 1.12346
```

Partial function application

```
val formatDouble: Int => (Double => String) =
  (scale: Int) => {
    (value: Double) => {
      BigDecimal(value)
        .setScale(scale, BigDecimal.RoundingMode.HALF_DOWN)
        .toDouble
        .toString
     }
  }
```

Partial function application

```scala
val formatDouble: Int => (Double => String) =
  (scale: Int) => {
    (value: Double) => {
      BigDecimal(value)
        .setScale(scale, BigDecimal.RoundingMode.HALF_DOWN)
        .toDouble
        .toString
     }
  }
```

```scala
val format2D: Double => String = formatDouble(2)
val format5D: Double => String = formatDouble(5)
```

Exercise 1

`exercises.function.FunctionExercises.scala`

Parametric types

```
Int
String
Direction
```

```
List[Int]
Map[Int, String]
```

Parametric types

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

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

```
scala> Point(3, 4)
res8: Point = Point(3,4)

scala> Pair(3, 4)
res9: Pair[Int] = Pair(3,4)

scala> Pair("John", "Doe")
res10: Pair[String] = Pair(John,Doe)
```

Parametric functions

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

```
scala> swap(Pair(1, 5))
res11: Pair[Int] = Pair(5,1)

scala> swap(Pair("John", "Doe"))
res12: Pair[String] = Pair(Doe,John)
```

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}, ${p.second})"
    case p: Pair[Double] => s"(${format2D(p.first)} , ${format2D(p.second)})"
    case _               => "N/A"
  }
```

```
scala> showPair(Pair(10, 99))
res13: String = (10, 99)

scala> showPair(Pair(1.12345, 0.000001))
res14: String = (1.12345, 1.0E-6)

scala> showPair(Pair("John", "Doe"))
res15: String = (John, Doe)
```

2. Type parameters should not be introspected

```scala
def show[A](value: A): String =
  value match {
    case x: Int       => x.toString
    case x: Double    => format2D(x)
    case _            => "N/A"
  }
```

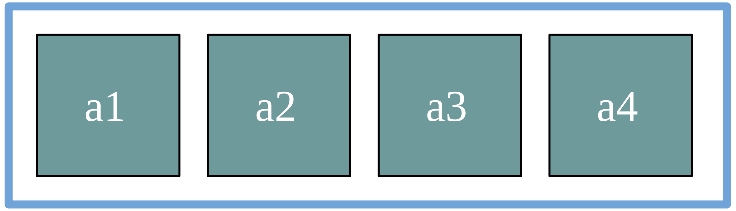
```
scala> show(1)
res16: String = 1

scala> show(2.3)
res17: String = 2.3

scala> show("Foo")
res18: String = N/A
```

