
FOUNDATION

Plan

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

Functions

Val function (Lambda)

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

Def function (Method)

```
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)

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

Val function (Lambda or anonymous function)

```
(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)
// 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

```
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 ")
// res17: String = "Hello Hello Hello "

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

Def function (Method)

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

Def function (Method)

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

```
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)

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

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

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

Def function (Method)

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

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

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

Function arguments

```
import java.time.LocalDate

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

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

Function arguments

```
import java.time.LocalDate

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

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

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

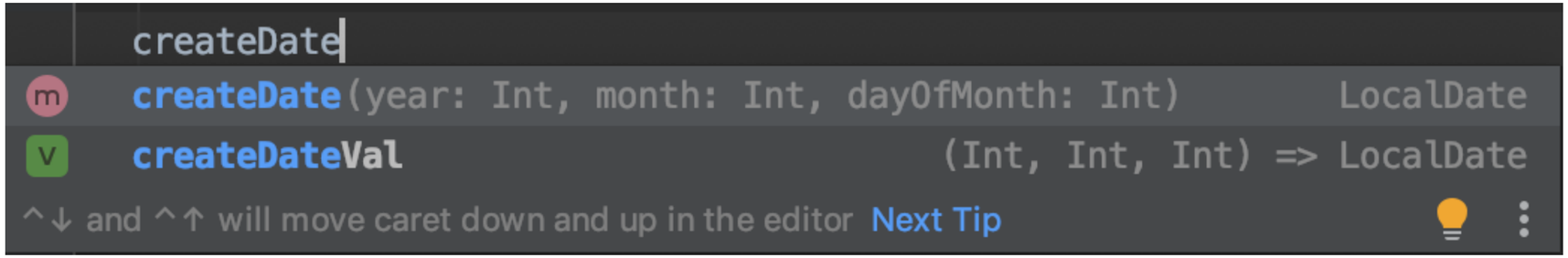
Function arguments

```
import java.time.LocalDate

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

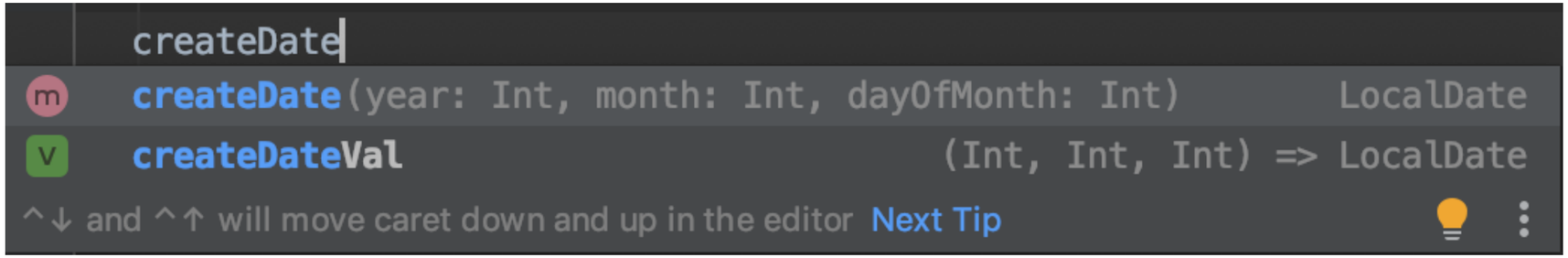
```
val createDateVal: (Int, Int, Int) => LocalDate =
  (year, month, dayOfMonth) => ...
```

```
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

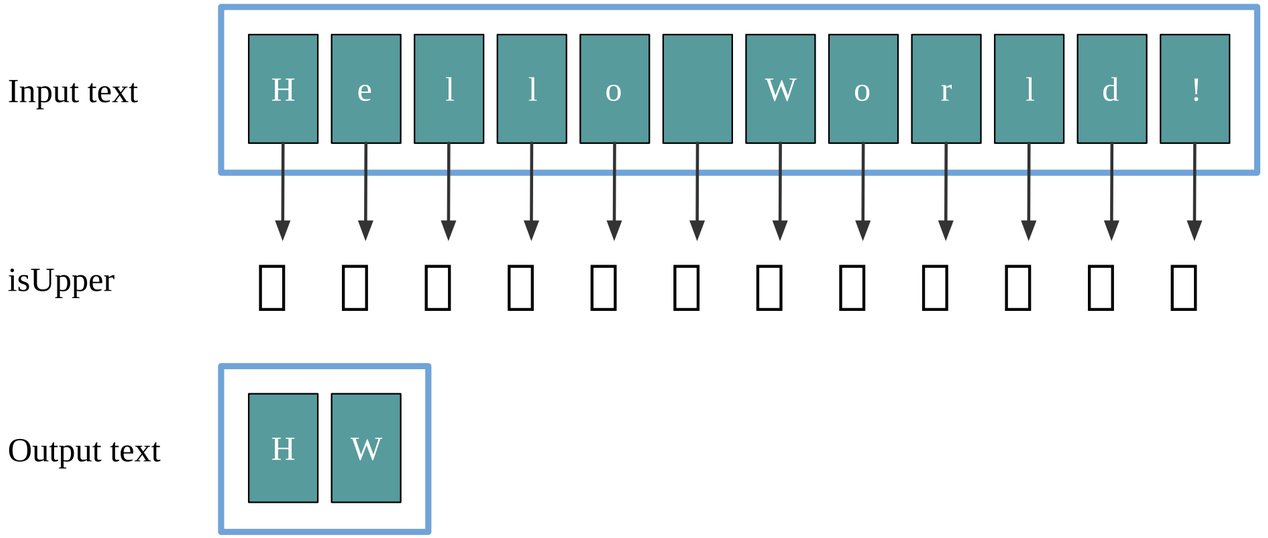
- Val functions are an ordinary objects
- Use def functions for API
- Easy to convert def to val

