SCALA 101 - INTRODUCTION TRAINING

SESSION 2

SESSION 2

- Functional programming
- Higher order functions
- For-comprehensions
- More Scala collections

HOMEWORK - EXERCISE 4

- Questions/remarks
- Showcase solutions

EXCERCISE 4: TODO.TXT - POSSIBLE SOLUTION

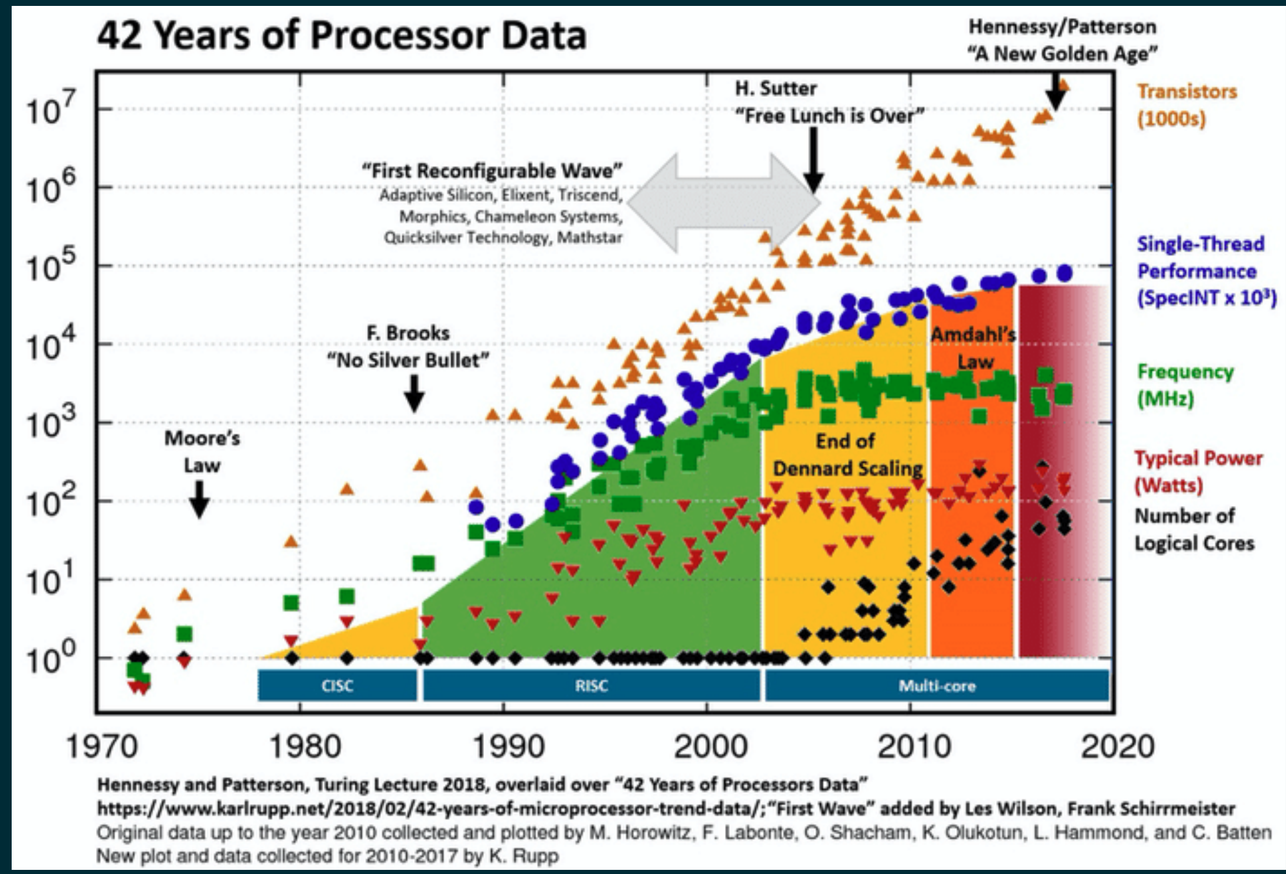
For example

```
import todo.TodoParser
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers

class TaskTest extends AnyFlatSpec with Matchers {
  it should "parse a simple task" in {
    val task = parse("(A) Call mom")

    task.isSuccess shouldBe true
    task.get.priority shouldBe Some("A")
    task.get.description shouldBe Some("Call mom")
  }

  private def parse(input: String) = {
    val parser = TodoParser(input)
    parser.task.run()
  }
}
```

MODULE 5 - FUNCTIONAL PROGRAMMING

WHY FUNCTIONAL PROGRAMMING?