A type parameter is a form of encapsulation

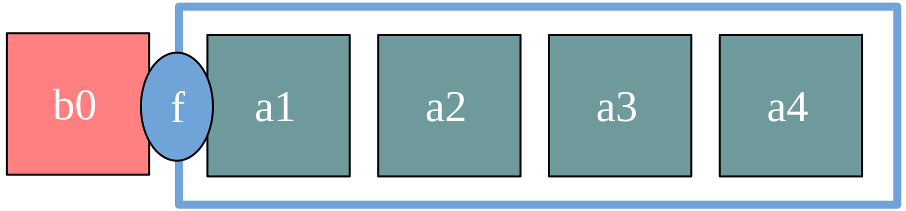
Exercises 2 and 3a-b

`exercises.function.FunctionExercises.scala`

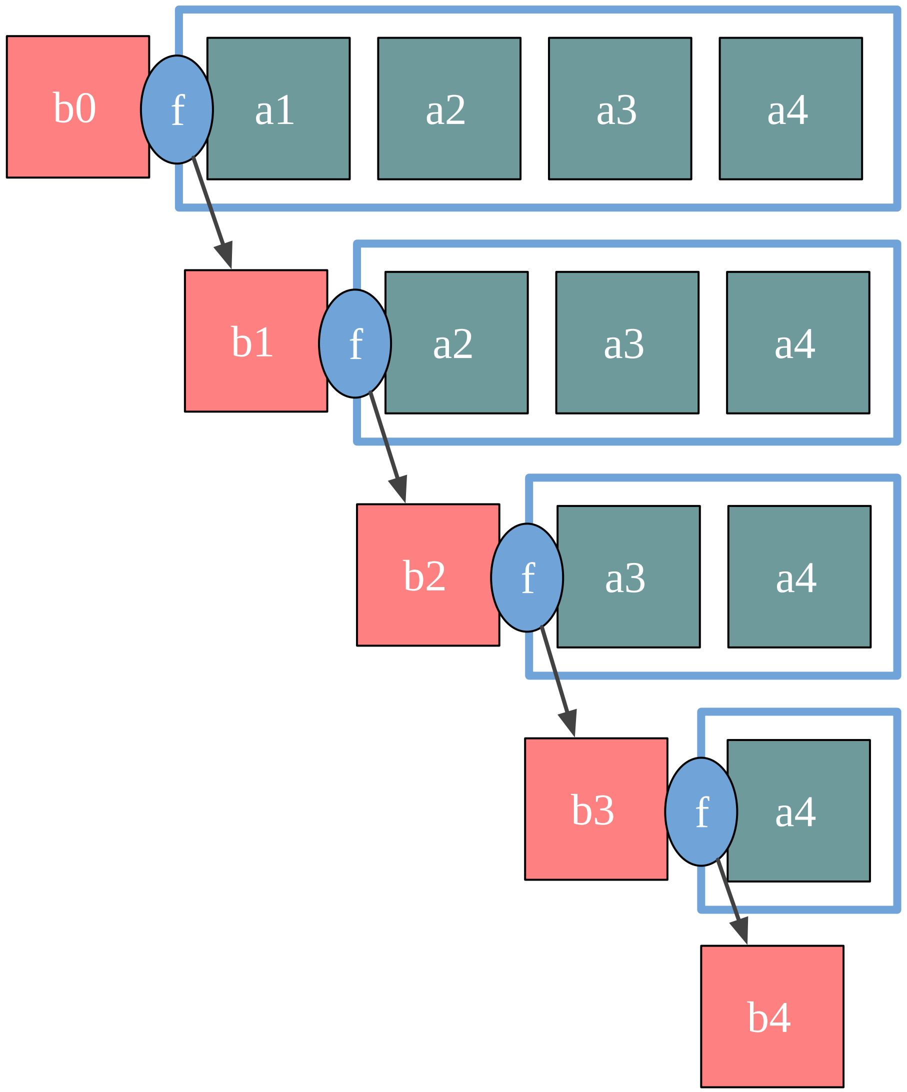
Folding



FoldLeft