Functions as input

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

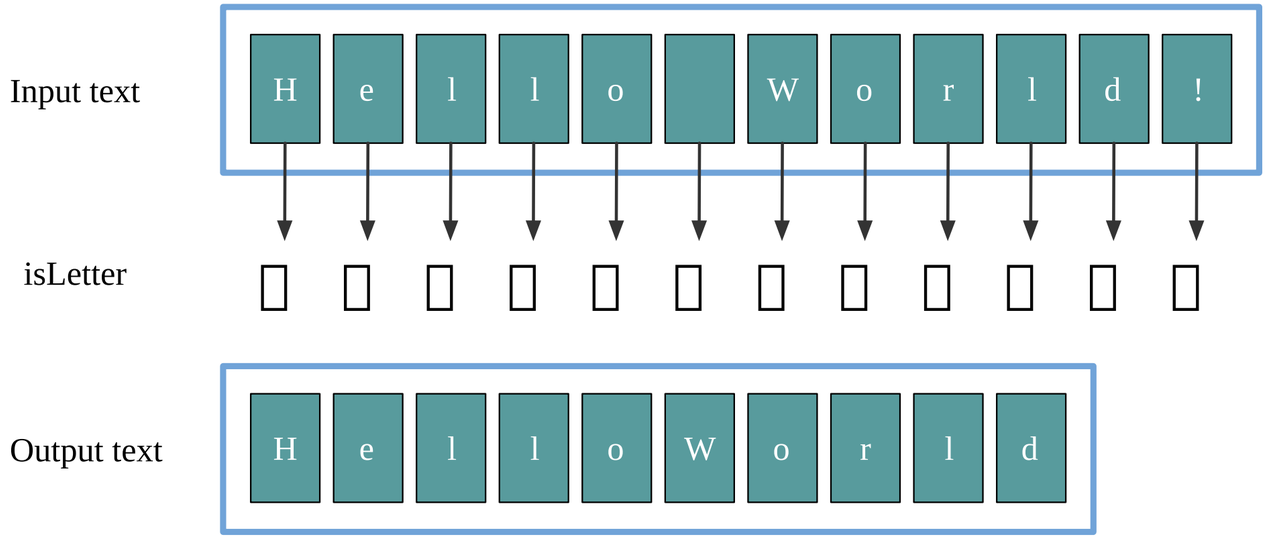
Functions as input

```
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"
```

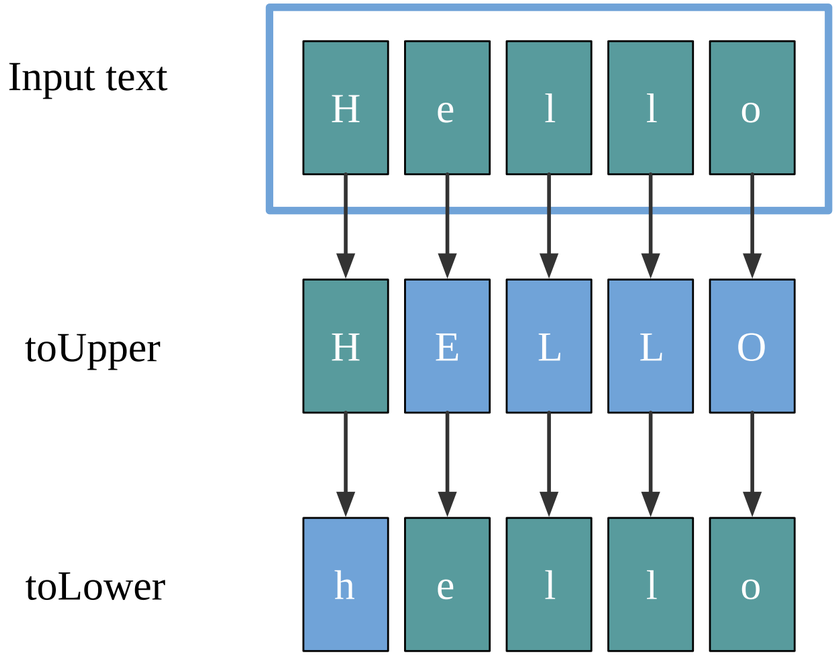
```
def lowerCase(text: String): String = {
  val characters = text.toArray
  for (i <- 0 until text.length) {
    characters(i) = characters(i).toLower
  }
  new String(characters)
}
```

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

Capture pattern

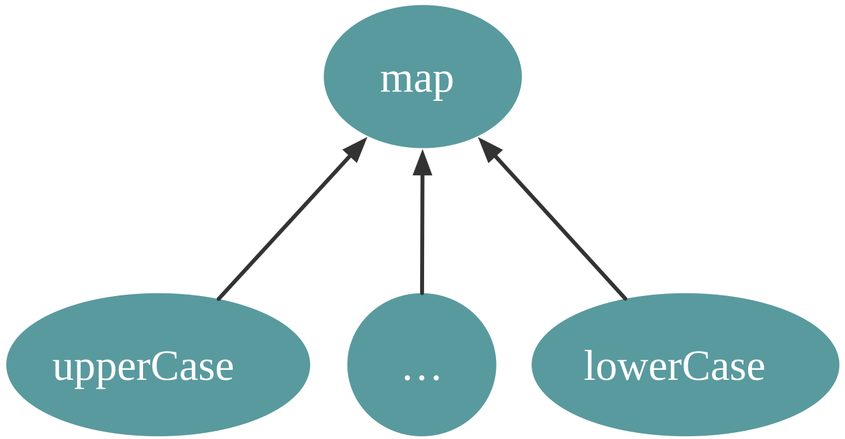
```
def map(text: String, update: Char => Char): String =
  val characters = text.toArray
  for (i <- 0 until text.length) {
    characters(i) = update(characters(i))
  }
  new String(characters)
}
```