- The free lunch is over
- Computers (CPU's) do not get faster anymore
- To speed up you need to scale
  - use more cores
  - use more servers
- To be able to scale you need to distribute your data
  - distributing *immutable* data is infinitely easier



OBJECT-ORIENTED VERSUS FUNCTIONAL PROGRAMMING

- OO versus FP makes no sense
  - They are *NOT* mutually exclusive

- OO can be combined with
  - imperative programming, like Java, C#, C++
  - functional programming, like Scala, F#, OCaml

IMPERATIVE VERSUS FUNCTIONAL

IMPERATIVE VERSUS FUNCTIONAL

- Imperative programming

IMPERATIVE VERSUS FUNCTIONAL

- Imperative programming
  - executing instructions and updating state

IMPERATIVE VERSUS FUNCTIONAL

- Imperative programming
  - executing instructions and updating state
  - first do X, then do Y

IMPERATIVE VERSUS FUNCTIONAL

- Imperative programming
  - executing instructions and updating state
  - first do X, then do Y
  - X and Y have side effects

IMPERATIVE VERSUS FUNCTIONAL

- Imperative programming
  - executing instructions and updating state
  - first do X, then do Y
  - X and Y have side effects
  - Z is updated

IMPERATIVE VERSUS FUNCTIONAL

- Imperative programming
  - executing instructions and updating state
  - first do X, then do Y
  - X and Y have side effects
  - Z is updated
- Functional programming

IMPERATIVE VERSUS FUNCTIONAL

- Imperative programming
  - executing instructions and updating state
  - first do X, then do Y
  - X and Y have side effects
  - Z is updated
- Functional programming
  - transforming data and calling functions

IMPERATIVE VERSUS FUNCTIONAL

- Imperative programming
  - executing instructions and updating state
  - first do X, then do Y
  - X and Y have side effects
  - Z is updated
- Functional programming
  - transforming data and calling functions
  - Call Y with the result of X

IMPERATIVE VERSUS FUNCTIONAL

- Imperative programming
  - executing instructions and updating state
  - first do X, then do Y
  - X and Y have side effects
  - Z is updated
- Functional programming
  - transforming data and calling functions
  - Call Y with the result of X
  - X and Y have no side effects

IMPERATIVE VERSUS FUNCTIONAL

- Imperative programming
  - executing instructions and updating state
  - first do X, then do Y
  - X and Y have side effects
  - Z is updated
- Functional programming
  - transforming data and calling functions
  - Call Y with the result of X
  - X and Y have no side effects
  - Z is immutable, transforming it will result in another data-object (Z')

REFERENTIAL TRANSPARENCY

An expression is said to be referentially transparent if it can be replaced with its value without changing the behaviour of the application

```
val a = 1 + 1

// is identical to

val a = 2
```

REFERENTIAL TRANSPARENCY

An expression is said to be referentially transparent if it can be replaced with its value without changing the behaviour of the application

```
val a = 1 + 1

// is identical to

val a = 2
```

This is true for any expression that only contains function calls to functions that have no side effects

`val` VERSUS `var`

Functional programming is about transforming *immutable* data

```
val i = 0
i = i + 1 // does not compile

var j = 0
j = j + 1 // OK
```

SINGLE COMPUTATION

SINGLE COMPUTATION

- An immutable value only needs to be computed once

SINGLE COMPUTATION

- An immutable value only needs to be computed once
  - Calculation happens when the value comes into scope

SINGLE COMPUTATION

- An immutable value only needs to be computed once
  - Calculation happens when the value comes into scope
- The `lazy` keyword indicates to the compiler that the evaluation should be done at the first actual use of the `val`

SINGLE COMPUTATION

- An immutable value only needs to be computed once
  - Calculation happens when the value comes into scope
- The `lazy` keyword indicates to the compiler that the evaluation should be done at the first actual use of the `val`
-
```
lazy val pi = veryExpensiveComputation
```

EVERYTHING IS AN EXPRESSION

```
val discount = if (totalPrice > 100) totalPrice * 0.05 else 0.0

val x: Unit = prinln("Hello world")
println(x)

// Hello world
// ()
```

EVERYTHING IS AN EXPRESSION

```scala
val discount = if (totalPrice > 100) totalPrice * 0.05 else 0.0

val x: Unit = prinln("Hello world")
println(x)

// Hello world
// ()
```

- Each statement in Scala can be treated as a value

EVERYTHING IS AN EXPRESSION

```scala
val discount = if (totalPrice > 100) totalPrice * 0.05 else 0.0

val x: Unit = prinln("Hello world")
println(x)

// Hello world
// ()
```

- Each statement in Scala can be treated as a value
- It has a type

EVERYTHING IS AN EXPRESSION

```scala
val discount = if (totalPrice > 100) totalPrice * 0.05 else 0.0

val x: Unit = prinln("Hello world")
println(x)

// Hello world
// ()
```

- Each statement in Scala can be treated as a value
- It has a type
  - Type can be `Unit`