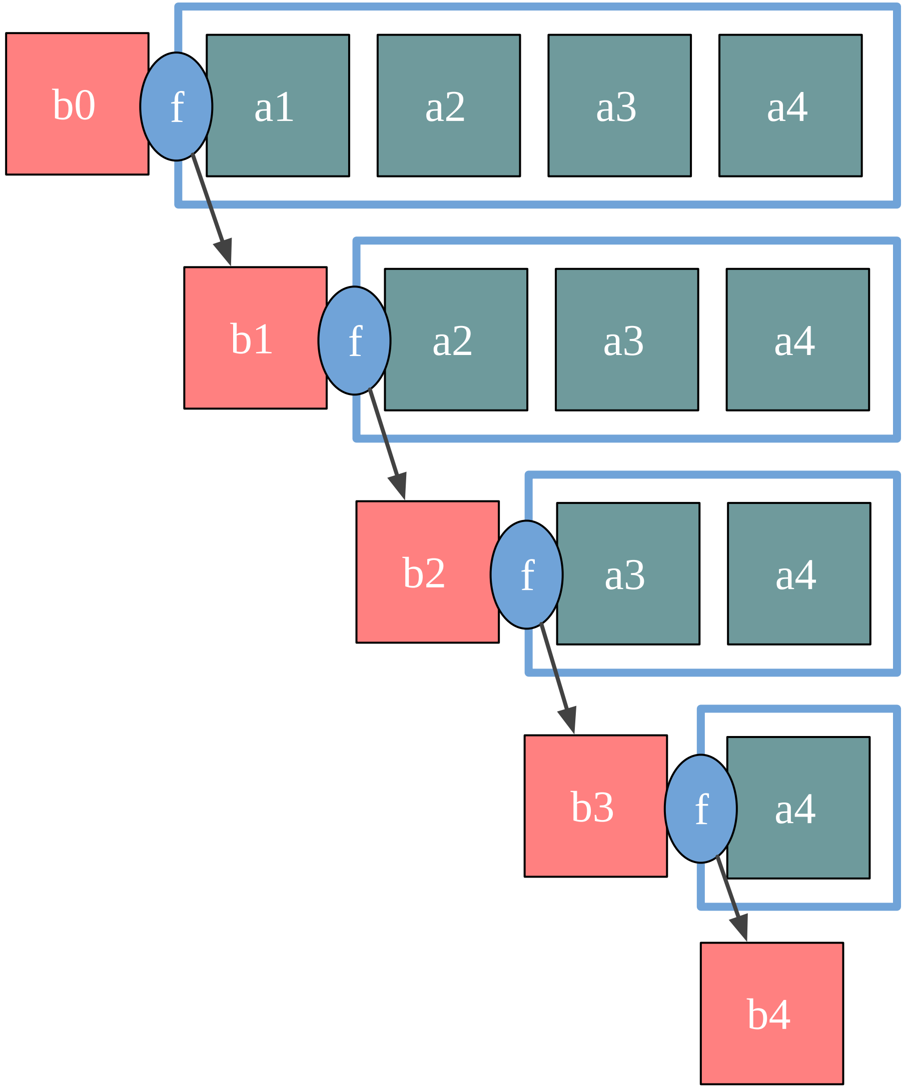


FoldLeft

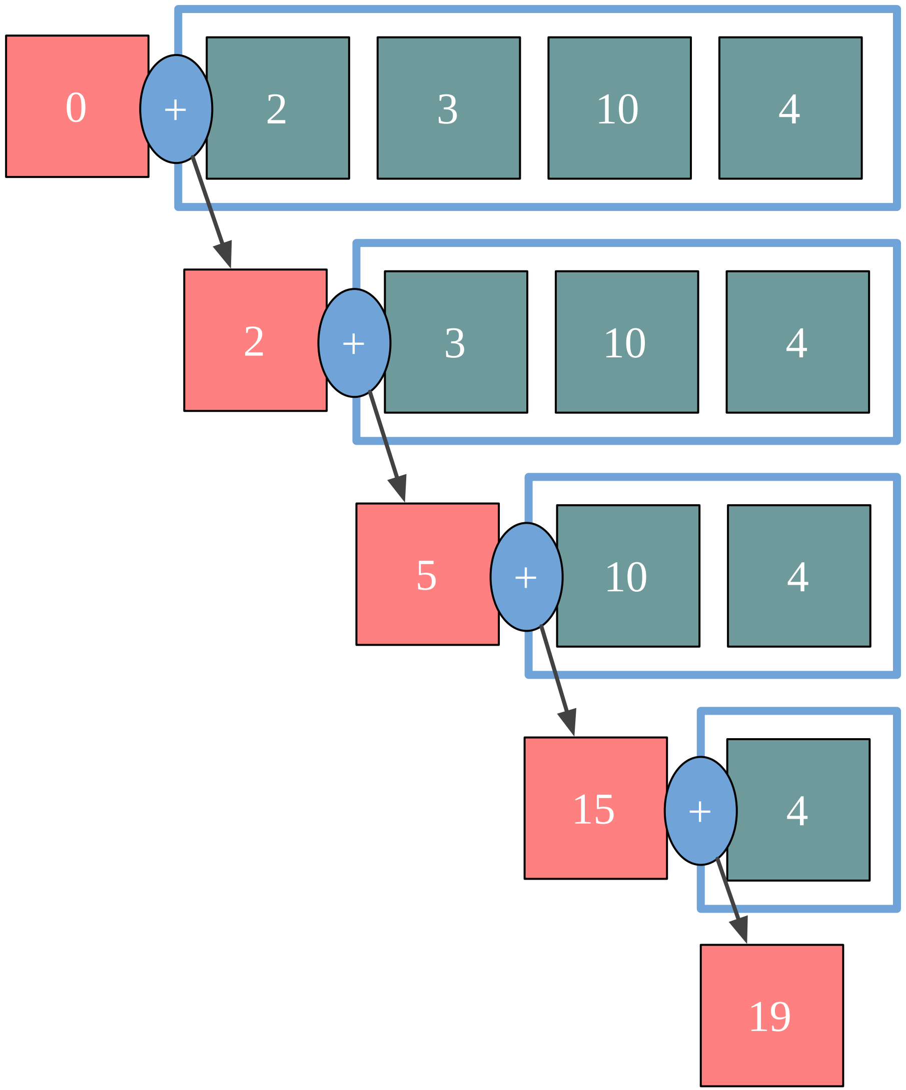


FoldLeft

```scala
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



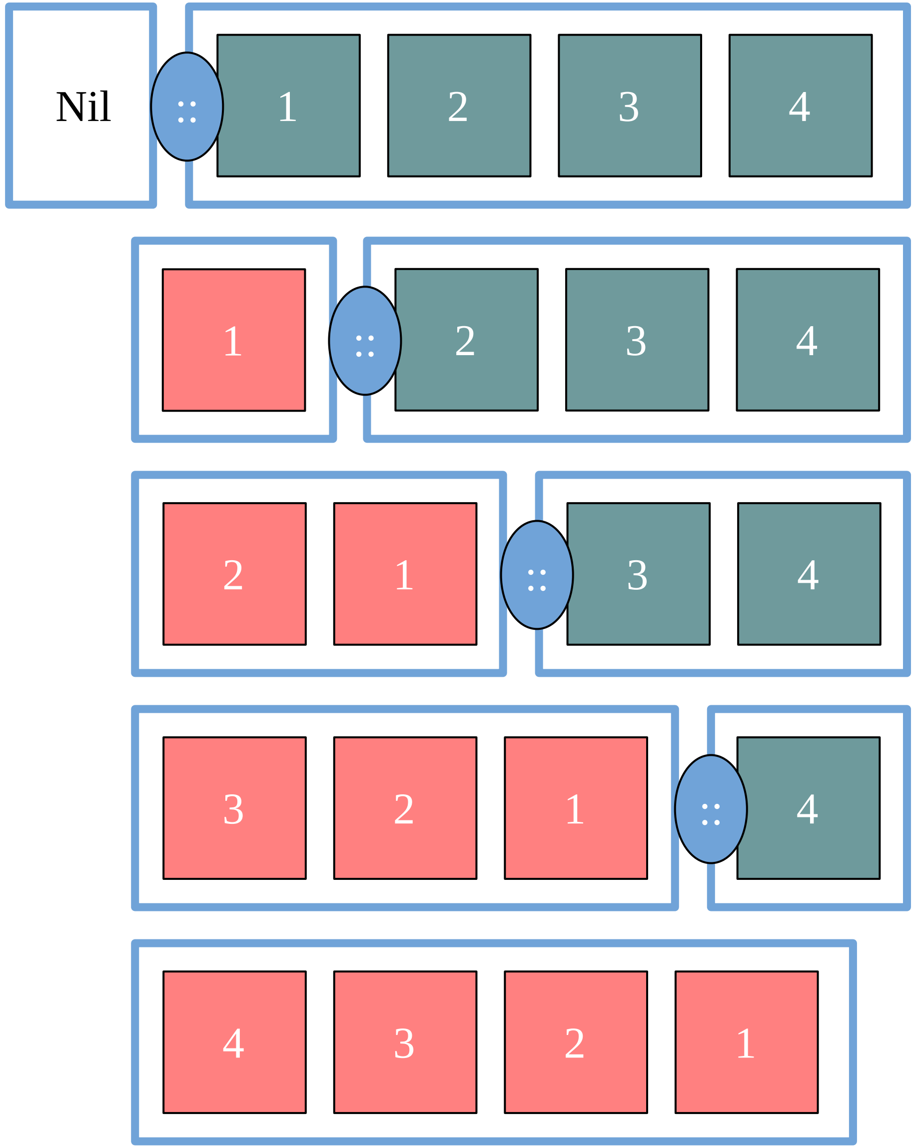

