FOUNDATION

Plan

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

Functions

Val function (Lambda)

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

Def function (Method)

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

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)

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

Val function (Lambda or anonymous function)

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

```scala
3

"Hello World!"

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

Val functions are ordinary objects

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

```scala
val counter = 3
val message = "Hello World!"
val john    = User("John Doe", 27)
```

Val functions are ordinary objects

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

```scala
val counter = 3

val message = "Hello World!"

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

```scala
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

```scala
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>)
```

```scala
functions(0)(10)
// res10: Int = 11

functions(2)(10)
// res11: Int = 20
```

Val function desugared

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

Val function desugared

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

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

Val function desugared

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

```scala
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

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

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

```scala
replicate.apply(3, "Hello ")
// res17: String = "Hello Hello Hello "

replicate(3, "Hello ")
// res18: String = "Hello Hello Hello "
```

Def function (Method)

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

Def function (Method)

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

```scala
List(replicate)
// error: missing argument list for method replicate in class App9
// Unapplied methods are only converted to functions when a function type is expected.
// You can make this conversion explicit by writing replicate _ or replicate(_,_) instead of replicate.
```

Def function (Method)

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

```
List(replicate _)
// res22: List[(Int, String) => String] = List(<function2>)
```

Def function (Method)

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

```scala
List(replicate _)
// res22: List[(Int, String) => String] = List(<function2>)
```

```scala
val replicateVal = replicate _
// replicateVal: (Int, String) => String = <function2>
```

Def function (Method)

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

```scala
List(replicate): List[(Int, String) => String]
```

```scala
val replicateVal: (Int, String) => String = replicate
```

Function arguments

```scala
import java.time.LocalDate

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

```scala
createDate(2020, 1, 5)
// res25: LocalDate = 2020-01-05
```

Function arguments

```scala
import java.time.LocalDate

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

```scala
createDate(2020, 1, 5)
// res25: LocalDate = 2020-01-05
```

```scala
createDate(dayOfMonth = 5, month = 1, year = 2020)
// res26: LocalDate = 2020-01-05
```

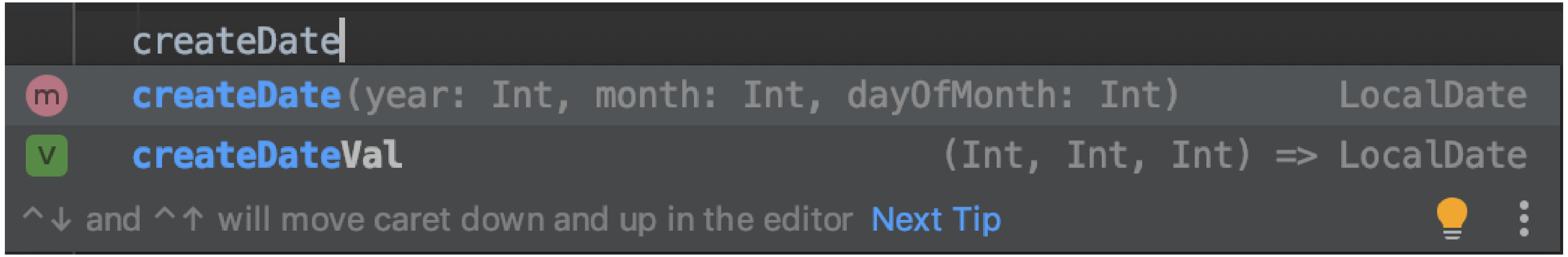
Function arguments

```scala
import java.time.LocalDate

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

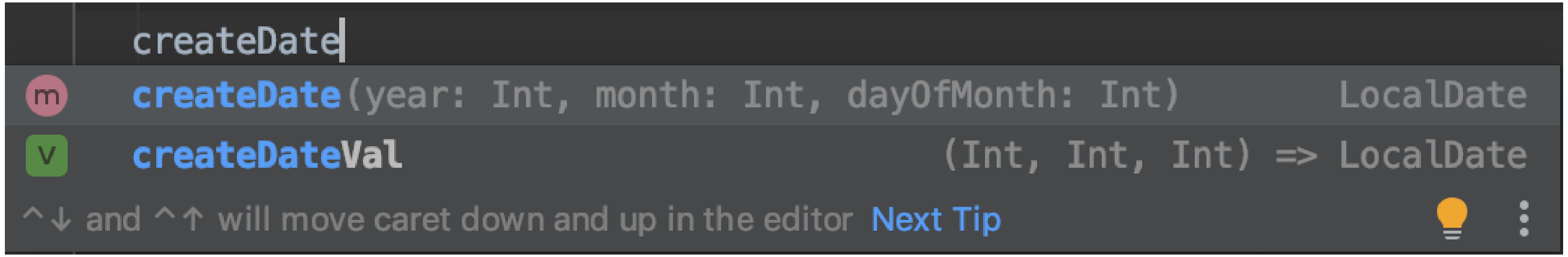
```scala
val createDateVal: (Int, Int, Int) => LocalDate =
  (year, month, dayOfMonth) => ...
```

```scala
createDate(2020, 1, 5)
// res27: LocalDate = 2020-01-05

createDateVal(2020, 1, 5)
// res28: LocalDate = 2020-01-05
```

IDE


createDate
createDate(year: Int, month: Int, dayOfMonth: Int) LocalDate
createDateVal (Int, Int, Int) => LocalDate
^↓ and ^↑ will move caret down and up in the editor Next Tip

IDE



Javadoc

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

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

Summary

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

Def vs Val functions details

Def function (Method)

In Scala

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

In Java

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

Conciseness

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

```scala
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

```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 =
  (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

```scala
val repeat = replicate

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

Definition order

```scala
val repeat = replicate

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

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

```scala
repeat(3, "Hello ")
// java.lang.NullPointerException
//     at repl.Session$App43$$anonfun$117.apply(1-Function.html:444)
//     at repl.Session$App43$$anonfun$117.apply(1-Function.html:444)
```

Definition order

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

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

```scala
repeat(3, "Hello ")
// res45: String = "Hello Hello Hello "
```

Definition order

```scala
lazy val repeat = replicate

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

```scala
repeat(3, "Hello ")
// res47: String = "Hello Hello Hello "
```

Unimplemented functions

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

Unimplemented functions

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

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

Unimplemented functions

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

```scala
val replicate: (Int, String) => String =
  ???
// scala.NotImplementedError: an implementation is missing
//     at scala.Predef$.$qmark$qmark$qmark(Predef.scala:347)
//     at repl.Session$App49$$anonfun$120.apply$mcV$sp(1-Function.html:512)
//     at repl.Session$App49$$anonfun$120.apply(1-Function.html:510)
//     at repl.Session$App49$$anonfun$120.apply(1-Function.html:510)
```

Unimplemented functions

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

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

Functions as input

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

Functions as input

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

```scala
val text = "Hello World!"
```

```scala
filter(text, (c: Char) => c.isUpper)
// res51: String = "HW"


filter(text, (c: Char) => c.isLetter)
// res52: String = "HelloWorld"
```

Reduce code duplication

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

```scala
upperCase("Hello")
// res53: String = "HELLO"
```

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

```scala
lowerCase("Hello")
// res54: 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)
```

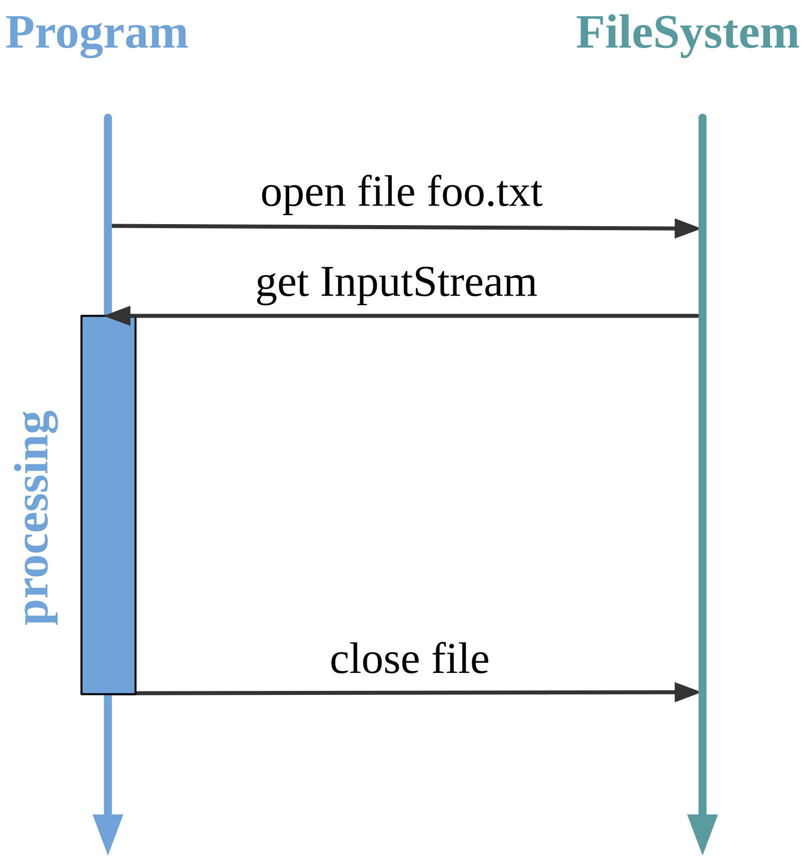
Capture pattern

```scala
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)
}
```