EVERYTHING IS AN EXPRESSION

```scala
val discount = if (totalPrice > 100) totalPrice * 0.05 else 0.0

val x: Unit = prinln("Hello world")
println(x)

// Hello world
// ()
```

- Each statement in Scala can be treated as a value
- It has a type
  - Type can be `Unit`
- The last statement in each block (`{ ... }`) is treated as the value of that block

ITERATION

```
def sum(xs: List[Int]): Int = {
  var result = 0
  var i = 0
  while (i < xs.size) {
    result += xs(i)
    i += 1
  }
  result
}
```

RECURSION

```
def sum(xs: List[Int]): Int = {
  if (xs.isEmpty)
    0
  else
    xs.head + sum(xs.tail)
}
```

TAIL RECURSION

```
def sum(xs: List[Int]): Int = {
  @tailrec
  def go(ys: List[Int], result: Int = 0): Int =
    if (ys.isEmpty)
      result
    else
      go(ys.tail, result + ys.head)

  go(xs)
}
```

TAIL RECURSION

```
def sum(xs: List[Int]): Int = {
  @tailrec
  def go(ys: List[Int], result: Int = 0): Int =
    if (ys.isEmpty)
      result
    else
      go(ys.tail, result + ys.head)

  go(xs)
}
```

- Tail recursion is rewritten to a regular iteration under the hood. That makes it fast and stack-friendly.

TAIL RECURSION

```
def sum(xs: List[Int]): Int = {
  @tailrec
  def go(ys: List[Int], result: Int = 0): Int =
    if (ys.isEmpty)
      result
    else
      go(ys.tail, result + ys.head)

  go(xs)
}
```

- Tail recursion is rewritten to a regular iteration under the hood. That makes it fast and stack-friendly.
- Not all recursive functions can be rewritten into tail recursive functions.

TAIL RECURSION

```scala
def sum(xs: List[Int]): Int = {
  @tailrec
  def go(ys: List[Int], result: Int = 0): Int =
    if (ys.isEmpty)
      result
    else
      go(ys.tail, result + ys.head)

  go(xs)
}
```

- Tail recursion is rewritten to a regular iteration under the hood. That makes it fast and stack-friendly.
- Not all recursive functions can be rewritten into tail recursive functions.
- Truly tail-recursive functions are not that easy to write, look for more examples on http://alvinalexander.com/scala/scala-recursion-examples-recursive-programming

A NOTE ABOUT STYLE

- Always use the `@tailrec` annotation when you expect the function to be tail recursive
  - You get a compiler warning if it is not

PATTERN MATCHING

```
def intToString(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "many"
}
```

PATTERN MATCHING

```
def intToString(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "many"
}
```

- The `_` acts as the default clause

PATTERN MATCHING

```
def maybeIntToString(x: Option[Int]): String = x match {
  case None => "I don't know"
  case Some(i) => i.toString
}
```

PATTERN MATCHING

```
case class Person(firstName: String, lastName: String)

def isAwesome(p: Person): Boolean = p match {
  case Person("Martin", "Odersky") => true
  case Person(fn, _) if fn endsWith "an" => true
  case _ => false
}
```

PATTERN MATCHING

```
case class Person(firstName: String, lastName: String)

def isAwesome(p: Person): Boolean = p match {
  case Person("Martin", "Odersky") => true
  case Person(fn, _) if fn endsWith "an" => true
  case _ => false
}
```

- You can use "constants" in the Person pattern

PATTERN MATCHING

```scala
case class Person(firstName: String, lastName: String)

def isAwesome(p: Person): Boolean = p match {
  case Person("Martin", "Odersky") => true
  case Person(fn, _) if fn endsWith "an" => true
  case _ => false
}
```

- You can use "constants" in the Person pattern
- You can add guards to refine the match (`... if ...`)

PATTERN MATCHING

```
case class Person(firstName: String, lastName: String)

def isAwesome(p: Person): Boolean = p match {
  case Person("Martin", "Odersky") => true
  case Person(fn, _) if fn endsWith "an" => true
  case _ => false
}
```

- You can use "constants" in the Person pattern
- You can add guards to refine the match (`... if ...`)
- You can use `_` in the pattern for "don't care" parts (everything will match)

PATTERN MATCHING

```
case class Person(firstName: String, lastName: String)

def isAwesome(p: Person): Boolean = p match {
  case Person("Martin", "Odersky") => true
  case Person(fn, _) if fn endsWith "an" => true
  case _ => false
}
```

- You can use "constants" in the Person pattern
- You can add guards to refine the match (`... if ...`)
- You can use `_` in the pattern for "don't care" parts (everything will match)
- This works because case classes have an `unapply` method that allow you to decompose