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

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



Capture pattern

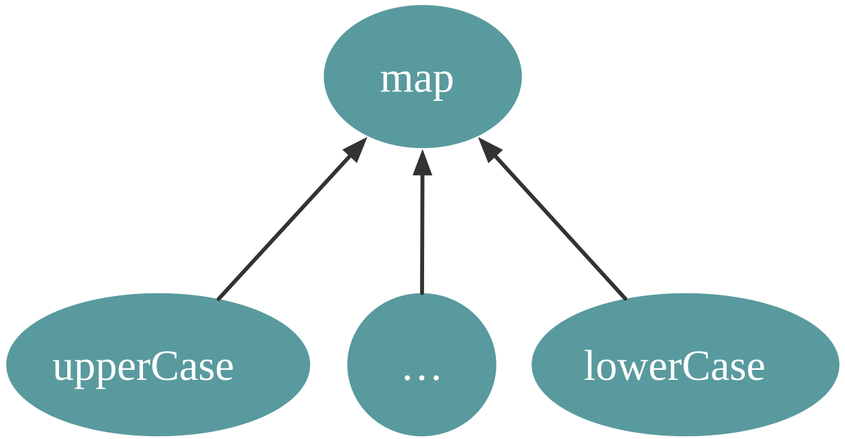
```
def map(text: String, update: Char => Char): String =
  val characters = text.toArray
  for (i <- 0 until text.length) {
    characters(i) = update(characters(i))
  }
  new String(characters)
}
```