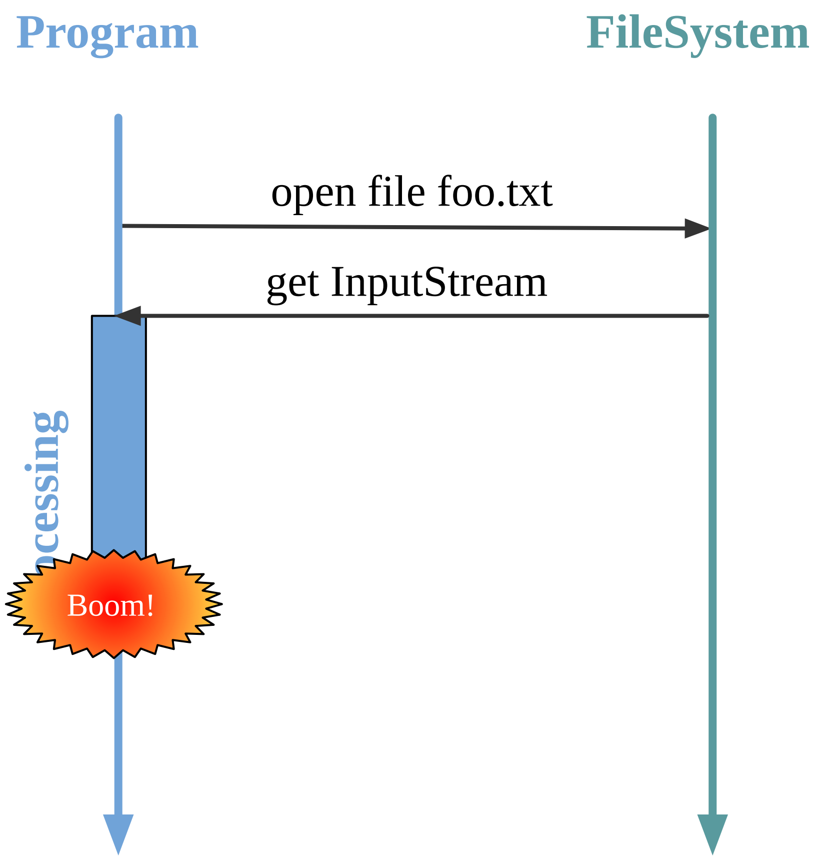
```scala
test("map does not change the size") {
  forAll((
    text  : String,
    update: Char => Char
  ) =>
    map(text, update).length == text.length
  )
}
```

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

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

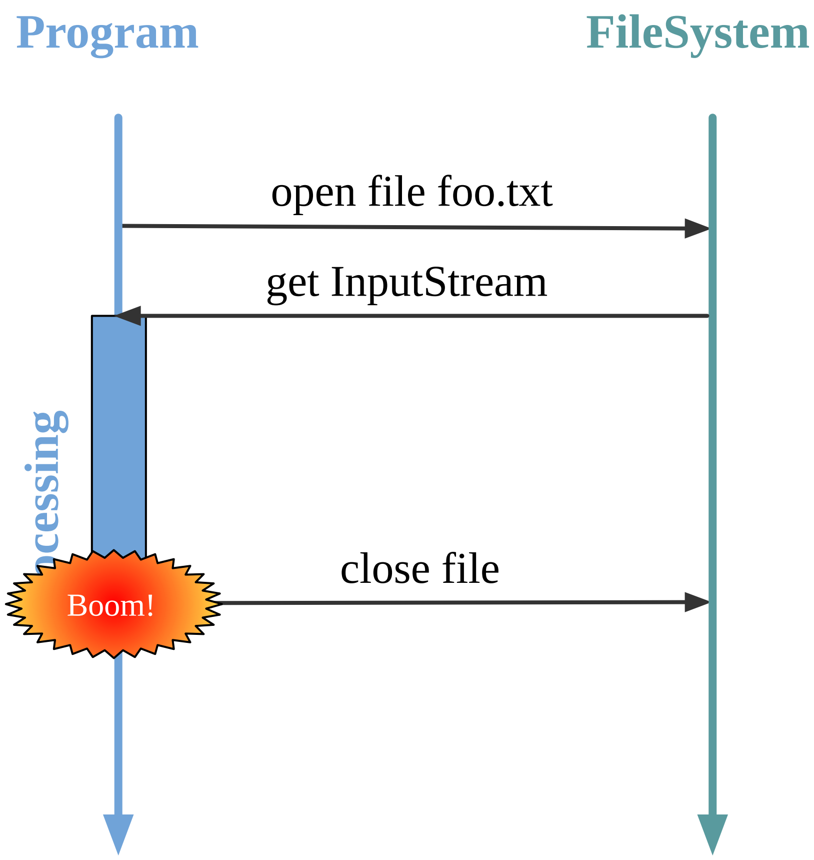
File processing



File processing



File processing



Write tricky code once

```scala
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

```scala
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()
  }
}
```

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

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

Write tricky code once

```scala
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()
  }
}
```

```scala
usingFile("50-word-count.txt", countLines)
// res57: Int = 2
```

```scala
usingFile("50-word-count.txt", countWords)
// res58: Int = 50
```

Exercise 1: Functions as input

`exercises.function.FunctionExercises.scala`

Functions as output

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

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

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

Functions as output

```scala
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)
```

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

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

Functions as output

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

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

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

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

Functions as output

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

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

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

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

Currying

```scala
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

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

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

```scala
truncate2D(0.123456789)
// res68: String = "0.12"

truncate5D(0.123456789)
// res69: String = "0.12345"
```

Syntax

Uncurried

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

Curried

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

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

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

Conversion (Currying)

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

Conversion (Currying)

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

```
truncate _
// res71: (Int, Double) => String = <function2>
```

Conversion (Currying)

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

```scala
truncate _
// res71: (Int, Double) => String = <function2>
```

```scala
(truncate _).curried
// res72: Int => Double => String = scala.Function2$$Lambda$5037/0x0000000101a2e840@e8263a2
```

Exercise 2: Functions as output

`exercises.function.FunctionExercises.scala`

Types

```
Int
String
User
```

```
val counter: Int = 5

val message: String = "Welcome!"

val alice: User = User("Alice", 23)
```

Parametric types

```
List
Map
JsonEncoder
```

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

Types

```
Int
String
User
```

```
val counter: Int = 5

val message: String = "Welcome!"

val alice: User = User("Alice", 23)
```

Parametric types

```
List
Map
JsonEncoder
```

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

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

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

Functions

```scala
def map(s: String, f: Char => Char): String = ...
```

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

Functions

```scala
def map(s: String, f: Char => Char): String = ...
```

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

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

Parametric functions

```scala
def map[To](list: List[Int], f: Int => To): List[To] = ...
```

```scala
map(List(1,2,3,4), (x: Int) => x + 1)
// res74: List[Int] = List(2, 3, 4, 5)

map(List(1,2,3,4), (x: Int) => x / 2.0)
// res75: List[Double] = List(0.5, 1.0, 1.5, 2.0)
```

Parametric functions

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

```scala
map(List(1,2,3,4), (x: Int) => x / 2.0)
// res77: List[Double] = List(0.5, 1.0, 1.5, 2.0)

map(List("Hello", "World"), (x: String) => x.toCharArray)
// res78: List[Array[Char]] = List(
//   Array('H', 'e', 'l', 'l', 'o'),
//   Array('W', 'o', 'r', 'l', 'd')
// )
```

Parametric functions

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

```scala
map(List(1,2,3,4), (x: Int) => x / 2.0)
// res77: List[Double] = List(0.5, 1.0, 1.5, 2.0)

map(List("Hello", "World"), (x: String) => x.toCharArray)
// res78: List[Array[Char]] = List(
//   Array('H', 'e', 'l', 'l', 'o'),
//   Array('W', 'o', 'r', 'l', 'd')
// )
```

#1 Benefit: code reuse

Interpretation

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

Interpretation

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

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

```scala
map[Int, String](List(1,2,3), (x: Int) => x.toString)
// res79: List[String] = List("1", "2", "3")
```

How can we implement `map`?

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

How can we implement `map`?

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

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

How can we implement `map`?

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

Does it compile?

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

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

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

Does it compile?

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

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

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

#2 Benefit: require less tests

Exercises 3: Parametric functions

`exercises.function.FunctionExercises.scala`

Type inference

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

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

Type inference

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

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

```scala
Pair(0, 2).zipWith(Pair(3, 3), (x, y) => x + y)
// error: missing parameter type
// Pair(0, 2).zipWith(Pair(3, 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(3, 3))((x, y) => x + y)
// res84: Pair[Int] = Pair(3, 5)
```

Type inference

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

```scala
Pair(0, 2).zipWith(Pair(3, 3))((x, y) => x + y)
// res84: Pair[Int] = Pair(3, 5)
```

```scala
Pair(0, 2).zipWith(Pair(3, 3))(_ + _)
// res85: Pair[Int] = Pair(3, 5)
```

Scala API design

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

Scala API design

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

Two apparently useless functions

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

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

```scala
identity(5)
// res86: Int = 5

identity("Hello")
// res87: String = "Hello"
```

```scala
constant(5)("Hello")
// res88: Int = 5

constant("Hello")(5)
// res89: String = "Hello"
```

Two apparently useless functions

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

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

Two apparently useless functions

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

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

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

```scala
toggle()
// res90: Boolean = true

toggle()
// res91: Boolean = false
```

Two apparently useless functions

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

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

How would you implement?

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

Two apparently useless functions

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

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

How would you implement?

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

Two apparently useless functions

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

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

How would you implement?

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

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

Two apparently useless functions

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

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

How would you implement?

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

Two apparently useless functions

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

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

How would you implement?

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

Consistent API

```scala
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)

}
```

What is the type of `identityVal`?

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

```
val identityVal = identity _
```

What is the type of `identityVal`?

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

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

```scala
identityVal(4)
// error: type mismatch;
//  found   : Int(4)
//  required: Nothing
```

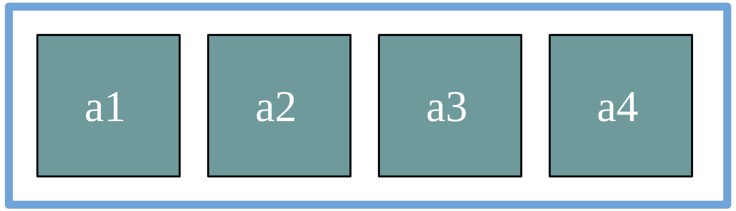
What is the type of `identityVal`?