```scala
def sum(xs: List[Int]): Int =
  foldLeft(xs, 0)(_ + _)
```

```
scala> sum(List(2,3,10,4))
res19: Int = 19
```

FoldLeft

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

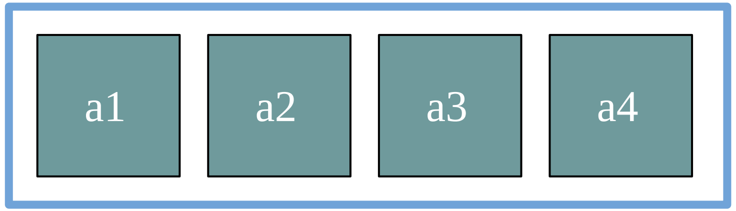
```scala
scala> reverse(List(1,2,3,4))
res20: List[Int] = List(4, 3, 2, 1)
```



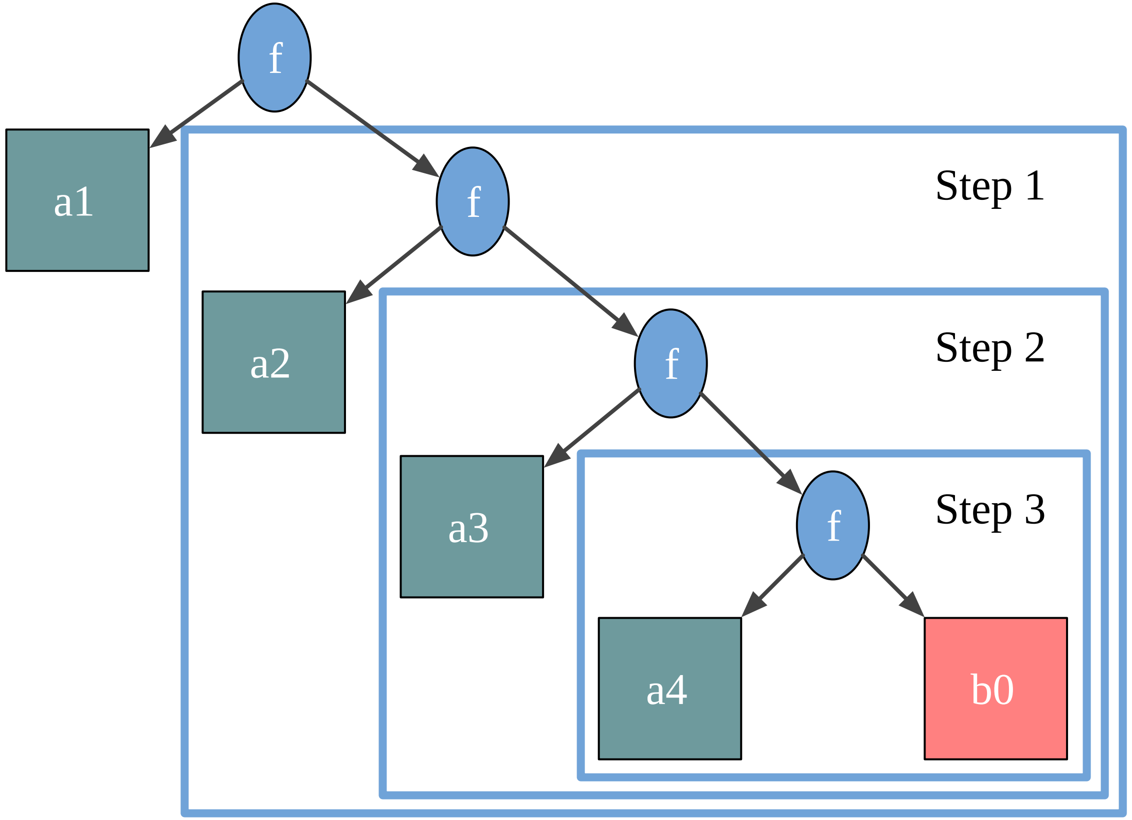
Exercise 3c-f

`exercises.function.FunctionExercises.scala`

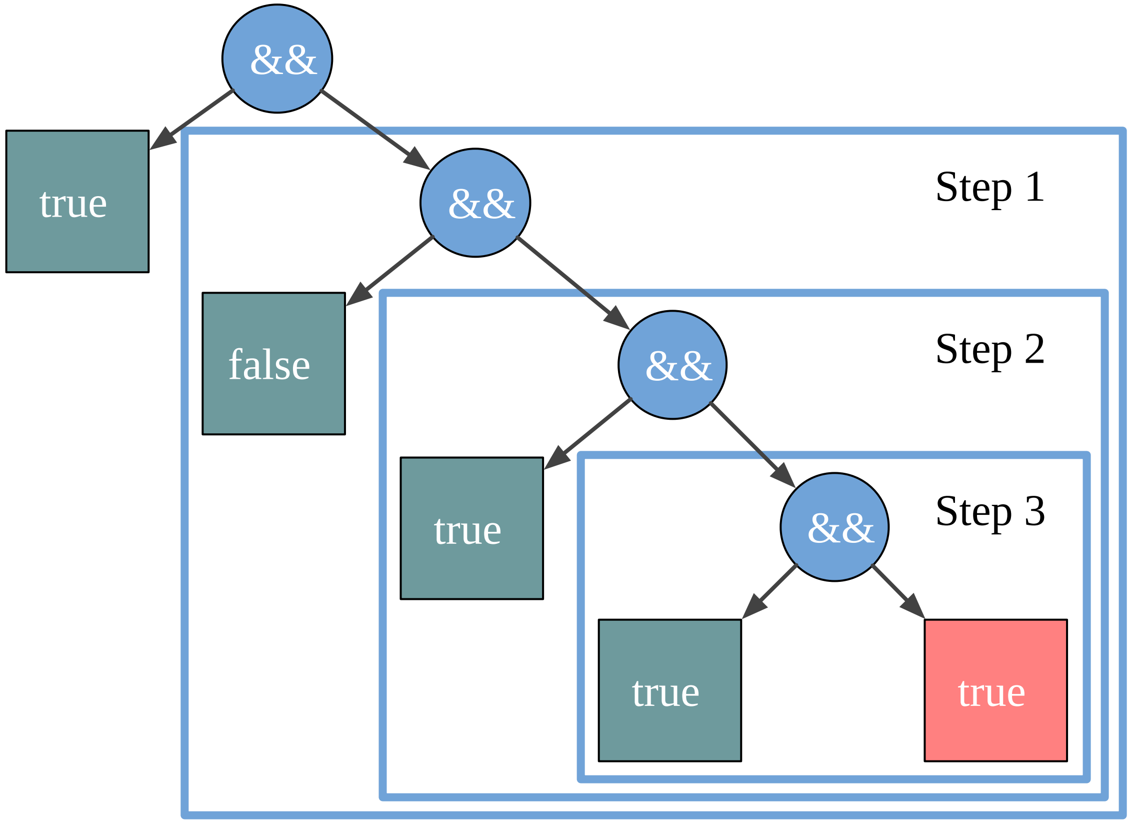
Folding



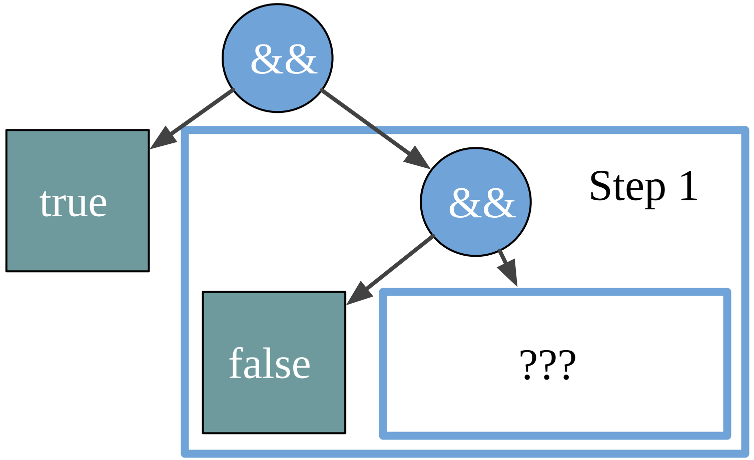
FoldRight



FoldRight

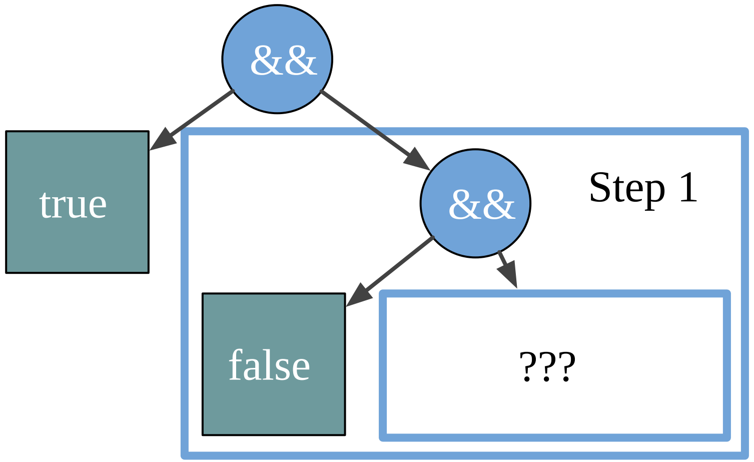


FoldRight is lazy

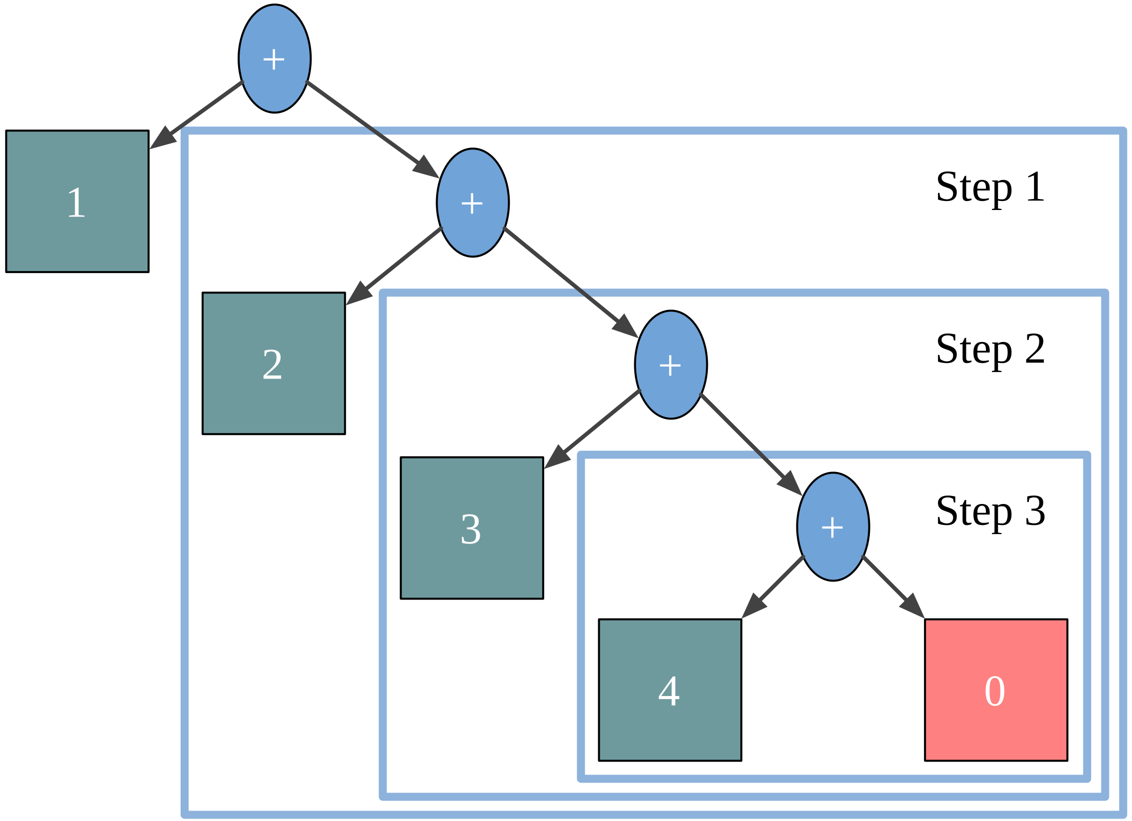


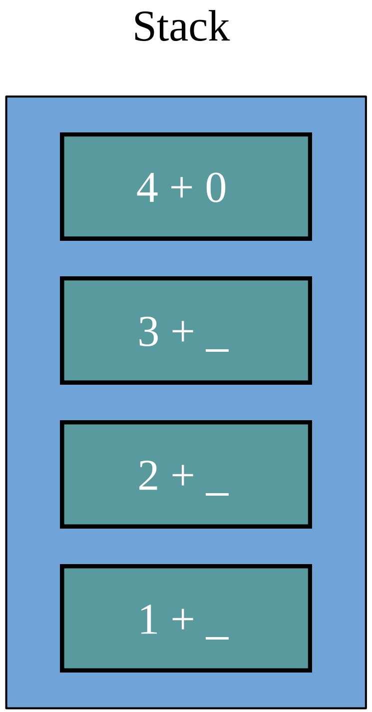
FoldRight is lazy