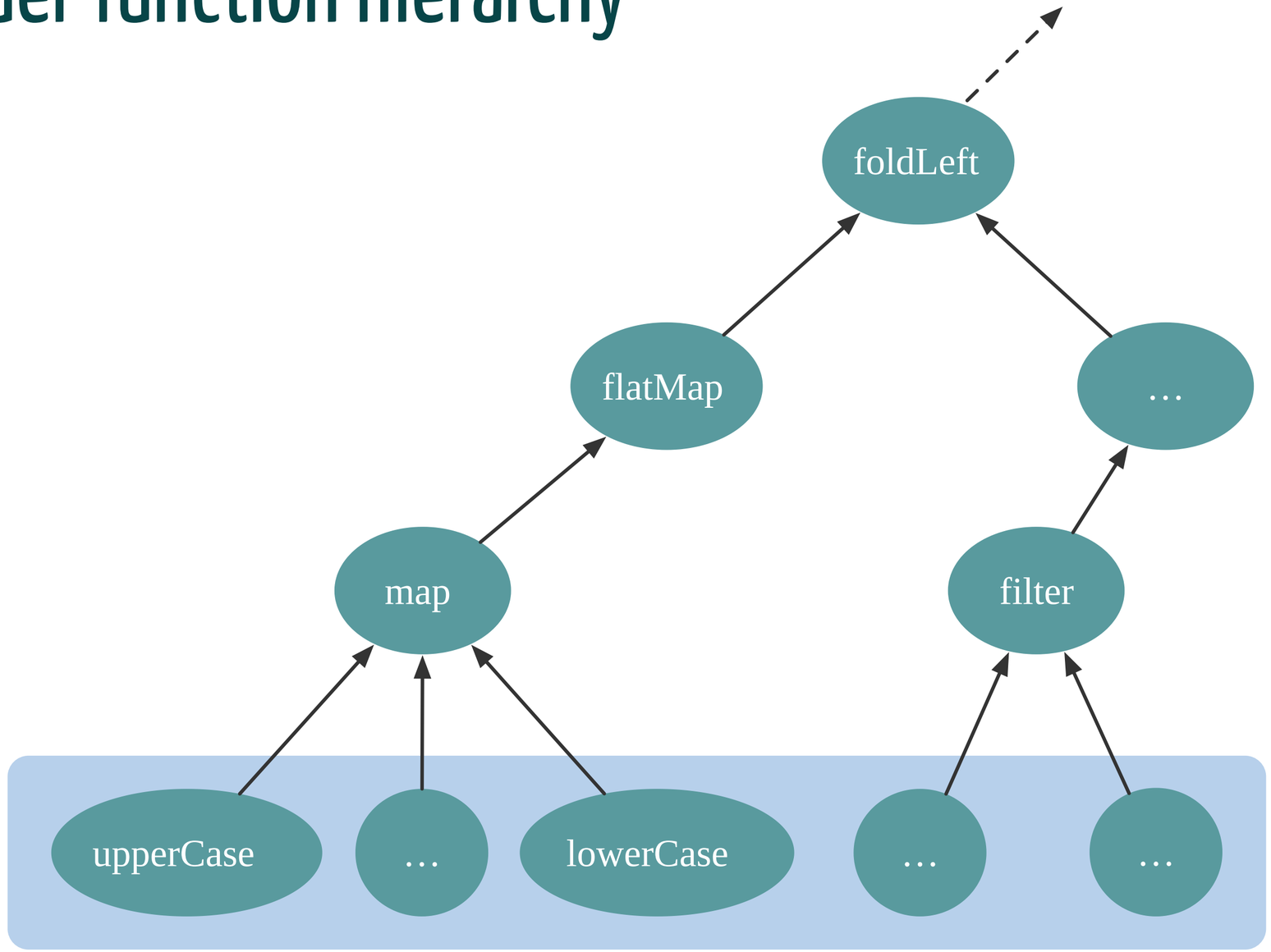

```
def upperCase(text: String): String =
  map(text, c => c.toUpper)

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

Capture pattern

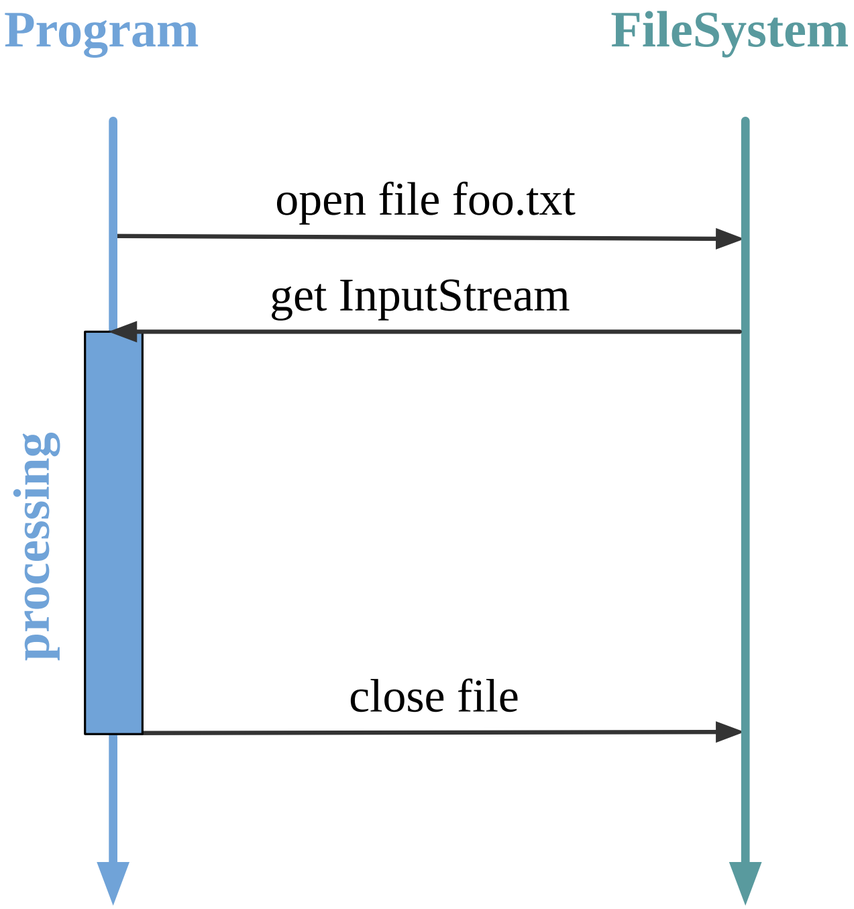
```
test("map does not modify the text size") {
  forAll((
    text  : String,
    update: Char => Char
  ) =>