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

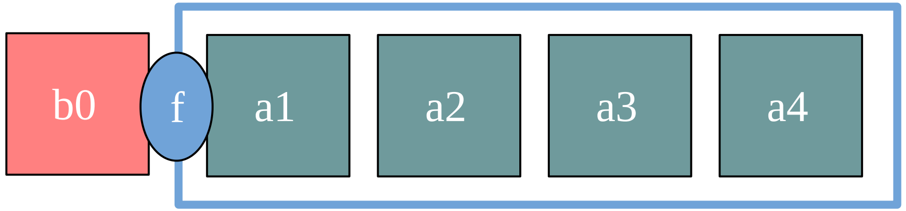
```scala
val identityVal: Int => Int = identity[Int] _
```

```scala
identityVal(4)
// res97: Int = 4
```

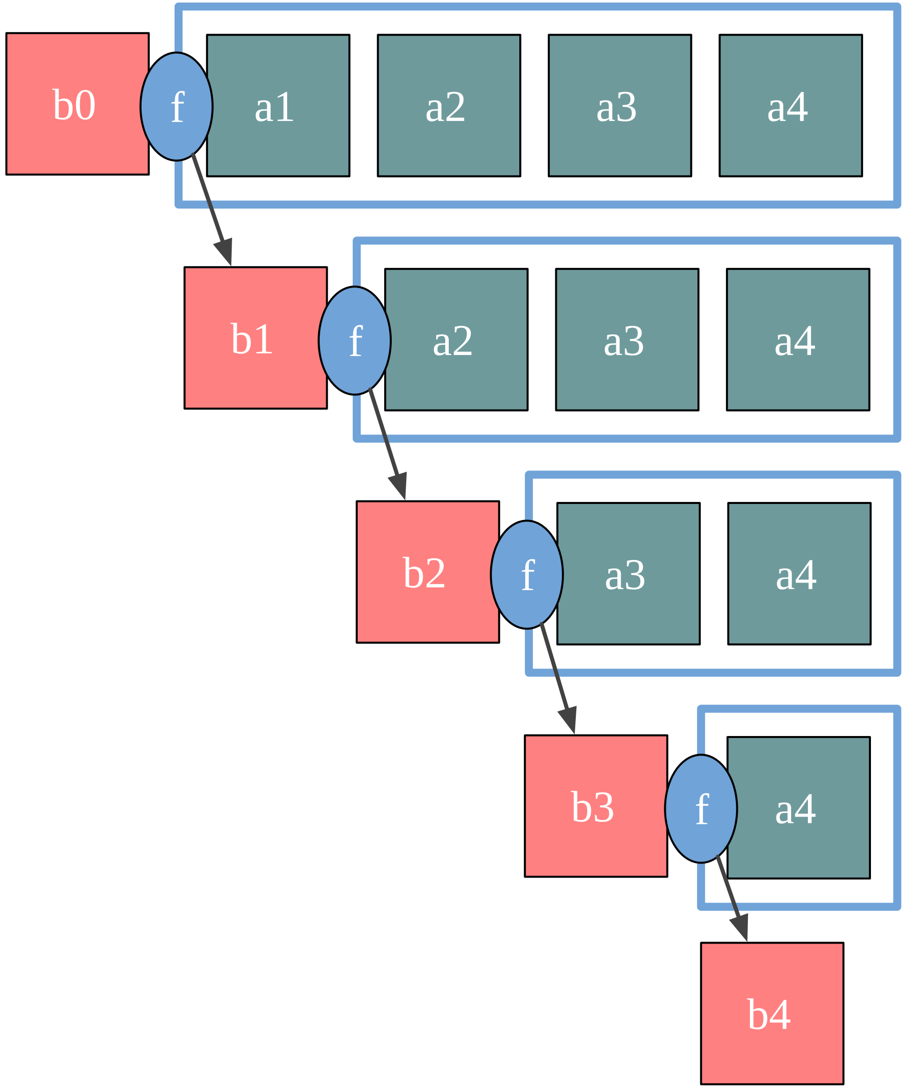
Folding



FoldLeft

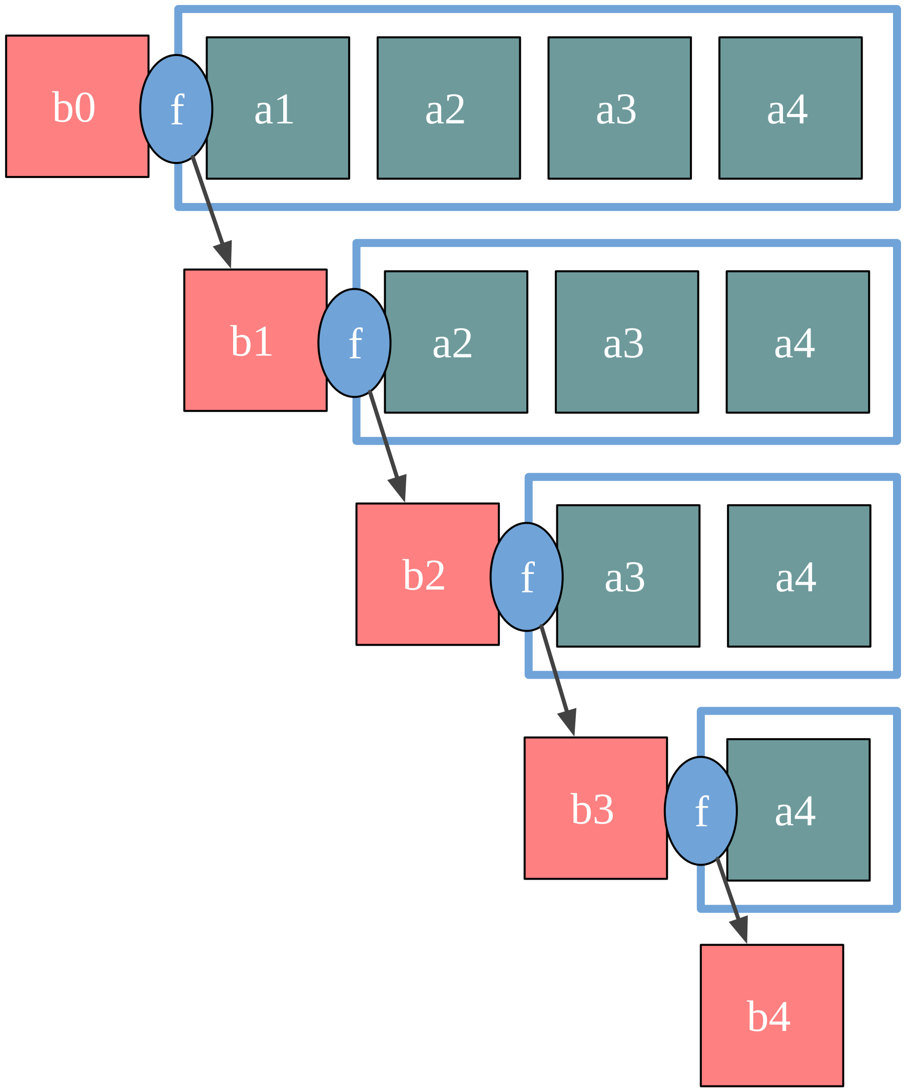


FoldLeft



FoldLeft

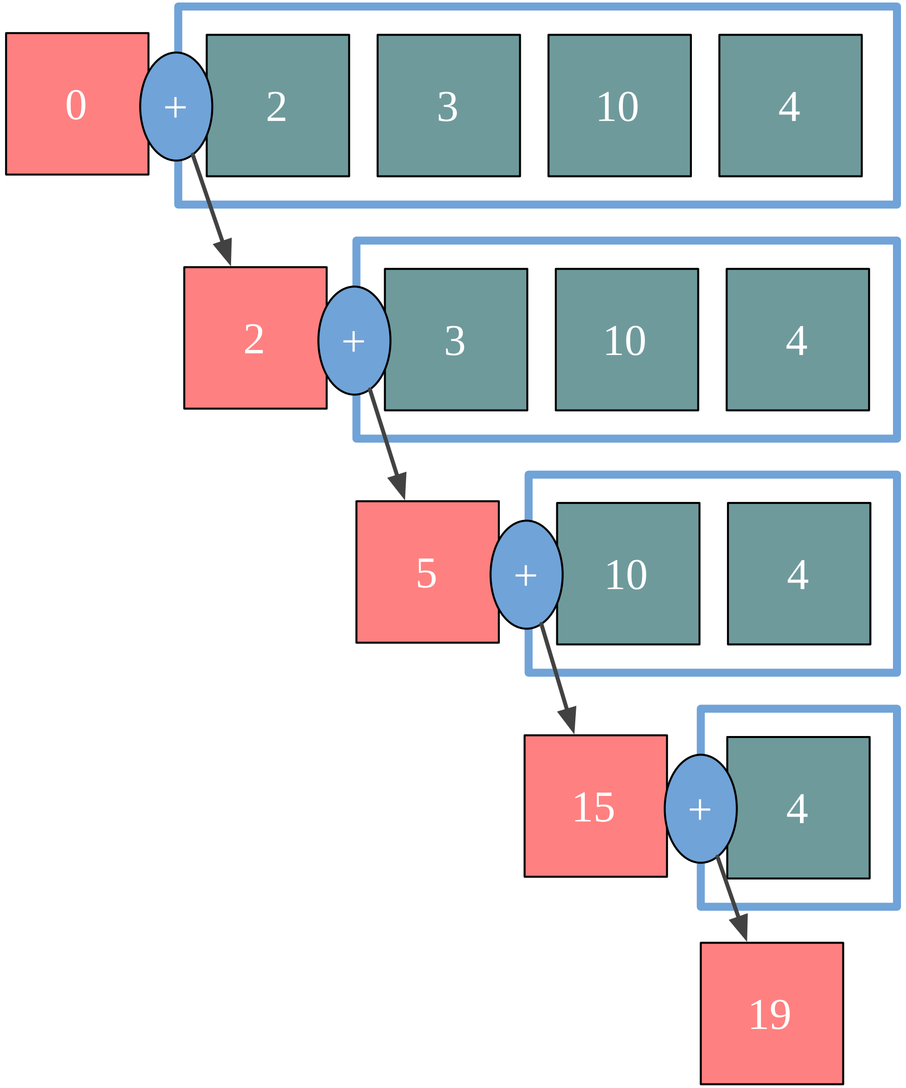
```scala
def foldLeft[A, B](fa: List[A], b: B)(f: (B, A) => B): B = {
  var acc = b
  for (a <- fa) {
    acc = f(acc, a)
  }
  acc
}
```