```
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

```scala
sealed trait List[A]

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

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

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

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

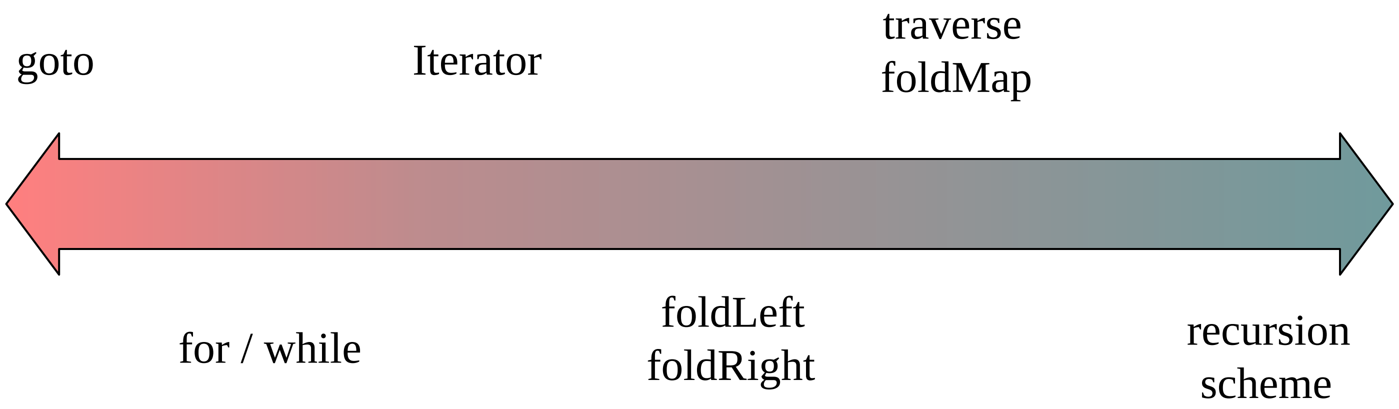
```
def foldRight[A, B](list: List[A], b: B)(f: (A, => B) => B): B

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

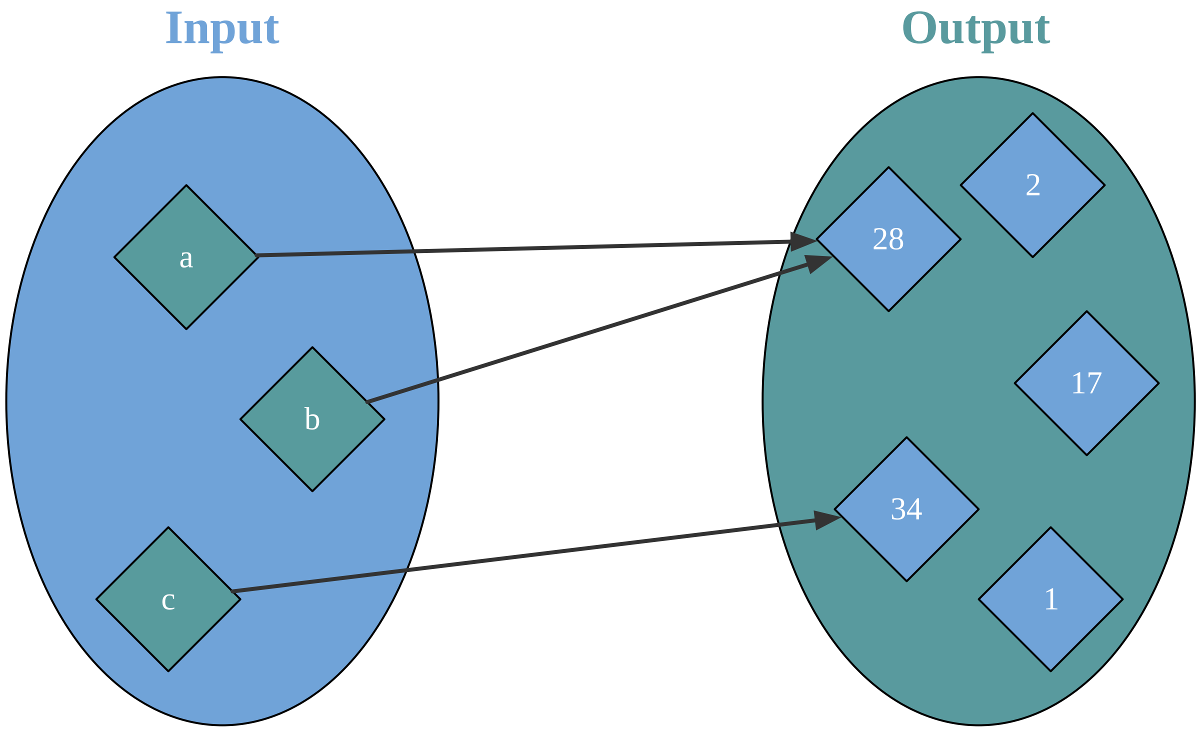
`exercises.function.FunctionExercises.scala`