    val outputText = map(text, update)
    outputText.length == text.length
  )
}
```



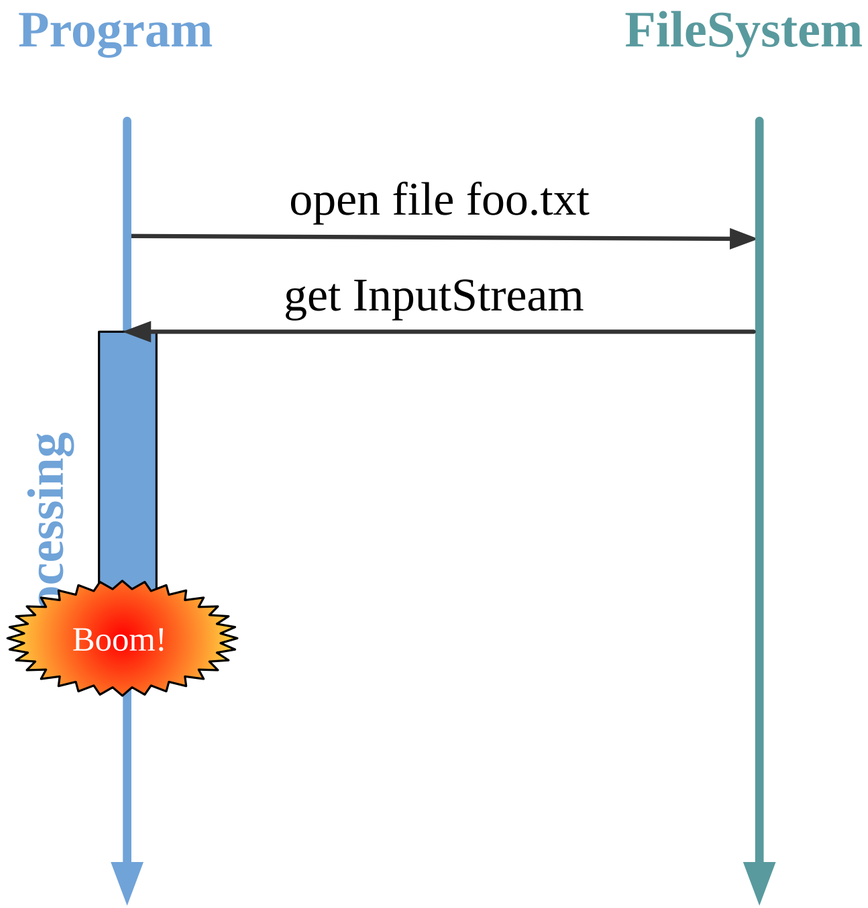
Higher order function hierarchy



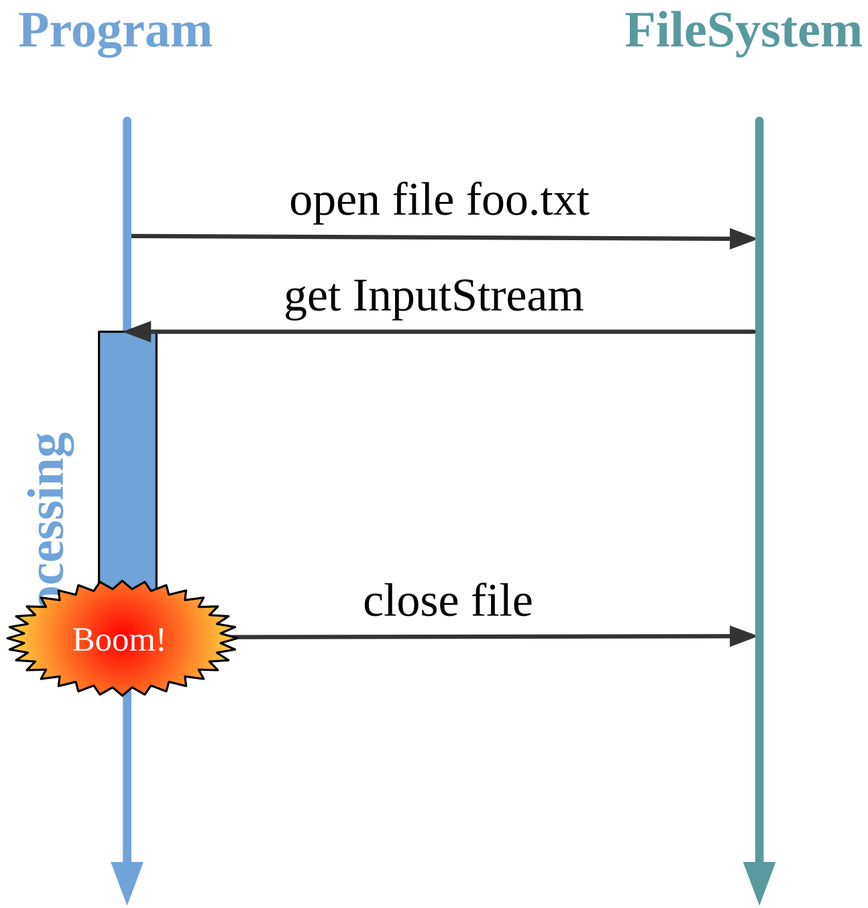
File processing



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

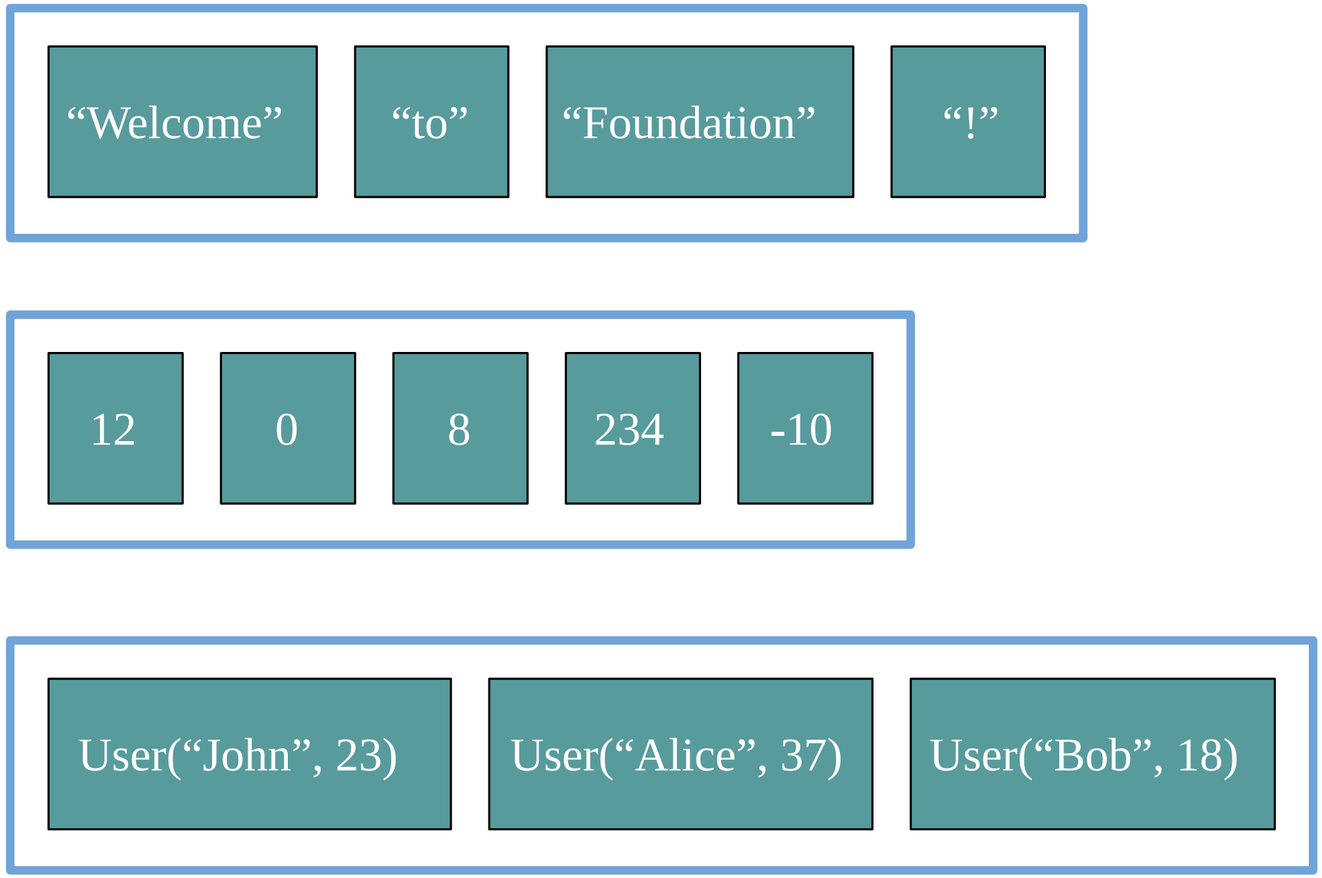
- Higher order function