PATTERN MATCHING

```
case class Person( firstName: String, lastName: String)

def areAwesome(team: List[Person]): Boolean = team match {
  case Nil => true
  case member :: Nil => isAwesome(member)
  case head :: tail => isAwesome(head) && areAwesome(tail)
}
```

PATTERN MATCHING

```
case class Person( firstName: String, lastName: String)

def areAwesome(team: List[Person]): Boolean = team match {
  case Nil => true
  case member :: Nil => isAwesome(member)
  case head :: tail => isAwesome(head) && areAwesome(tail)
}
```

- `Nil` is the empty list

PATTERN MATCHING

```
case class Person( firstName: String, lastName: String)

def areAwesome(team: List[Person]): Boolean = team match {
  case Nil => true
  case member :: Nil => isAwesome(member)
  case head :: tail => isAwesome(head) && areAwesome(tail)
}
```

- `Nil` is the empty list
- The `::` operator divides the list between the head element and the remainder (tail)

PATTERN MATCHING

```
case class Person( firstName: String, lastName: String)

def areAwesome(team: List[Person]): Boolean = team match {
  case Nil => true
  case member :: Nil => isAwesome(member)
  case head :: tail => isAwesome(head) && areAwesome(tail)
}
```

- `Nil` is the empty list
- The `::` operator divides the list between the head element and the remainder (tail)
- Order matters in pattern matching: first match wins

PATTERN MATCHING

```
case class Person( firstName: String, lastName: String)

def areAwesome(team: List[Person]): Boolean = team match {
  case Nil => true
  case member :: Nil => isAwesome(member)
  case head :: tail => isAwesome(head) && areAwesome(tail)
}
```

- `Nil` is the empty list
- The `::` operator divides the list between the head element and the remainder (tail)
- Order matters in pattern matching: first match wins
- The method is tail recursive

PATTERN MATCHING

```
case class Person( firstName: String, lastName: String)

def areAwesome(team: List[Person]): Boolean = team match {
  case Nil => true
  case member :: Nil => isAwesome(member)
  case head :: tail => isAwesome(head) && areAwesome(tail)
}
```

- `Nil` is the empty list
- The `::` operator divides the list between the head element and the remainder (tail)
- Order matters in pattern matching: first match wins
- The method is tail recursive
  - Possible because of Scala's short-circuiting of boolean expressions

LAMBDAS

```scala
def twice(n: Int): Int = n * 2

val twice: Int => Int = (n: Int) => n * 2

val twice: Int => Int = _ * 2
```

LAMBDAS

```scala
def twice(n: Int): Int = n * 2

val twice: Int => Int = (n: Int) => n * 2

val twice: Int => Int = _ * 2
```

LAMBDAS

```scala
def twice(n: Int): Int = n * 2

val twice: Int => Int = (n: Int) => n * 2

val twice: Int => Int = _ * 2
```

LAMBDAS

LAMBDAS

- The `Int ⇒ Int` is the type: a function from Int to Int

LAMBDAS

- The `Int ⇒ Int` is the type: a function from Int to Int
- The `(n: Int) ⇒ n * 2` is the function parameter list and function body

LAMBDAS

- The `Int ⇒ Int` is the type: a function from Int to Int
- The `(n: Int) ⇒ n * 2` is the function parameter list and function body
- The `_` is used as placeholder for the first parameter

LAMBDAS

- The `Int ⇒ Int` is the type: a function from Int to Int
- The `(n: Int) ⇒ n * 2` is the function parameter list and function body
- The `_` is used as placeholder for the first parameter
- When functions have multiple parameters, you can use multiple `_`

LAMBDAS

- The `Int ⇒ Int` is the type: a function from Int to Int
- The `(n: Int) ⇒ n * 2` is the function parameter list and function body
- The `_` is used as placeholder for the first parameter
- When functions have multiple parameters, you can use multiple `_`
  - Be careful to keep it legible, the `(a: Int, b: Int) ⇒ ???` syntax is usually preferred

TYPE ALIAS

TYPE ALIAS

- Useful when you define lambdas

TYPE ALIAS

- Useful when you define lambdas
  - `type Twice = Int ⇒ Int`

TYPE ALIAS

- Useful when you define lambdas
  - `type Twice = Int ⇒ Int`
  - `val twice: Twice = _ * 2`

TYPE ALIAS

- Useful when you define lambdas
  - `type Twice = Int ⇒ Int`
  - `val twice: Twice = _ * 2`
- But also for very long or complex types

TYPE ALIAS

- Useful when you define lambdas
  - `type Twice = Int ⇒ Int`
  - `val twice: Twice = _ * 2`
- But also for very long or complex types
  - `type MultiMap[K, V] = HashMap[K, List[V]]`