Different level of abstractions



Pure function

Pure functions are mappings between two sets

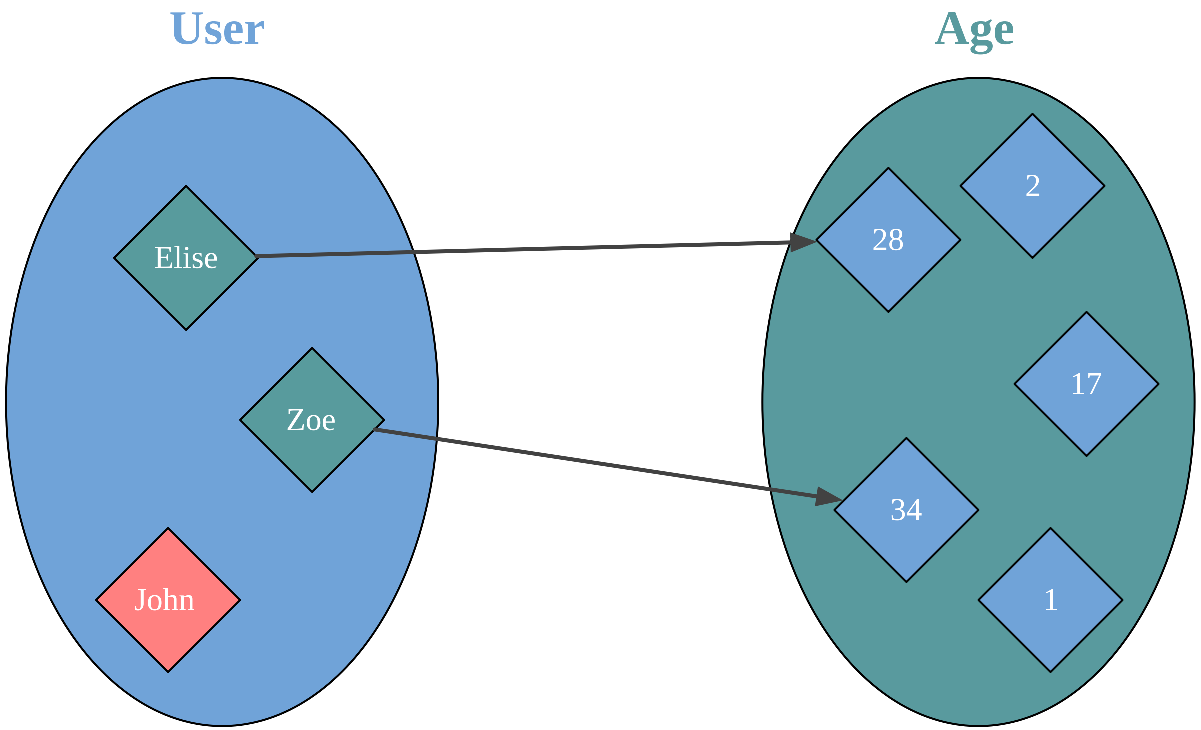


Programming function

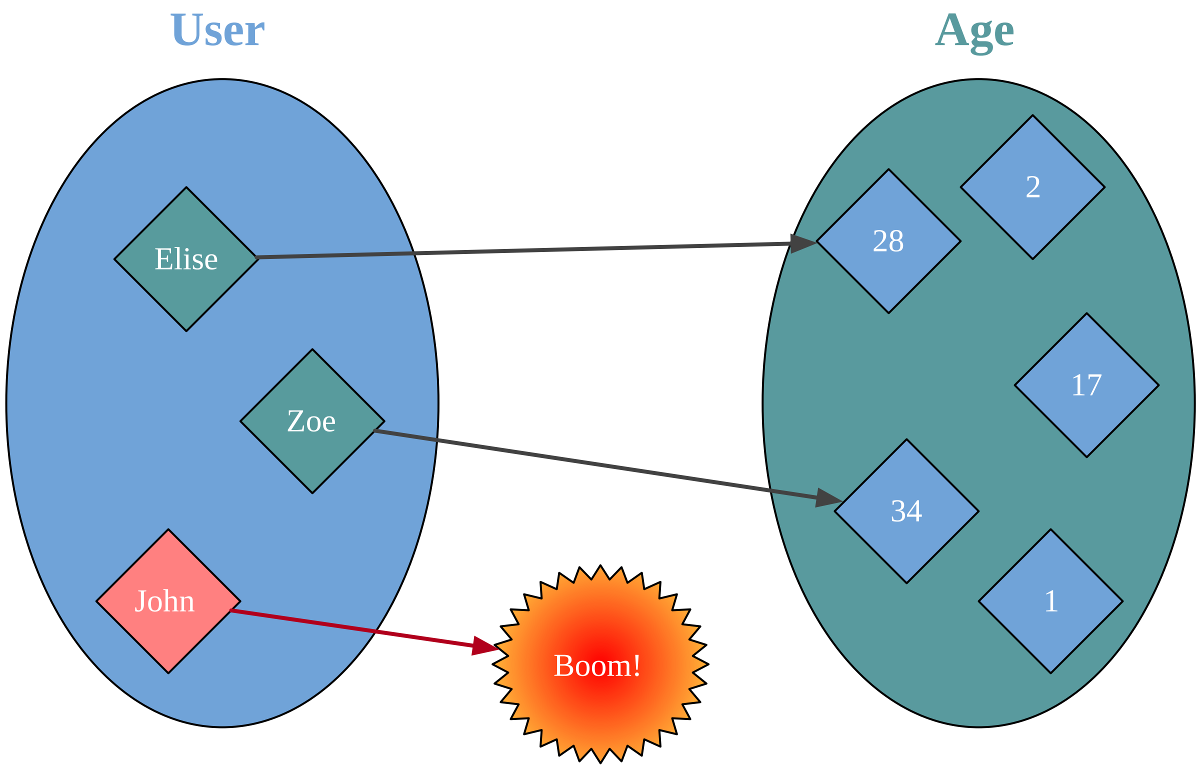
!=

Pure function

Partial function



Partial function



Partial function

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

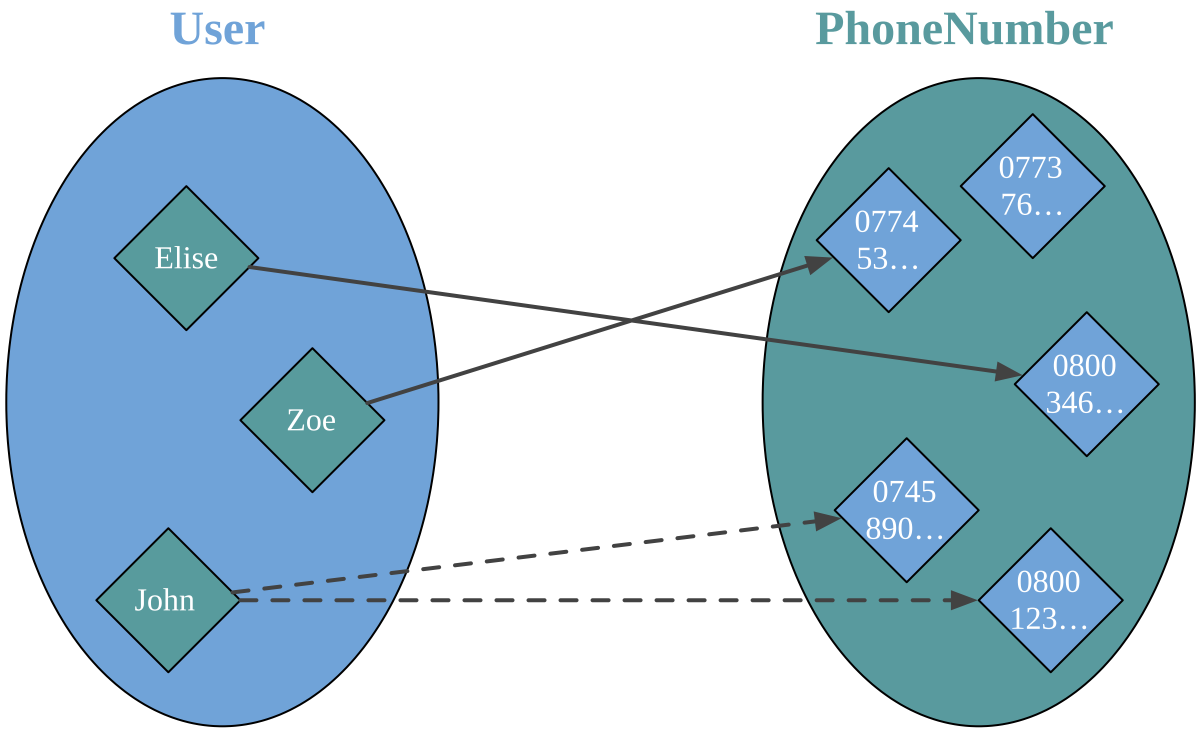
```
scala> head(Nil)
scala.MatchError: List() (of class scala.collection.immutable.Nil$)
  at .head(<console>:3)
  ... 42 elided
```

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")
  }
```

```
scala> submit(Order("Delivered", Nil))
java.lang.Exception: Invalid Command
  at .submit(<console>:7)
  ... 42 elided
```

Nondeterministic



Nondeterministic

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

```
scala> UUID.randomUUID()
res2: java.util.UUID = c4c7ae3e-ad84-4fb1-b635-73601474d8f0

scala> UUID.randomUUID()
res3: java.util.UUID = c8d698a4-2455-4be2-bf73-1bc5f324c914
```

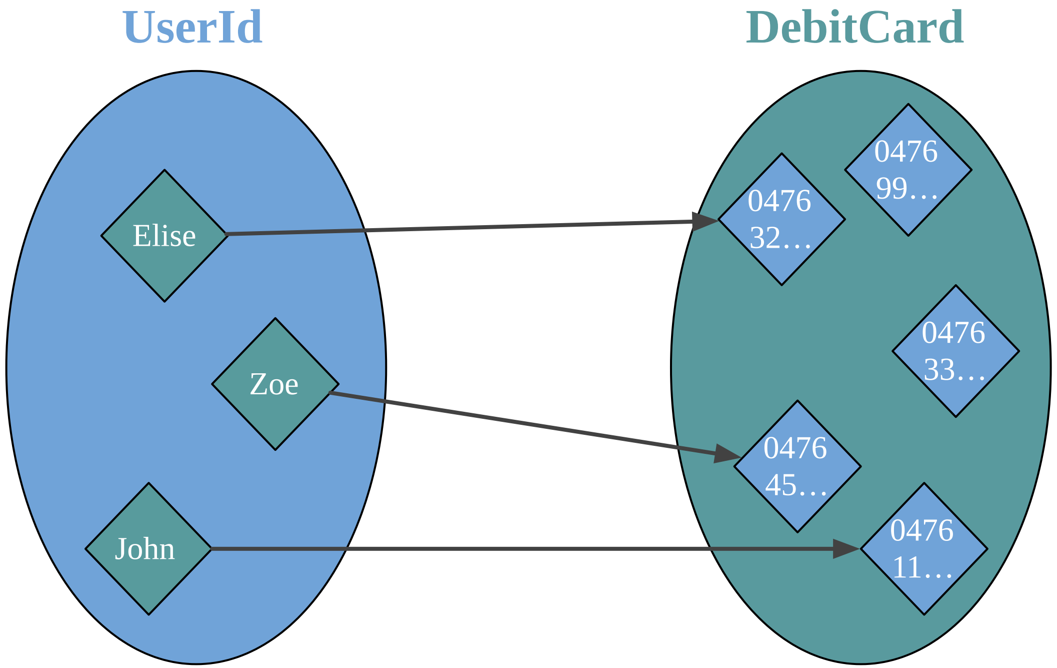
```
scala> Instant.now()
res4: java.time.Instant = 2019-12-19T10:28:26.004004Z