- Reduce code duplication
- Improve code quality

Exercise 1: Functions as input

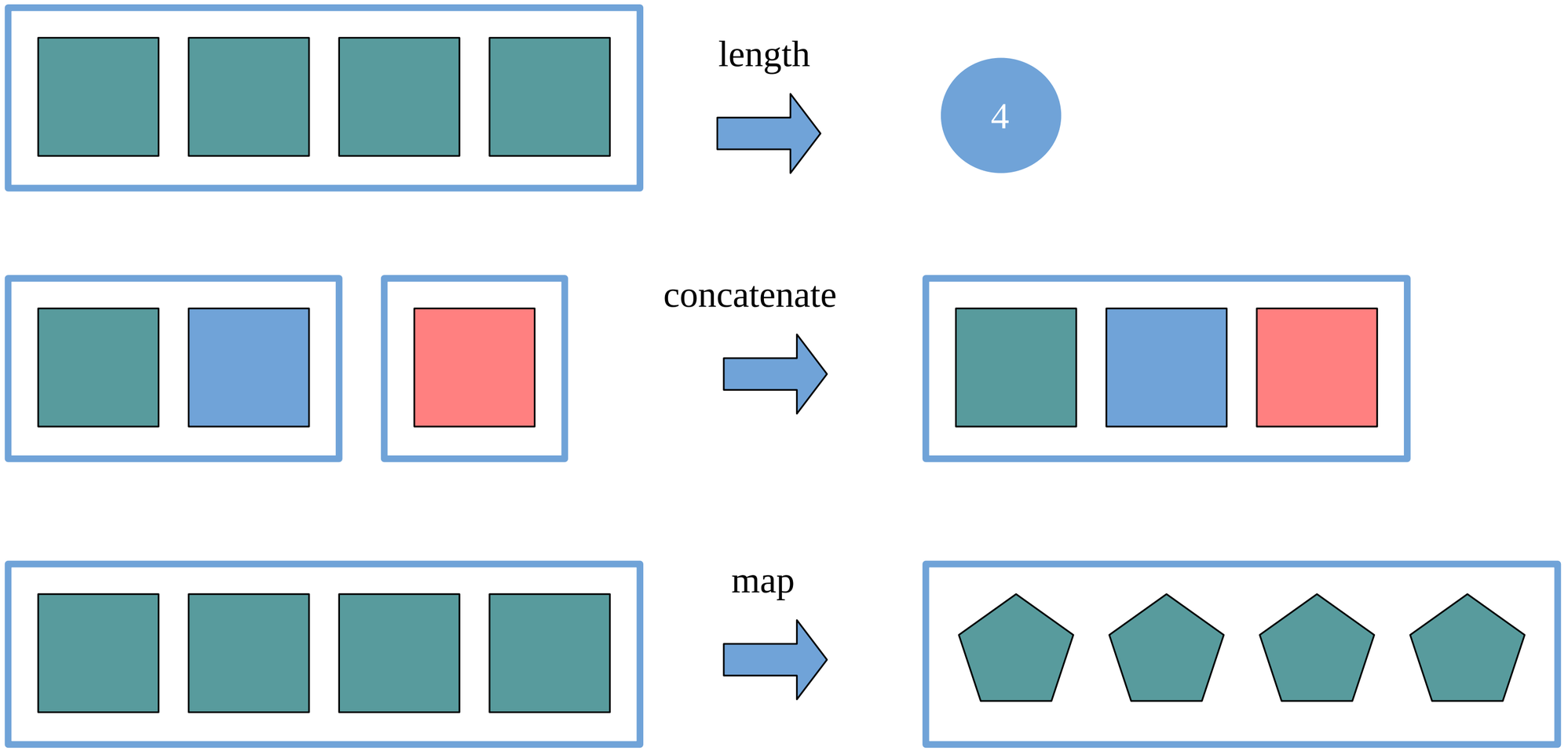
`exercises.function.FunctionExercises.scala`

Parametric functions

Sequence is a generic data structure

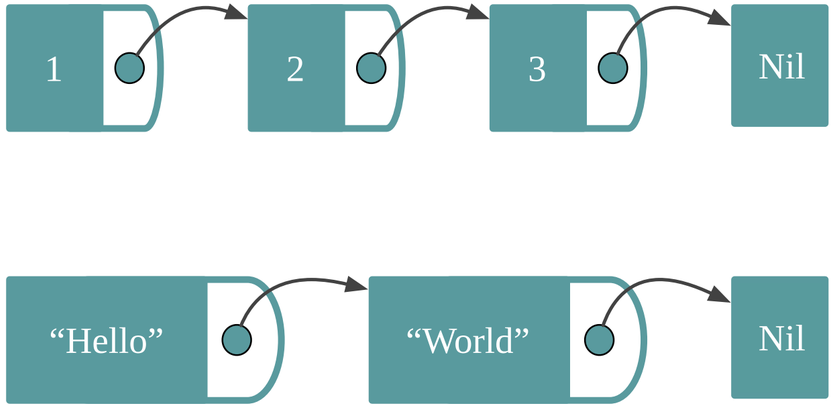


Generic operations



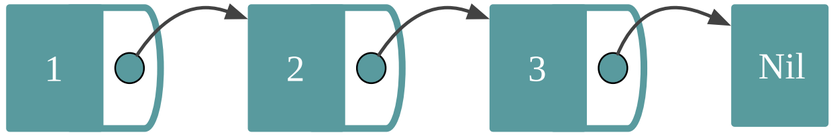
Linked List