EXERCISE 5: PASCAL'S TRIANGLE

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

EXERCISE 5: PASCAL'S TRIANGLE

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

- This pattern of numbers is called *Pascal's Triangle*

EXERCISE 5: PASCAL'S TRIANGLE

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

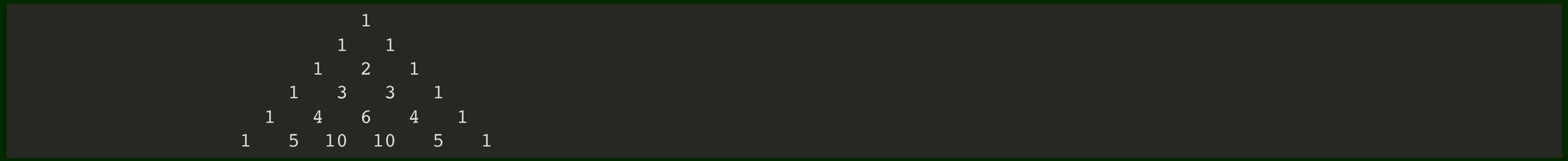
- This pattern of numbers is called *Pascal's Triangle*
- The numbers on the 2 sides are always 1

EXERCISE 5: PASCAL'S TRIANGLE

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

- This pattern of numbers is called *Pascal's Triangle*
- The numbers on the 2 sides are always 1
- Each number inside the triangle is the sum of the numbers above it

EXERCISE 5: PASCAL'S TRIANGLE



```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

- This pattern of numbers is called *Pascal's Triangle*
- The numbers on the 2 sides are always 1
- Each number inside the triangle is the sum of the numbers above it
- Useful because they give the factors of

$$(a+b)^p$$

EXERCISE 5: PASCAL'S TRIANGLE

EXERCISE 5: PASCAL'S TRIANGLE

- p = 0:

$$(a+b)^0 = 1$$

EXERCISE 5: PASCAL'S TRIANGLE

- p = 0:

$$(a+b)^0 = 1$$

- p = 1:

$$(a+b)^1 = 1a + 1b$$

EXERCISE 5: PASCAL'S TRIANGLE

- p = 0:

$$(a+b)^0 = 1$$

- p = 1:

$$(a+b)^1 = 1a + 1b$$

- p = 2:

$$(a+b)^2 = 1a^2 + 2ab + 1b^2$$

EXERCISE 5: PASCAL'S TRIANGLE

- p = 0:

$$(a+b)^0 = 1$$

- p = 1:

$$(a+b)^1 = 1a + 1b$$

- p = 2:

$$(a+b)^2 = 1a^2 + 2ab + 1b^2$$

- p = 3:

$$(a+b)^3 = 1a^3 + 3a^2b + 3ab^2 + 1b^3$$

EXERCISE 5: PASCAL'S TRIANGLE

- p = 0:

$$(a+b)^0 = 1$$

- p = 1:

$$(a+b)^1 = 1a + 1b$$

- p = 2:

$$(a+b)^2 = 1a^2 + 2ab + 1b^2$$

- p = 3:

$$(a+b)^3 = 1a^3 + 3a^2b + 3ab^2 + 1b^3$$

- ...

EXERCISE 5: PASCAL'S TRIANGLE

- Implement the `pascal` function, which takes a column `c` and a row `r` (counting from 0)
- The function returns the number at that spot in the triangle
- For example
  - pascal(0, 2) = 1
  - pascal(1, 2) = 2
  - pascal(1, 3) = 3

EXERCISE 5: PASCAL'S TRIANGLE

- Implement the `pascal` function, which takes a column `c` and a row `r` (counting from 0)
- The function returns the number at that spot in the triangle
- For example
  - pascal(0, 2) = 1
  - pascal(1, 2) = 2
  - pascal(1, 3) = 3

Open `exercises/src/main/scala/exercise5/Pascal.scala` and add/update the code to make the program work as expected.

Exercises available at https://github.com/code-star/scala101-full-course.git

EXERCISE 6: SETS OF INTEGERS

EXERCISE 6: SETS OF INTEGERS

- How could we represent sets of integers

EXERCISE 6: SETS OF INTEGERS

- How could we represent sets of integers
  - For example: the set with all negative integers

EXERCISE 6: SETS OF INTEGERS

- How could we represent sets of integers
  - For example: the set with all negative integers
  - infinite number of elements -> we cannot list them

EXERCISE 6: SETS OF INTEGERS

- How could we represent sets of integers
  - For example: the set with all negative integers
  - infinite number of elements -> we cannot list them
- One possible way:

EXERCISE 6: SETS OF INTEGERS

- How could we represent sets of integers
  - For example: the set with all negative integers
  - infinite number of elements -> we cannot list them
- One possible way:
  - If you give me an integer, I'll say if it's in the set

EXERCISE 6: SETS OF INTEGERS

- How could we represent sets of integers
  - For example: the set with all negative integers
  - infinite number of elements -> we cannot list them
- One possible way:
  - If you give me an integer, I'll say if it's in the set
  - For 3, I'll say "no"

EXERCISE 6: SETS OF INTEGERS

- How could we represent sets of integers
  - For example: the set with all negative integers
  - infinite number of elements -> we cannot list them
- One possible way:
  - If you give me an integer, I'll say if it's in the set
  - For 3, I'll say "no"
  - For -4, I'll say "yes"

EXERCISE 6: SETS OF INTEGERS

- How could we represent sets of integers
  - For example: the set with all negative integers
  - infinite number of elements -> we cannot list them
- One possible way:
  - If you give me an integer, I'll say if it's in the set
  - For 3, I'll say "no"
  - For -4, I'll say "yes"
- We make a function for that

EXERCISE 6: SETS OF INTEGERS

EXERCISE 6: SETS OF INTEGERS

- We define "characteristic functions" for sets

EXERCISE 6: SETS OF INTEGERS

- We define "characteristic functions" for sets
- All negative integers

EXERCISE 6: SETS OF INTEGERS

- We define "characteristic functions" for sets
- All negative integers
  - `val negative: Int ⇒ Boolean = (x: Int) ⇒ x < 0`

EXERCISE 6: SETS OF INTEGERS

- We define "characteristic functions" for sets
- All negative integers
  - `val negative: Int ⇒ Boolean = (x: Int) ⇒ x < 0`
- All even integers

EXERCISE 6: SETS OF INTEGERS

- We define "characteristic functions" for sets
- All negative integers
  - `val negative: Int ⇒ Boolean = (x: Int) ⇒ x < 0`
- All even integers
  - `val even: Int ⇒ Boolean = (x: Int) ⇒ x % 2 == 0`

EXERCISE 6: SETS OF INTEGERS