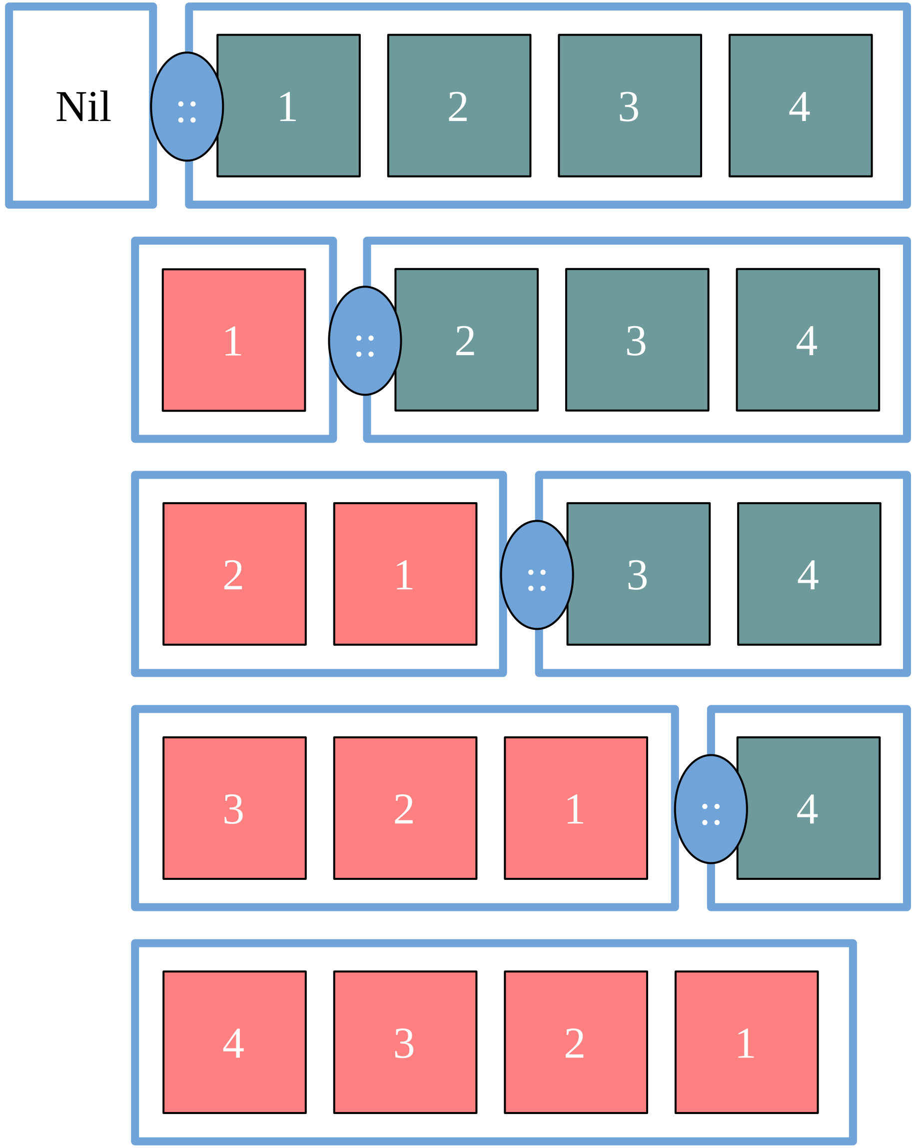
FoldLeft

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

```
sum(List(2,3,10,4))
// res99: Int = 19
```



FoldLeft



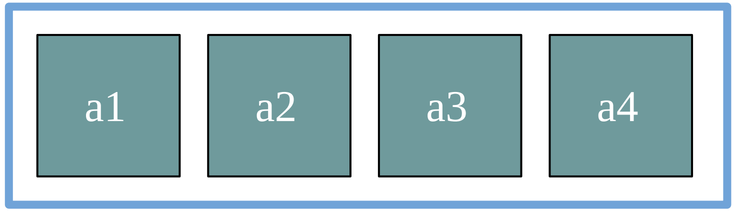

```scala
def reverse[A](xs: List[A]): List[A] =
  foldLeft(xs, List.empty[A])((acc, a) => a :: acc)
```

```scala
reverse(List(1,2,3,4))
// res100: List[Int] = List(4, 3, 2, 1)
```