```
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 parametric functions

- More generic, more reusable functions
- Lots of details encoded in the signature
- Applicable to most data structures, but not only …

Exercise 2: Parametric functions

`exercises.function.FunctionExercises.scala`

Functions as output

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

```
truncate(2, 0.123456789)
// res47: String = "0.12"

truncate(5, 0.123456789)
// res48: 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)
// res50: String = "0.12"

truncate5D(0.123456789)
// res51: String = "0.12345"
```

Functions as output

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

```
truncate(2, 0.123456789)
// res53: String = "0.12"
```

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

```
truncate(2)(0.123456789)
// res55: String = "0.12"
```

Functions as output

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

```
truncate(2, 0.123456789)
// res53: String = "0.12"
```

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

```
truncate(2)(0.123456789)
// res55: 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)
// res56: String = "0.12"

truncate5D(0.123456789)
// res57: 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 _
// res59: (Int, Double) => String = <function2>
```

Conversion (Currying)

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

```
truncate _
// res59: (Int, Double) => String = <function2>
```

```
(truncate _).curried
// res60: Int => Double => String = scala.Function2$$Lambda$5040/0x0000000101a16840@3e387c87
```

Exercise 3: Functions as output

`exercises.function.FunctionExercises.scala`

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)
// res61: Pair[Int] = Pair(7, 5)

Pair(2, 3).zipWith(Pair("Hello ", "World "), replicate)
// res62: 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)
// res65: 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)
// res65: Pair[Int] = Pair(7, 5)
```

```
Pair(0, 2).zipWith(Pair(7, 3))(_ + _)
// res66: 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]
```

Finish Exercise 3: Parametric functions

`exercises.function.FunctionExercises.scala`

Two apparently useless functions

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

```
identity(5)
// res67: Int = 5

identity("Hello")
// res68: String = "Hello"
```

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

```
constant(5)("Hello")
// res69: Int = 5

constant("Hello")(5)
// res70: 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()
// res71: Boolean = true

toggle()
// res72: 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)

}
```

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
```

What is the type of `identityVal`?

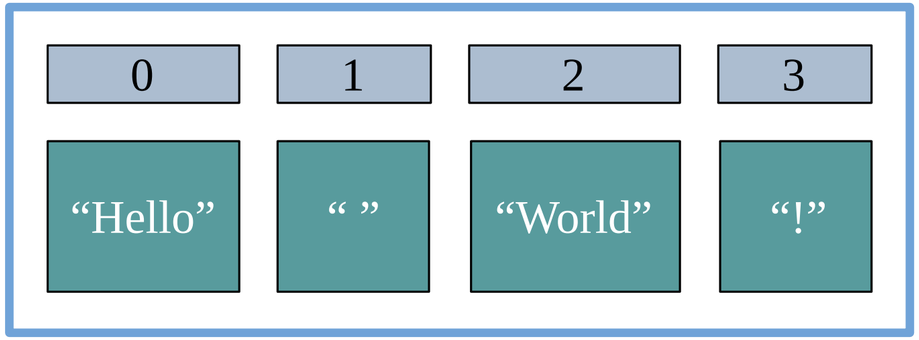
```
def identity[A](value: A): A =
  value
```

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

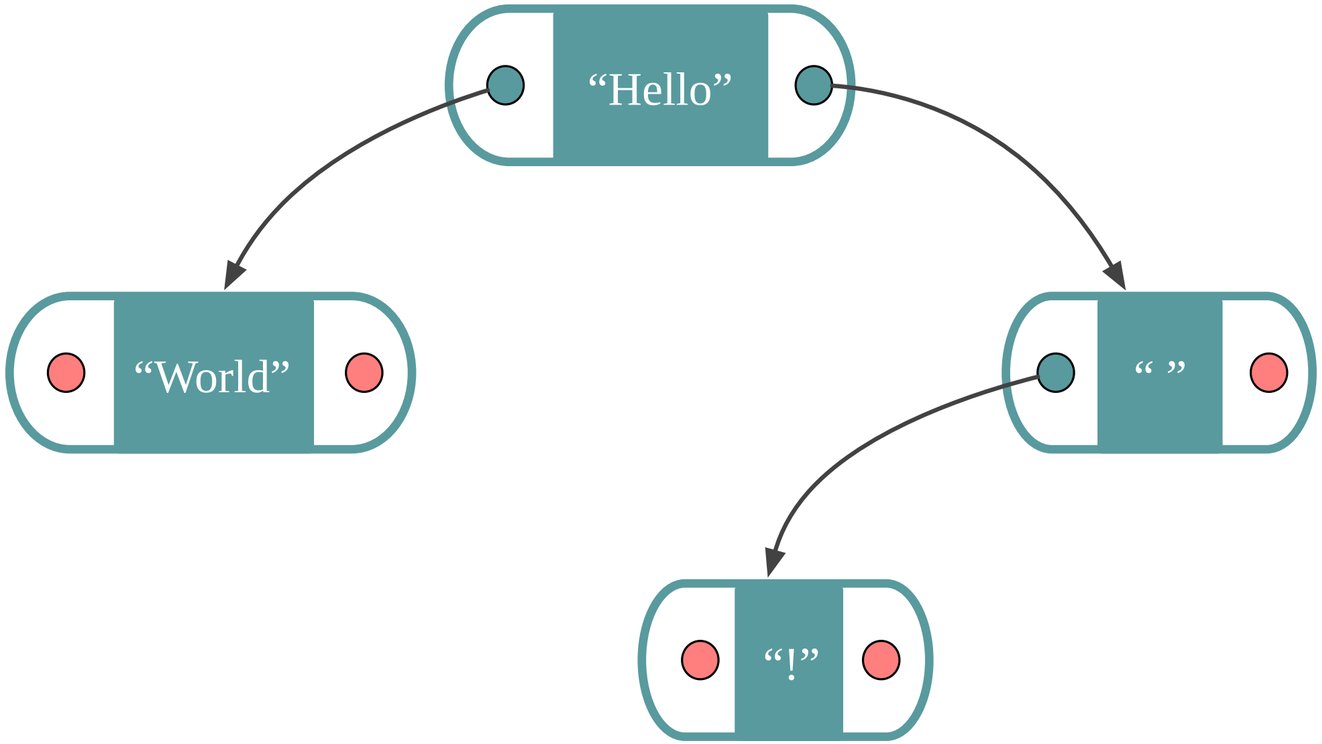
```
identityVal(4)
// res78: Int = 4
```