```
val negative: Int => Boolean =
  (x: Int) => x < 0

val even: Int => Boolean =
  (x: Int) => x % 2 == 0
```

EXERCISE 6: SETS OF INTEGERS

```
val negative: Int => Boolean =
  (x: Int) => x < 0

val even: Int => Boolean =
  (x: Int) => x % 2 == 0
```

```
type Set = Int => Boolean

val negative: Set =
  (x: Int) => x < 0

val even: Set =
  (x: Int) => x % 2 == 0
```

EXERCISE 6: SETS OF INTEGERS

```
val negative: Int => Boolean =
  (x: Int) => x < 0

val even: Int => Boolean =
  (x: Int) => x % 2 == 0
```

```
type Set = Int => Boolean

val negative: Set =
  (x: Int) => x < 0

val even: Set =
  (x: Int) => x % 2 == 0
```

```
// a function that tests for the presence of a value in a set
def contains(s: Set, elem: Int): Boolean = s(elem)
```

EXERCISE 6: SETS OF INTEGERS

- We need more functions to work effectively with sets
- Your task is to define them
- Hint: return lambdas

EXERCISE 6: SETS OF INTEGERS

- We need more functions to work effectively with sets
- Your task is to define them
- Hint: return lambdas

Open `exercises/src/main/scala/exercise6/Sets.scala` and add/update the code to make the program work as expected.

Exercises available at https://github.com/code-star/scala101-full-course.git

EXERCISE 6: SETS OF INTEGERS

- We need more functions to work effectively with sets
- Your task is to define them
- Hint: return lambdas

Open `exercises/src/main/scala/exercise6/Sets.scala` and add/update the code to make the program work as expected.

Exercises available at https://github.com/code-star/scala101-full-course.git

```
def singletonSet(elem: Int): Set = ???    // set with only 1 element

def union(s: Set, t: Set): Set = ???      // set with all elements from s and t

def intersect(s: Set, t: Set): Set = ???  // set with elements that are in s and in t

def diff(s: Set, t: Set): Set = ???       // set with elements from s that are not in t
```

MODULE 6 - HIGHER ORDER FUNCTIONS

HIGHER ORDER FUNCTIONS

```
1 def applyFunction(n: Int, f: Int => Int): Int = f(n)
2
3 applyFunction(2, n => n + 1)                // returns 3
4
5 applyFunction(2, _ + 1)                    // also returns 3
6
7 applyFunction(3, twice)                    // returns 6
```

HIGHER ORDER FUNCTIONS

```
1 def applyFunction(n: Int, f: Int => Int): Int = f(n)
2
3 applyFunction(2, n => n + 1)                // returns 3
4
5 applyFunction(2, _ + 1)                    // also returns 3
6
7 applyFunction(3, twice)                    // returns 6
```

HIGHER ORDER FUNCTIONS

```
1 def applyFunction(n: Int, f: Int => Int): Int = f(n)
2
3 applyFunction(2, n => n + 1)                // returns 3
4
5 applyFunction(2, _ + 1)                    // also returns 3
6
7 applyFunction(3, twice)                    // returns 6
```

HIGHER ORDER FUNCTIONS