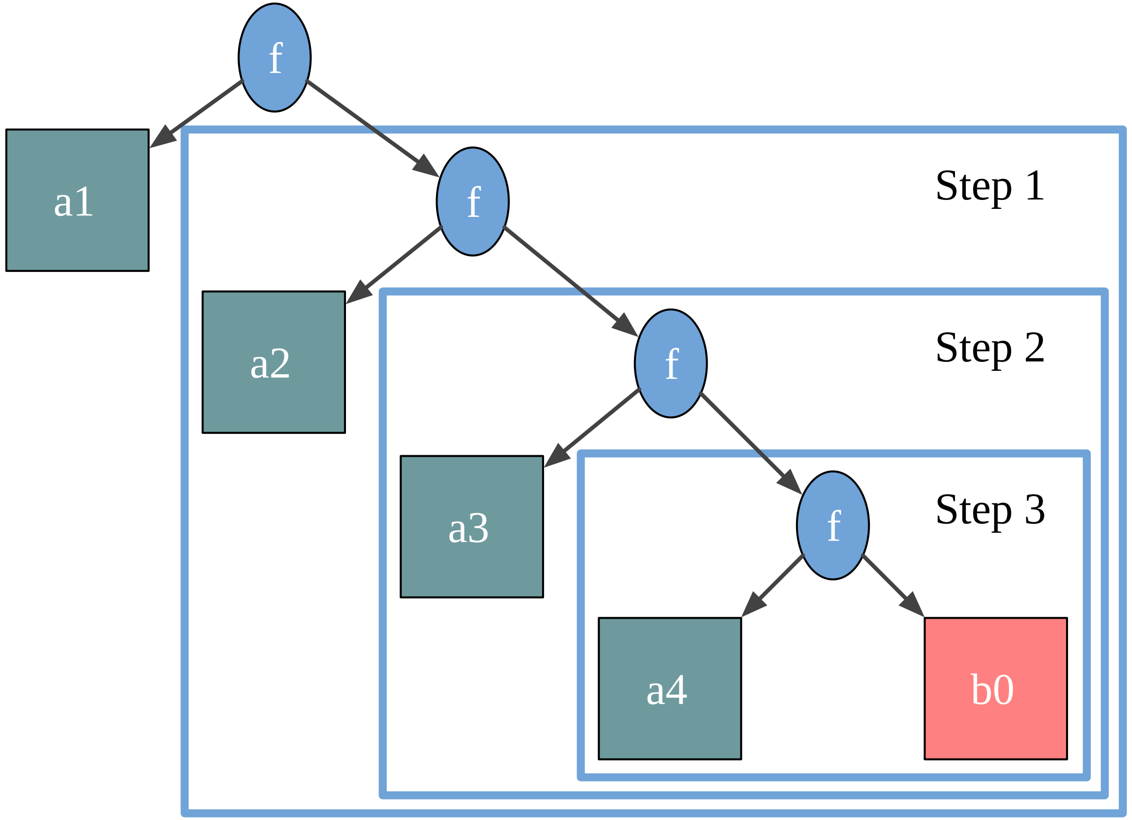
Exercise 3c-f

`exercises.function.FunctionExercises.scala`

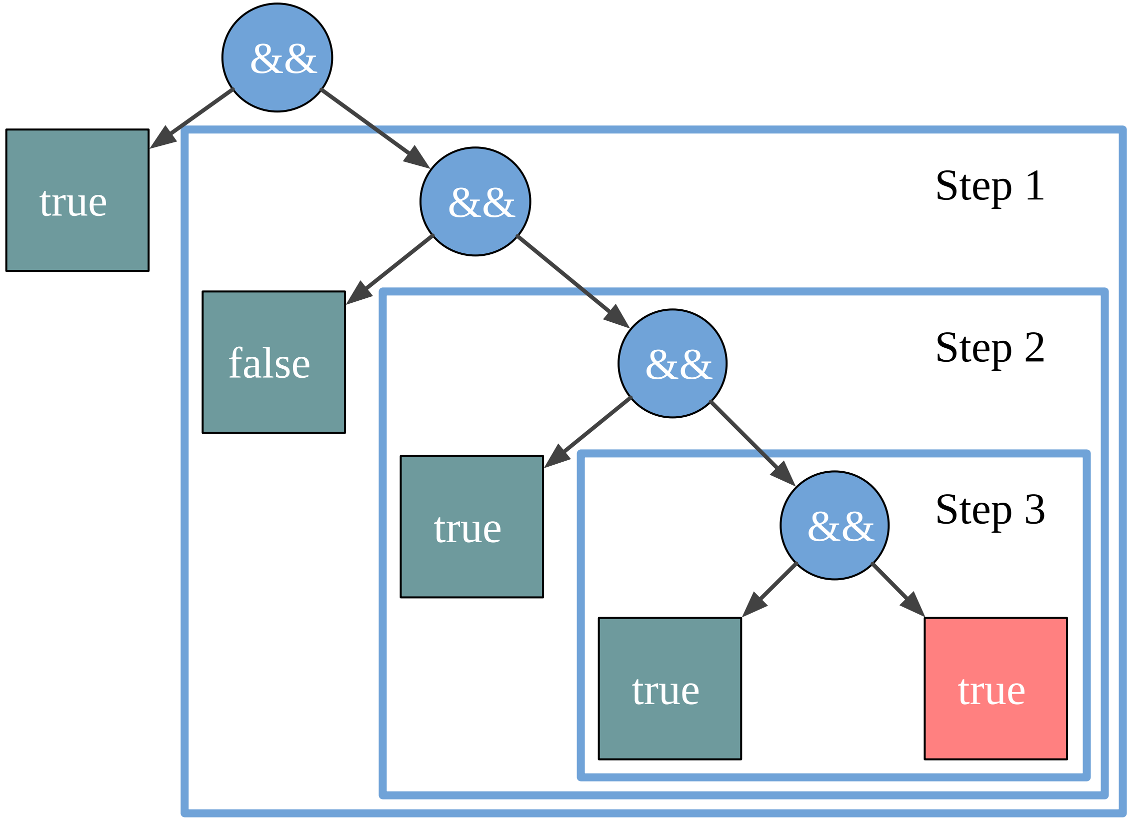
Folding



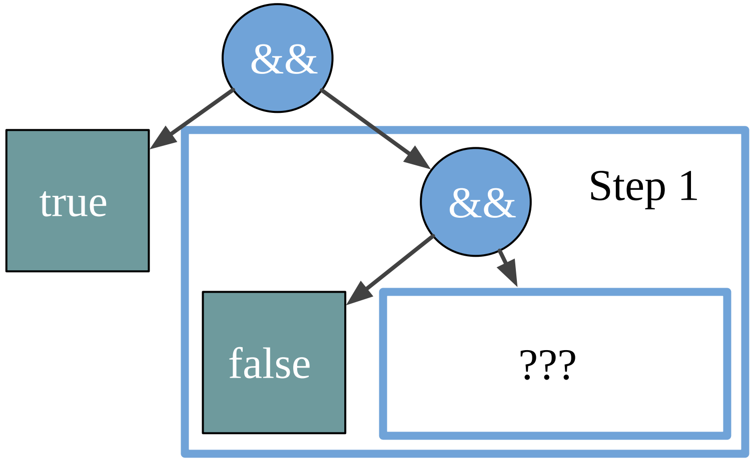
FoldRight



FoldRight

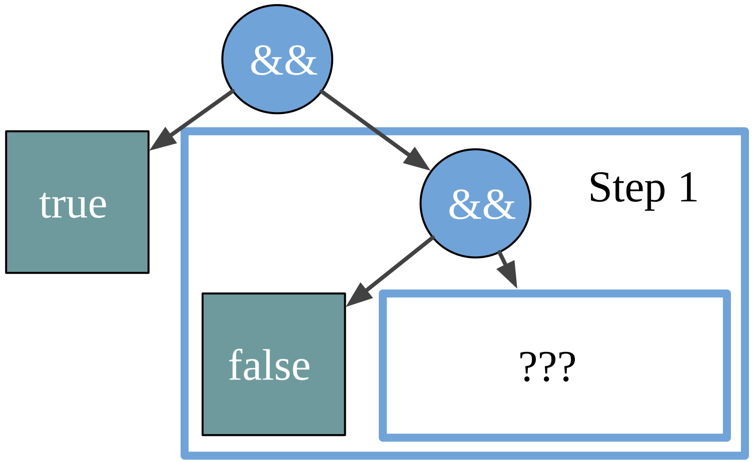


FoldRight is lazy

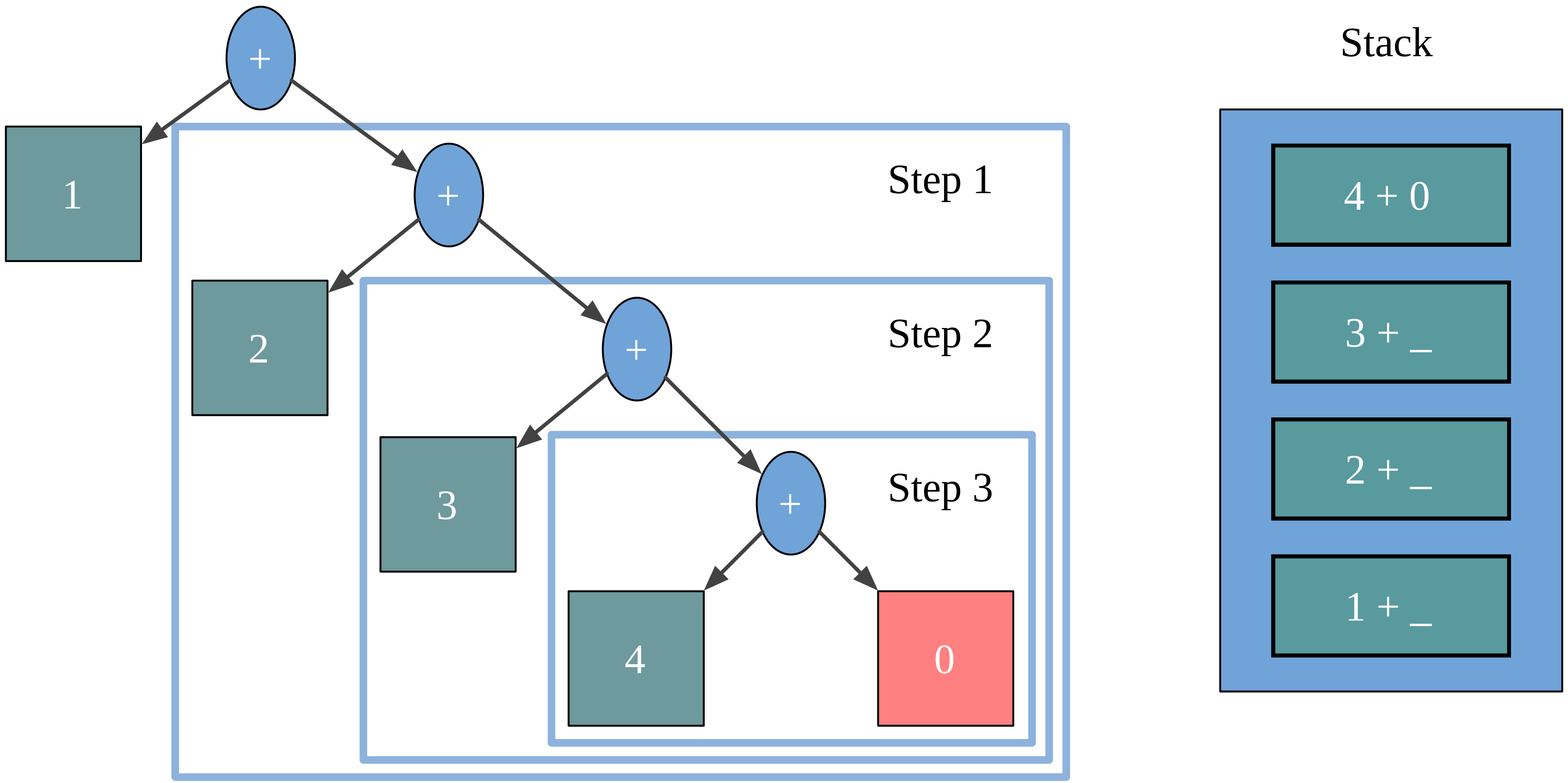


FoldRight is lazy

```
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())))
```