scala> Instant.now()
res5: java.time.Instant = 2019-12-19T10:28:26.075626Z
```

Mutation

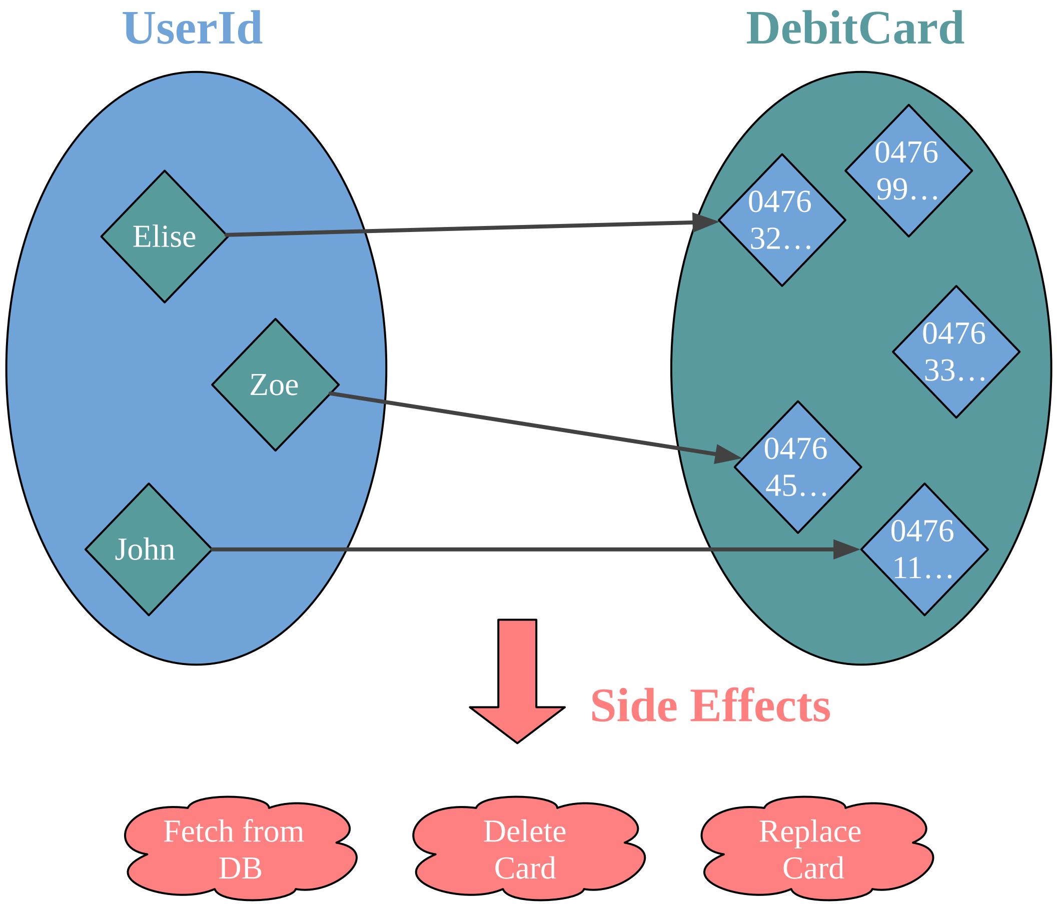
```scala
class User(initialAge: Int) {
  var age: Int = initialAge

  def getAge: Int = age

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

val john = new User(24)
```

```
scala> john.getAge
res6: Int = 24

scala> john.setAge(32)

scala> john.getAge
res8: Int = 32
```

Side effect


UserId
DebitCard
Elise
Zoe
John
0476 99…
0476 32…
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

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

```scala
scala> val x = println("Hello")
Hello
x: Unit = ()
```

Side effect

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