```
1 def applyFunction(n: Int, f: Int => Int): Int = f(n)
2
3 applyFunction(2, n => n + 1)                // returns 3
4
5 applyFunction(2, _ + 1)                    // also returns 3
6
7 applyFunction(3, twice)                    // returns 6
```

HIGHER ORDER FUNCTIONS - COLLECTIONS

Collections have many higher order functions that allow you to perform operations on their elements

```
1 val languages = List("Scala", "Java", "C#", "Python")
2
3 languages.filter(s => s.contains("a"))      // List(Scala, Java)
4
5 languages.map(_.toUpperCase)                // List(SCALA, JAVA, C#, PYTHON)
6
7 languages.sortBy(_.length)                  // List(C#, Java, Scala, Python)
```

HIGHER ORDER FUNCTIONS - COLLECTIONS

Collections have many higher order functions that allow you to perform operations on their elements

```
1 val languages = List("Scala", "Java", "C#", "Python")
2
3 languages.filter(s => s.contains("a"))      // List(Scala, Java)
4
5 languages.map(_.toUpperCase)                // List(SCALA, JAVA, C#, PYTHON)
6
7 languages.sortBy(_.length)                  // List(C#, Java, Scala, Python)
```

HIGHER ORDER FUNCTIONS - COLLECTIONS

Collections have many higher order functions that allow you to perform operations on their elements

```scala
val languages = List("Scala", "Java", "C#", "Python")

languages.filter(s => s.contains("a"))      // List(Scala, Java)

languages.map(_.toUpperCase)                // List(SCALA, JAVA, C#, PYTHON)

languages.sortBy(_.length)                  // List(C#, Java, Scala, Python)
```

HIGHER ORDER FUNCTIONS - COLLECTIONS

Collections have many higher order functions that allow you to perform operations on their elements

```
1 val languages = List("Scala", "Java", "C#", "Python")
2
3 languages.filter(s => s.contains("a"))      // List(Scala, Java)
4
5 languages.map(_.toUpperCase)                // List(SCALA, JAVA, C#, PYTHON)
6
7 languages.sortBy(_.length)                  // List(C#, Java, Scala, Python)
```

HIGHER ORDER FUNCTIONS - COLLECTIONS

Alternative notations

```
1 val languages = List("Scala", "Java", "C#", "Python")
2
3 languages filter { s => s.contains("a") }    // List(Scala, Java)
4
5 languages map { _.toUpperCase }              // List(SCALA, JAVA, C#, PYTHON)
6
7 languages sortBy { _.length }                // List(C#, Java, Scala, Python)
```

HIGHER ORDER FUNCTIONS - COLLECTIONS

Alternative notations

```
1 val languages = List("Scala", "Java", "C#", "Python")
2
3 languages filter { s => s.contains("a") }   // List(Scala, Java)
4
5 languages map { _.toUpperCase }             // List(SCALA, JAVA, C#, PYTHON)
6
7 languages sortBy { _.length }               // List(C#, Java, Scala, Python)
```

HIGHER ORDER FUNCTIONS - COLLECTIONS

Alternative notations

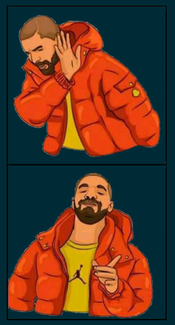
```
1 val languages = List("Scala", "Java", "C#", "Python")
2
3 languages filter { s => s.contains("a") }    // List(Scala, Java)
4
5 languages map { _.toUpperCase }              // List(SCALA, JAVA, C#, PYTHON)
6
7 languages sortBy { _.length }                // List(C#, Java, Scala, Python)
```

HIGHER ORDER FUNCTIONS - COLLECTIONS

Alternative notations

```
1 val languages = List("Scala", "Java", "C#", "Python")
2
3 languages filter { s => s.contains("a") }    // List(Scala, Java)
4
5 languages map { _.toUpperCase }              // List(SCALA, JAVA, C#, PYTHON)
6
7 languages sortBy { _.length }                // List(C#, Java, Scala, Python)
```

A NOTE ABOUT STYLE



A NOTE ABOUT STYLE

- Method call notation

A NOTE ABOUT STYLE

- Method call notation
  - `languages.filter(_.contains("a")).map(_.toUpperCase)`

A NOTE ABOUT STYLE

- Method call notation
  - `languages.filter(_.contains("a")).map(_.toUpperCase)`
- Infix notation

A NOTE ABOUT STYLE

- Method call notation
  - `languages.filter(_.contains("a")).map(_.toUpperCase)`
- Infix notation
  - `languages filter { _.contains("a") } map { _.toUpperCase }`

A NOTE ABOUT STYLE

- Method call notation
  - `languages.filter(_.contains("a")).map(_.toUpperCase)`
- Infix notation
  - `languages filter { _.contains("a") } map { _.toUpperCase }`
- Do not mix the notations, that makes it confusing

HIGHER ORDER COLLECTION FUNCTIONS