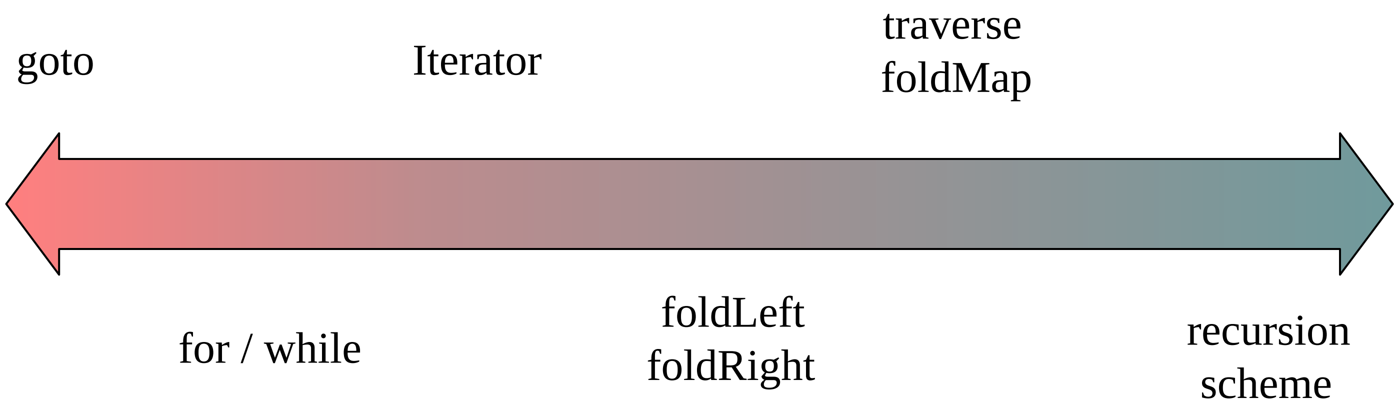
```
def foldRight[A, B](list: List[A], b: B)(f: (A, => B) => B): B

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

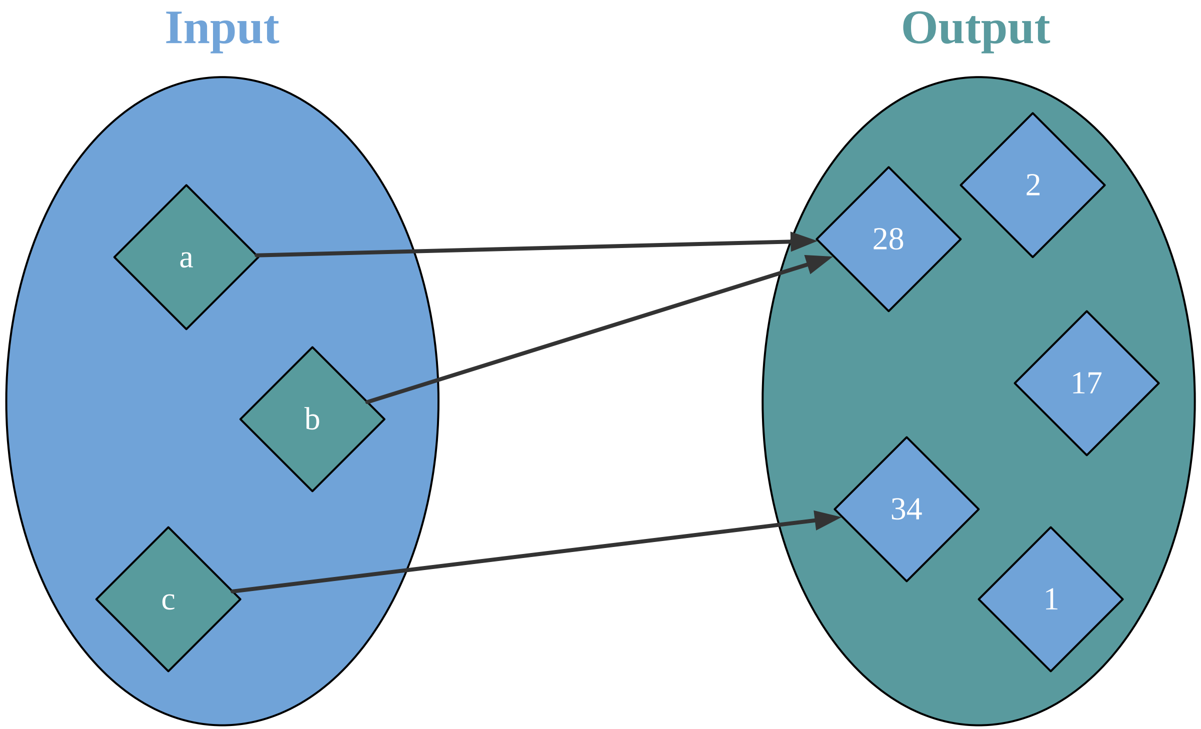
`exercises.function.FunctionExercises.scala`

Different level of abstractions



Pure function

Pure functions are mappings between two sets

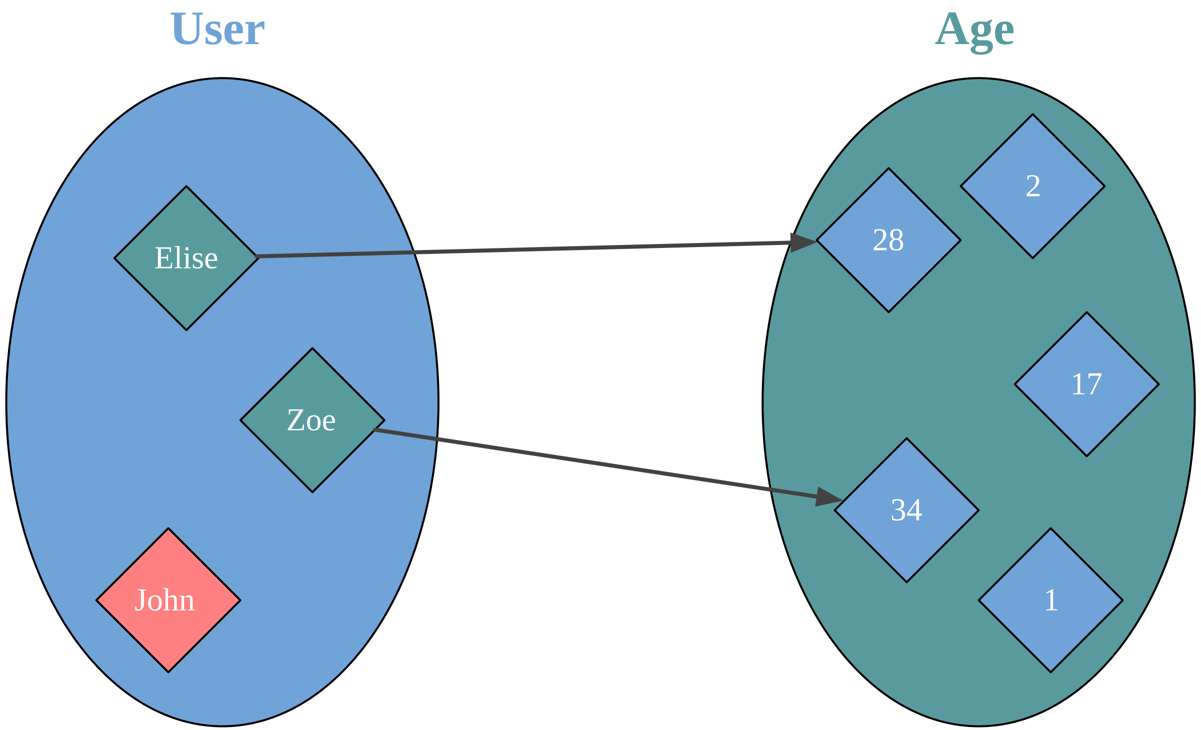


Programming function

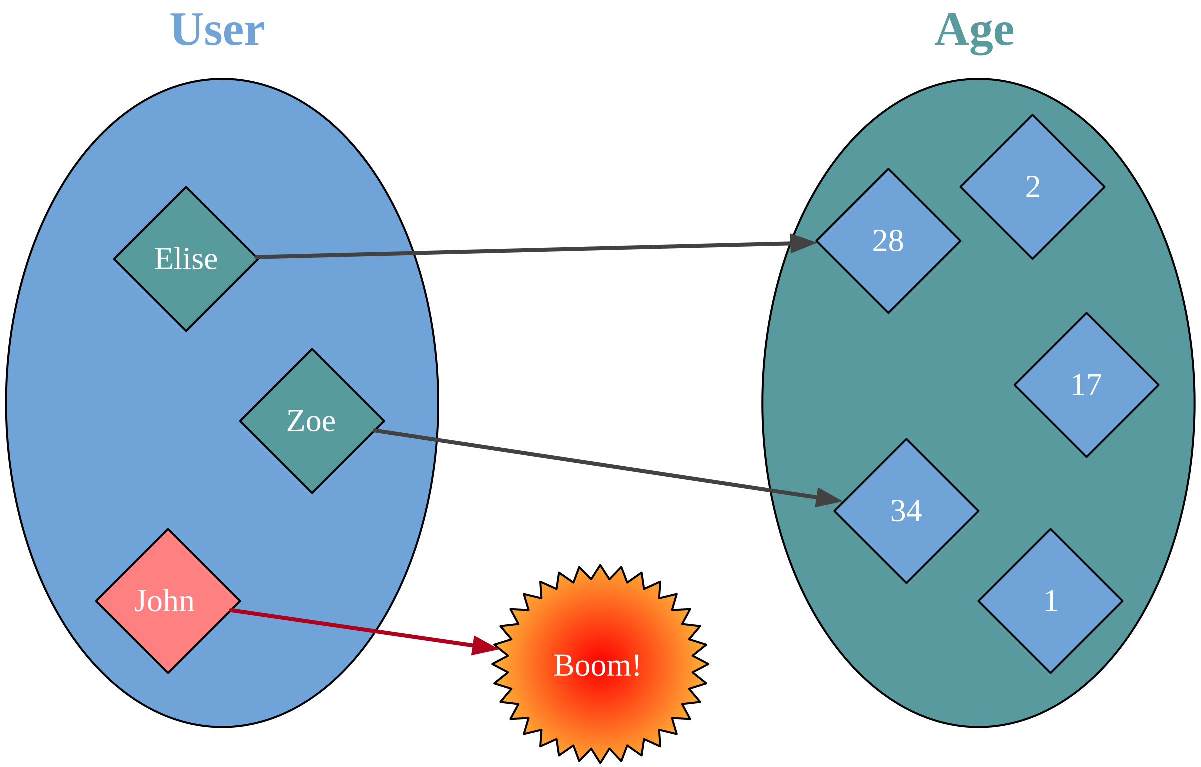
!=

Pure function

Partial function



Partial function



Partial function

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

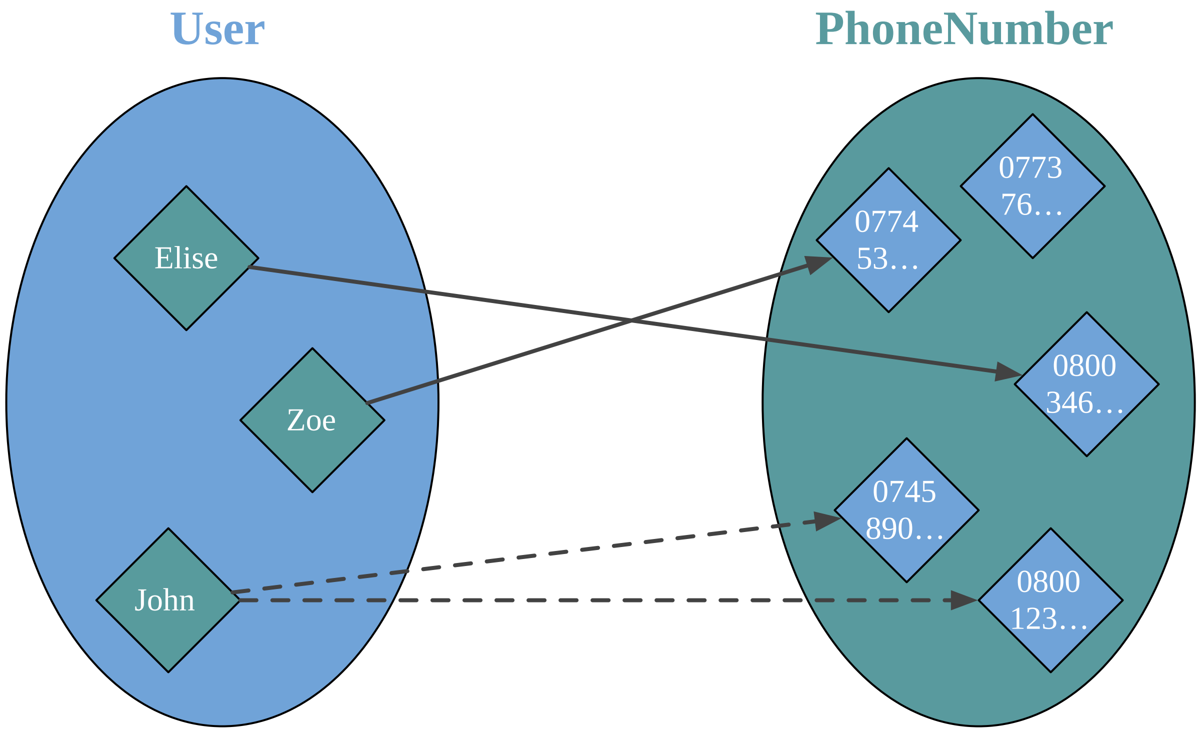
```scala
head(Nil)
// scala.MatchError: List() (of class scala.collection.immutable.Nil$)
//     at repl.Session$App101.head(1-Function.html:1155)
//     at repl.Session$App101$$anonfun$199.apply$mcI$sp(1-Function.html:1164)
//     at repl.Session$App101$$anonfun$199.apply(1-Function.html:1164)
//     at repl.Session$App101$$anonfun$199.apply(1-Function.html:1164)
```

Exception