```
scala> val x = println("Hello")
Hello
x: Unit = ()
```

```
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

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

```
scala> val x = println("Hello")
Hello
x: Unit = ()
```

```
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>
```

```scala
var x: Int = 0

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

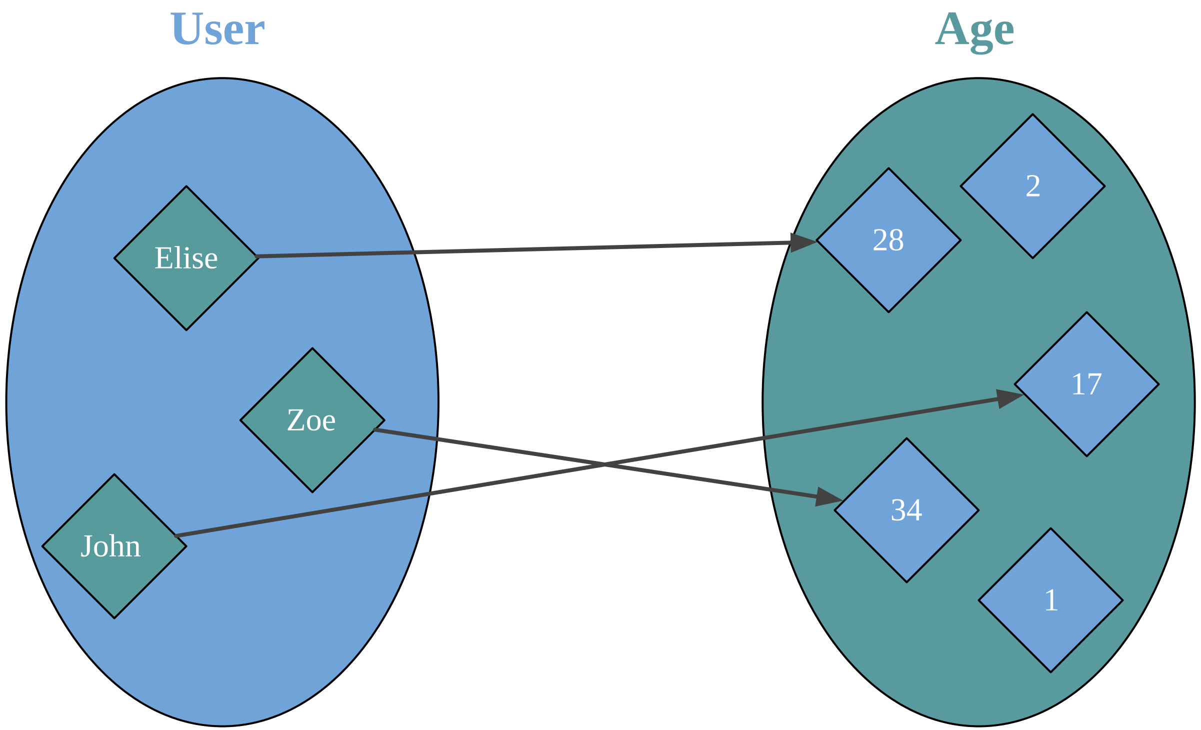
A function without side effects only returns a value

Pure function

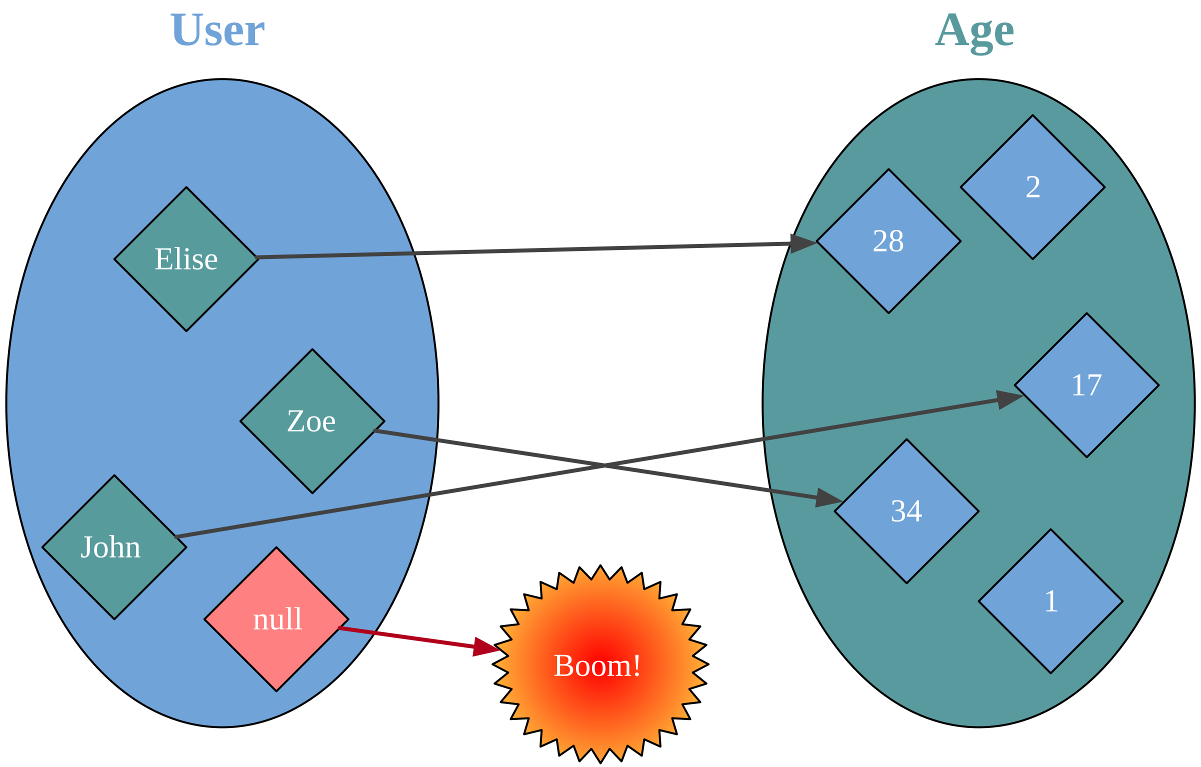
- total (not partial)
- no exception
- deterministic (not nondeterministic)
- no mutation
- no side effect

Functional subset = pure function + …

Null



Null



Null

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

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

Null

```scala
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

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

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

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

```scala
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

```scala
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

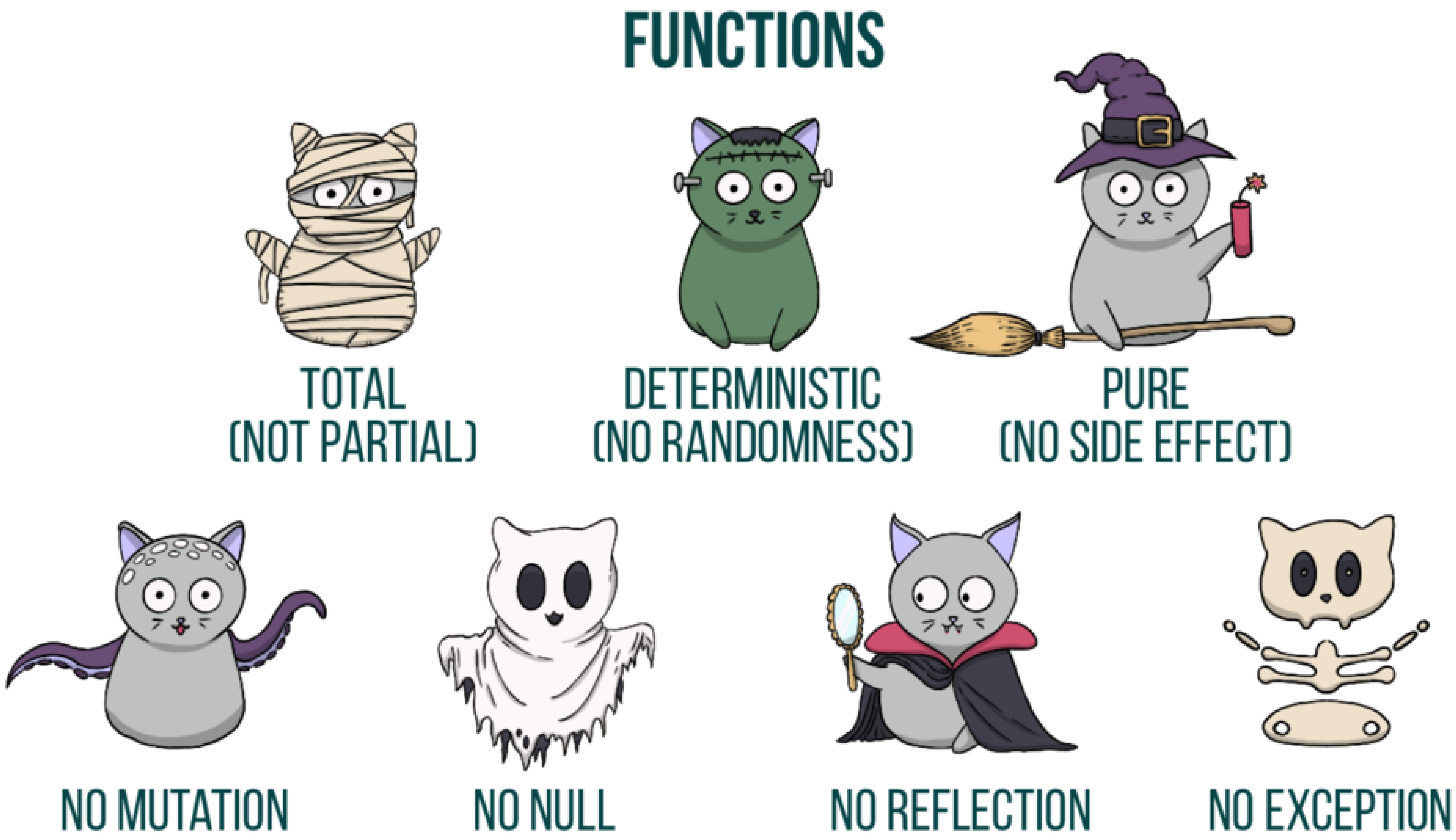
Any, AnyRef, AnyVal are all OPEN trait

```
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



Exercise 4

`exercises.function.FunctionExercises.scala`

Why should we use the functional subset?

1. Refactoring: remove unused code

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

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

1. Refactoring: remove unused code

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

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

Counter example

```scala
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

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

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

Counter example

```scala
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

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

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

Counter example

```scala
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

```scala
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
```

```scala
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

```scala
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

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

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

Caching

```scala
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