Iteration

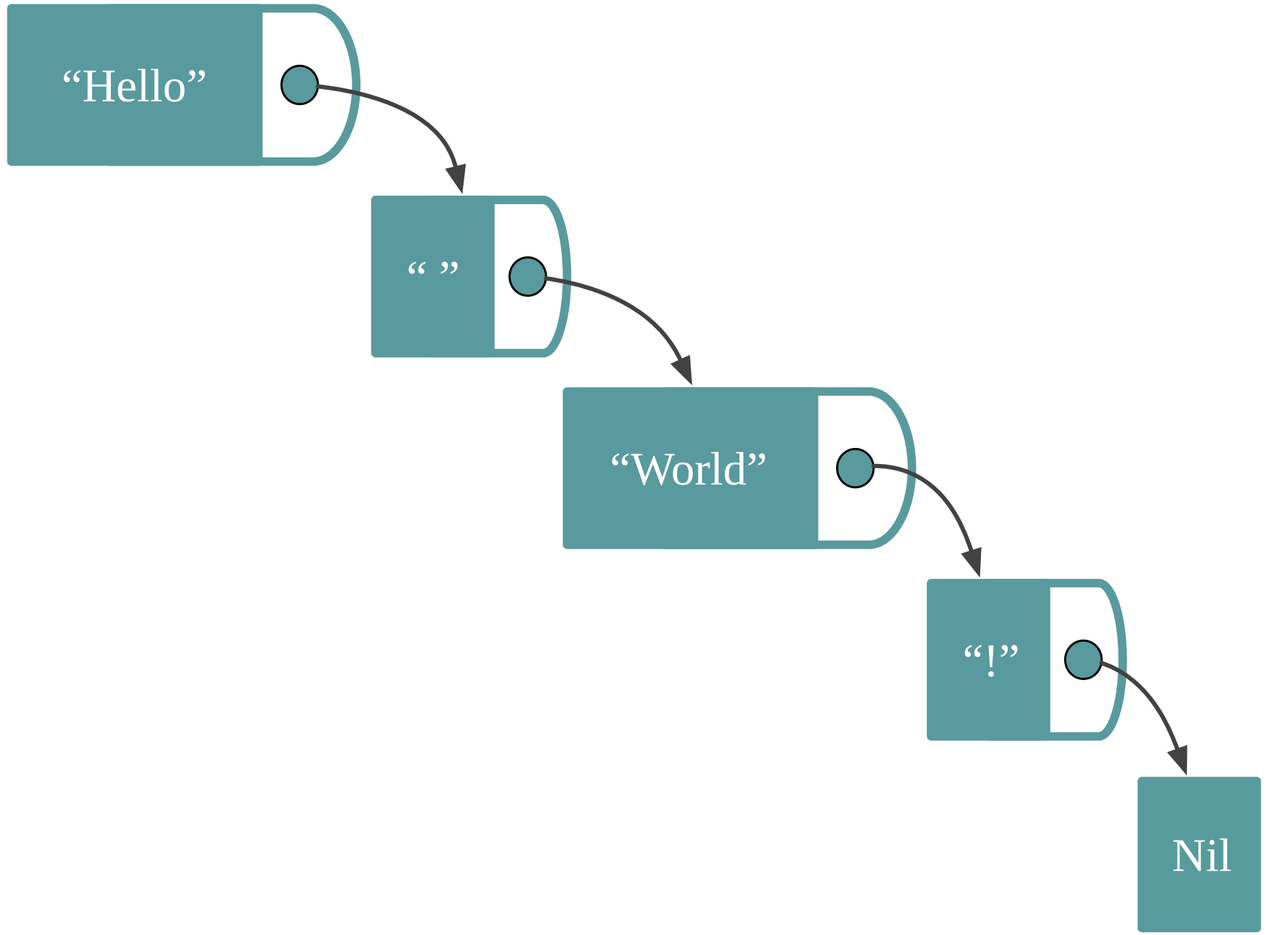
Array



Tree



Linked List



Iteration

---

background-image: url(img/function/fold.svg)

Folding

---

background-image: url(img/function/fold-left-1.svg)

FoldLeft

---

background-image: url(img/function/fold-left-all.svg)

FoldLeft