```scala
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 repl.Session$App101.submit(1-Function.html:1182)
//     at repl.Session$App101$$anonfun$200.apply(1-Function.html:1190)
//     at repl.Session$App101$$anonfun$200.apply(1-Function.html:1190)
```

Nondeterministic



Nondeterministic

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

```scala
UUID.randomUUID()
// res102: UUID = c3d2b463-3ced-4147-8a0c-819e8889005e

UUID.randomUUID()
// res103: UUID = 9e4b75fb-1103-4903-8a9f-9b5d69288f2f
```

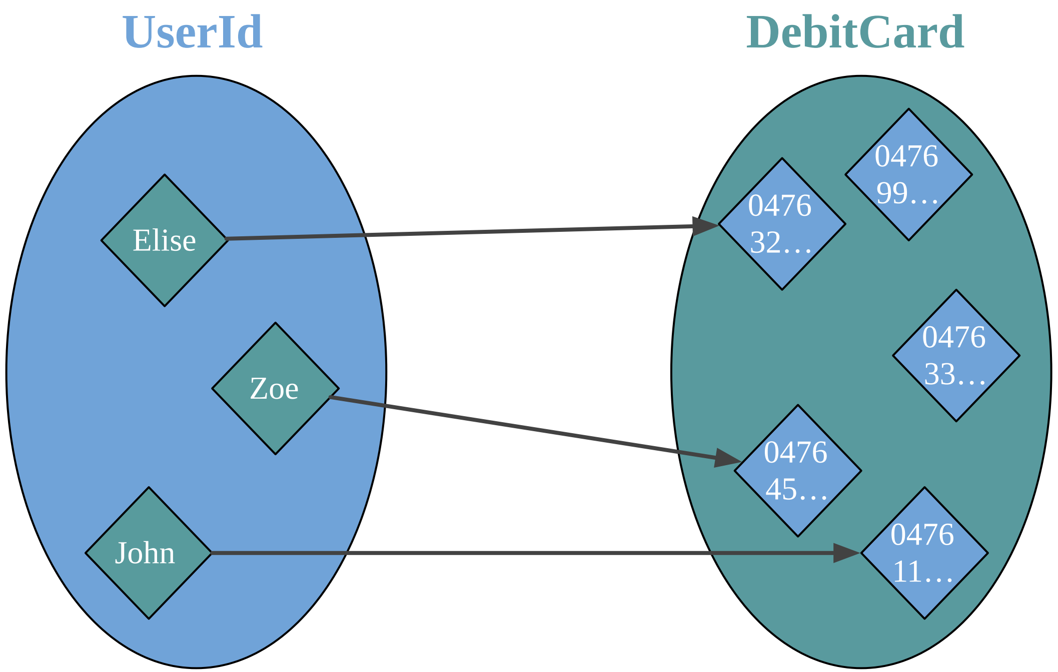
```scala
Instant.now()
// res104: Instant = 2020-03-19T19:49:49.832395Z

Instant.now()
// res105: Instant = 2020-03-19T19:49:49.833529Z
```

Mutation

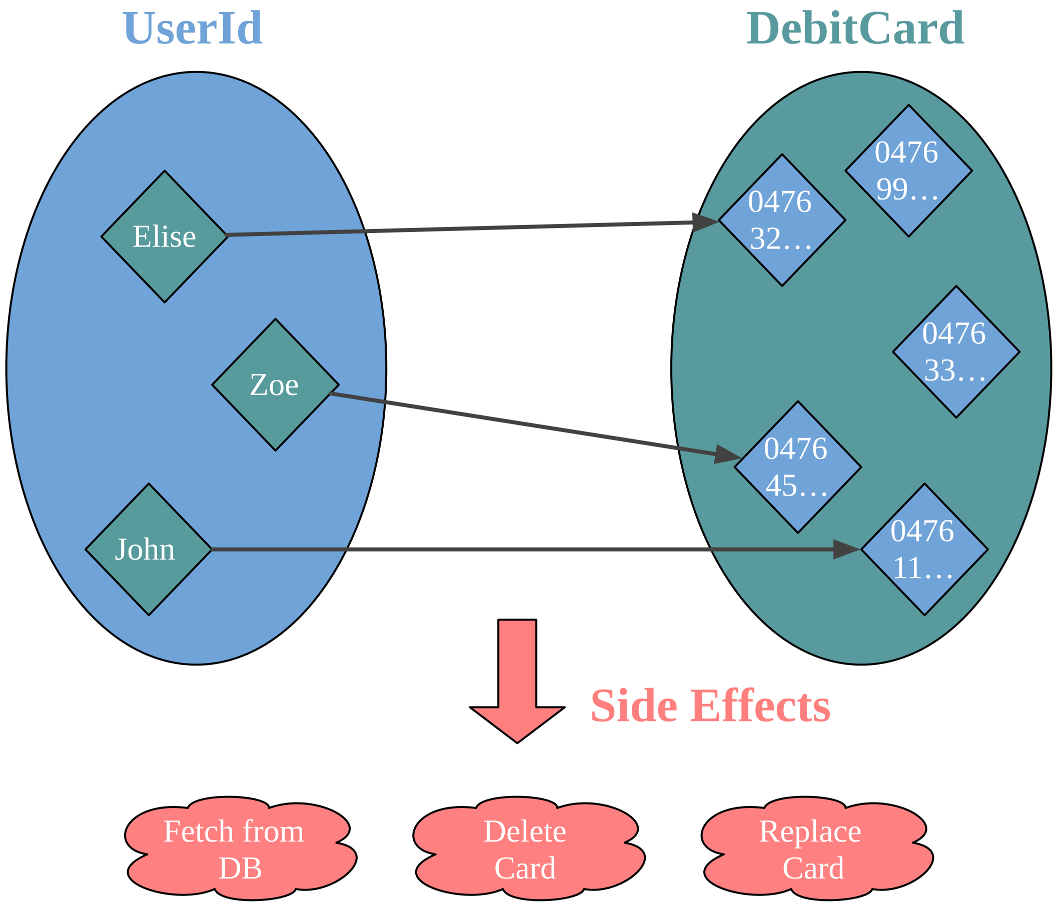
```scala
class User(initialAge: Int) {
  var age: Int = initialAge

  def getAge: Int = age

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

val john = new User(24)
```

```scala
john.getAge
// res106: Int = 24

john.setAge(32)

john.getAge
// res108: 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

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

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

Side effect

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

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

```scala
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
```

```scala
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
}
```

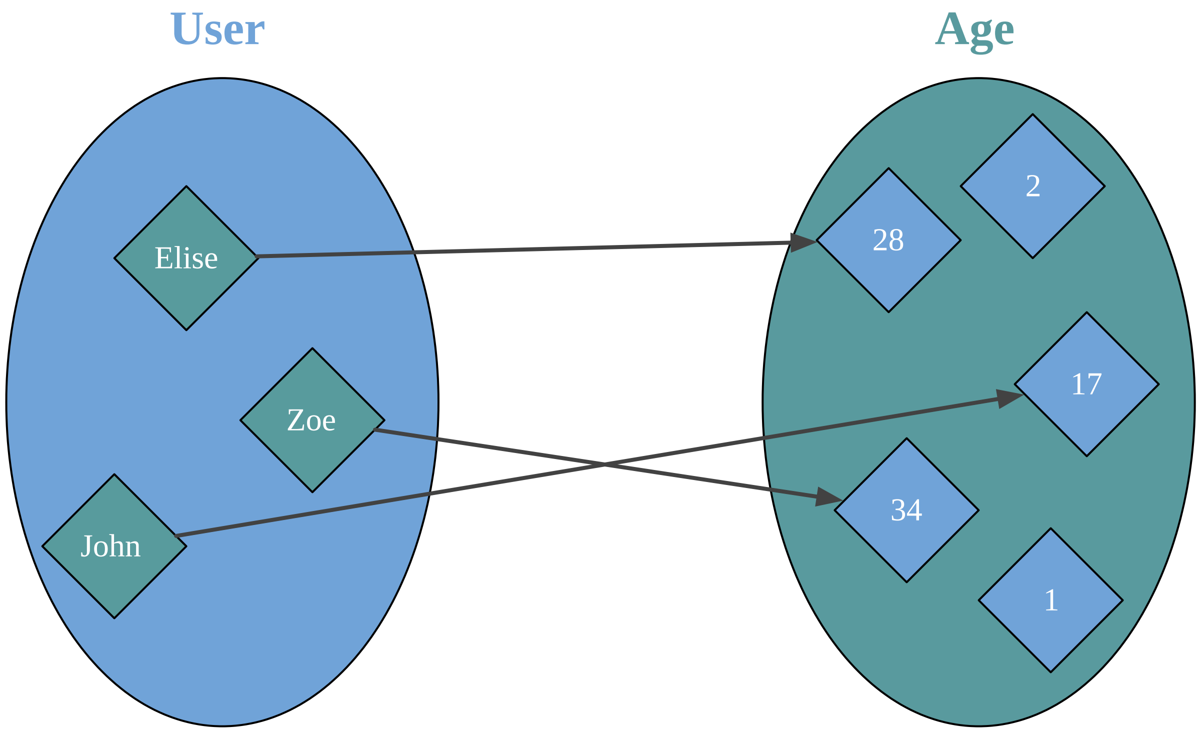
A function without side effects only returns a value

Pure function

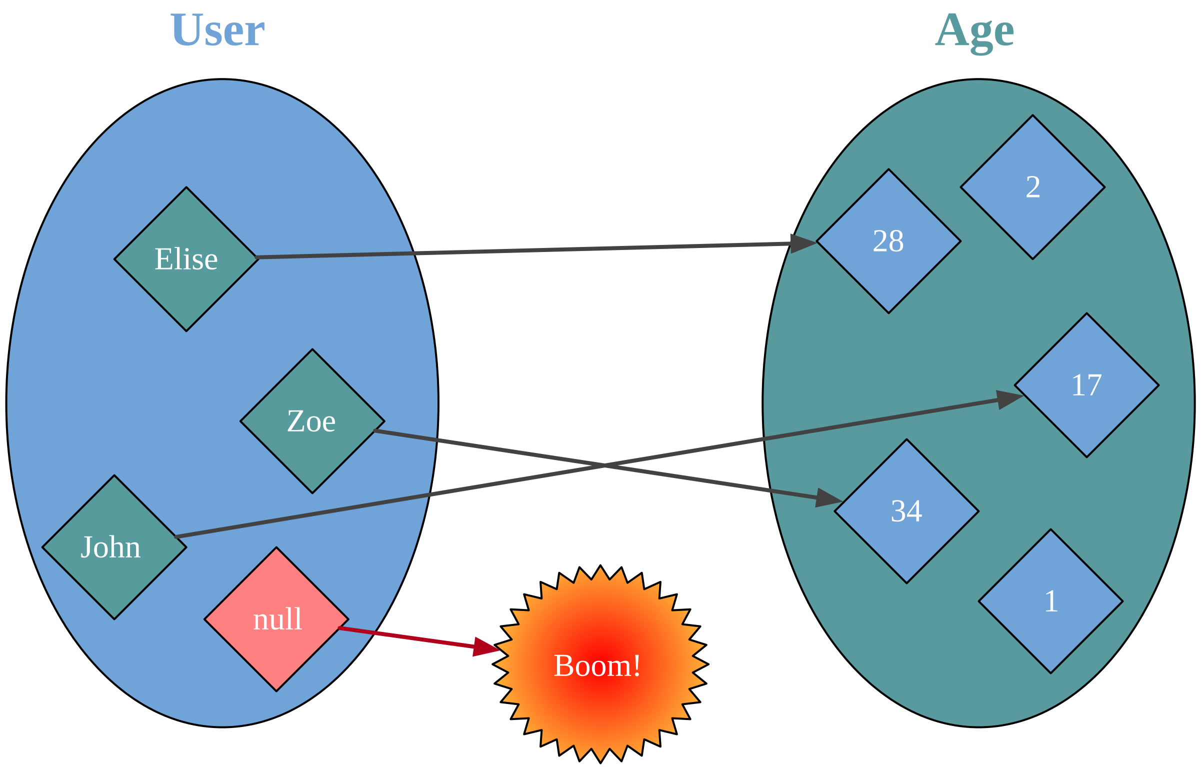
- total (not partial)
- no exception
- deterministic (not nondeterministic)
- no mutation
- no side effect

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

```
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

```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

```scala
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

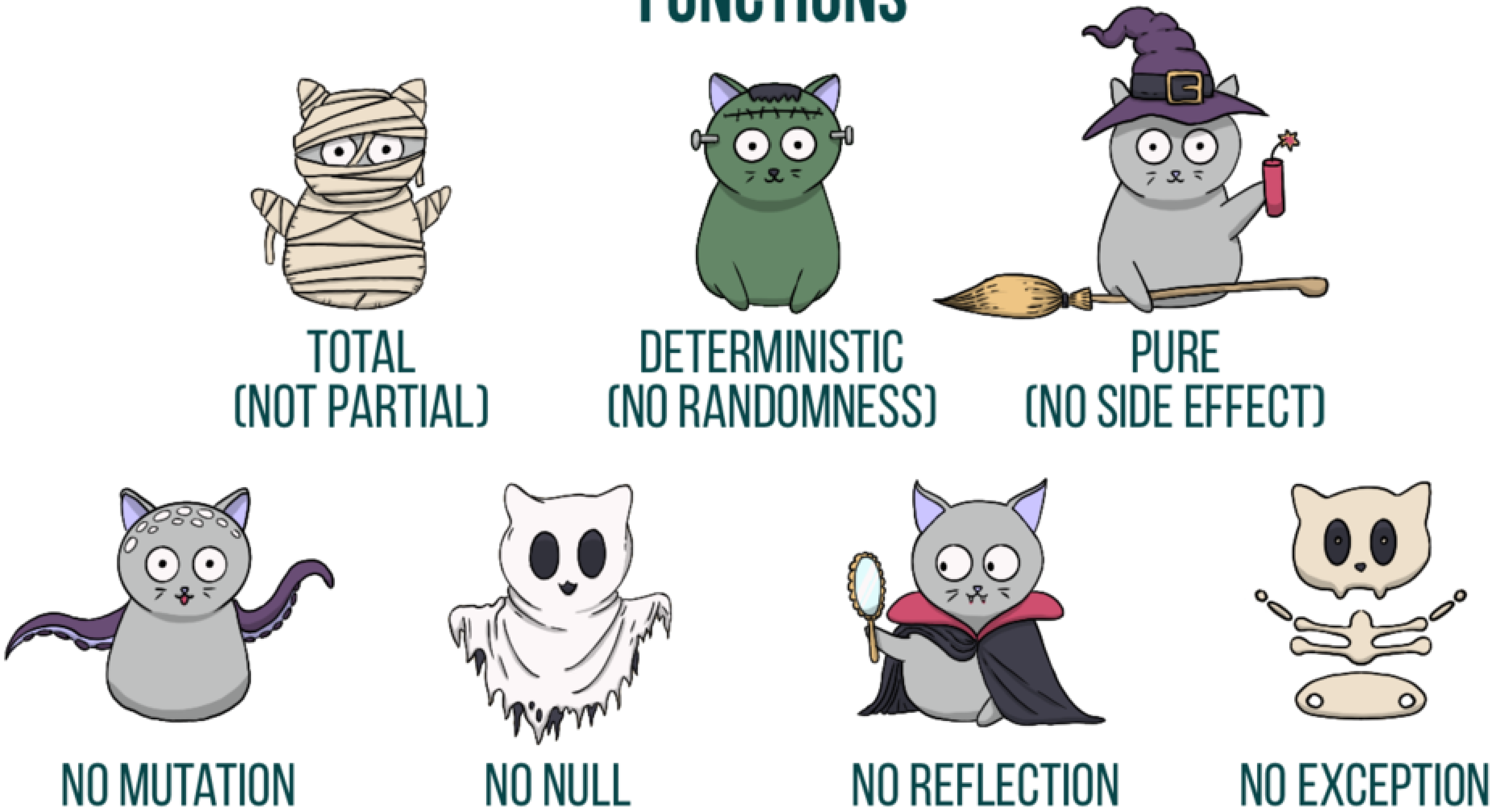
Any, AnyRef, AnyVal are all OPEN trait

```
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

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

```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

```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)
}
```

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

```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))
}
```

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

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

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

1. Refactoring: de-duplicate

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

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

Counter example

```scala
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

```scala
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

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

```scala
Point(3, 4)
// res114: Point = Point(3, 4)
```

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

```scala
Pair(3, 4)
// res115: Pair[Int] = Pair(3, 4)

Pair("John", "Doe")
// res116: 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))
// res117: Pair[Int] = Pair(5, 1)
swap(Pair("John", "Doe"))
// res118: 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

```scala
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)
  }
```

```scala
swap(Pair(1, 5))
// res120: Pair[Int] = Pair(2, 4)
```

```scala
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$App119.swap(1-Function.html:1434)
//     at repl.Session$App119$$anonfun$223.apply(1-Function.html:1450)
//     at repl.Session$App119$$anonfun$223.apply(1-Function.html:1450)
```

Type erasure

```scala
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 TM

```
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

```scala
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"
  }
```

```scala
showPair(Pair(10, 99))
showPair(Pair(1.12345, 0.000001))
showPair(Pair("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    => truncate2(x)
    case _            => "N/A"
  }
```

```scala
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)
//            ^^^^
```