```
1  val xs = List(1, 2, 3, 4, 5)
2
3  xs takeWhile { n => n < 3 }                  // List(1, 2)
4  xs dropWhile { n => n < 3 }                  // List(3, 4, 5)
5  xs find { n => n > 3}                        // Some(4)
6
7  xs exists { n => n > 10 }                    // false
8  xs forall { n => n < 10 }                    // true
9
10 xs maxBy { n => -n }                         // -1
11 xs minBy { n => -n }                         // -5
12
13 xs sortWith { (n, m) => n > m }              // List(5, 4, 3, 2, 1)
14 xs groupBy { n => n % 2 }                    // Map(0 -> List(2,4), 1 -> List(1,3,5))
```

HIGHER ORDER COLLECTION FUNCTIONS

```
1  val xs = List(1, 2, 3, 4, 5)
2
3  xs takeWhile { n => n < 3 }                  // List(1, 2)
4  xs dropWhile { n => n < 3 }                  // List(3, 4, 5)
5  xs find { n => n > 3}                        // Some(4)
6
7  xs exists { n => n > 10 }                    // false
8  xs forall { n => n < 10 }                    // true
9
10 xs maxBy { n => -n }                         // -1
11 xs minBy { n => -n }                         // -5
12
13 xs sortWith { (n, m) => n > m }              // List(5, 4, 3, 2, 1)
14 xs groupBy { n => n % 2 }                    // Map(0 -> List(2,4), 1 -> List(1,3,5))
```

HIGHER ORDER COLLECTION FUNCTIONS

```scala
val xs = List(1, 2, 3, 4, 5)

xs takeWhile { n => n < 3 }                  // List(1, 2)
xs dropWhile { n => n < 3 }                  // List(3, 4, 5)
xs find { n => n > 3}                        // Some(4)

xs exists { n => n > 10 }                    // false
xs forall { n => n < 10 }                    // true

xs maxBy { n => -n }                         // -1
xs minBy { n => -n }                         // -5

xs sortWith { (n, m) => n > m }              // List(5, 4, 3, 2, 1)
xs groupBy { n => n % 2 }                    // Map(0 -> List(2,4), 1 -> List(1,3,5))
```

HIGHER ORDER COLLECTION FUNCTIONS

```scala
val xs = List(1, 2, 3, 4, 5)

xs takeWhile { n => n < 3 }                 // List(1, 2)
xs dropWhile { n => n < 3 }                 // List(3, 4, 5)
xs find { n => n > 3}                       // Some(4)

xs exists { n => n > 10 }                   // false
xs forall { n => n < 10 }                   // true

xs maxBy { n => -n }                        // -1
xs minBy { n => -n }                        // -5

xs sortWith { (n, m) => n > m }             // List(5, 4, 3, 2, 1)
xs groupBy { n => n % 2 }                   // Map(0 -> List(2,4), 1 -> List(1,3,5))
```

HIGHER ORDER COLLECTION FUNCTIONS

```
1  val xs = List(1, 2, 3, 4, 5)
2
3  xs takeWhile { n => n < 3 }                  // List(1, 2)
4  xs dropWhile { n => n < 3 }                  // List(3, 4, 5)
5  xs find { n => n > 3}                        // Some(4)
6
7  xs exists { n => n > 10 }                    // false
8  xs forall { n => n < 10 }                    // true
9
10 xs maxBy { n => -n }                         // -1
11 xs minBy { n => -n }                         // -5
12
13 xs sortWith { (n, m) => n > m }              // List(5, 4, 3, 2, 1)
14 xs groupBy { n => n % 2 }                    // Map(0 -> List(2,4), 1 -> List(1,3,5))
```

HIGHER ORDER COLLECTION FUNCTIONS

```scala
val xs = List(1, 2, 3, 4, 5)

xs.reduce((acc: Int, cur: Int) => acc + cur)          // 15

xs.foldLeft(5) { (acc: Int, cur: Int) => acc + cur}   // 20

xs.foldLeft("") { (acc: Int, cur: Int) =>
    acc + cur.toString }                              // "12345"
```

- acc = accumulator
- cur = current element

HIGHER ORDER COLLECTION FUNCTIONS

```
1 val xs = List(1, 2, 3, 4, 5)
2
3 xs.reduce((acc: Int, cur: Int) => acc + cur)          // 15
4
5 xs.foldLeft(5) { (acc: Int, cur: Int) => acc + cur}   // 20
6
7 xs.foldLeft("") { (acc: Int, cur: Int) =>
8     acc + cur.toString }                              // "12345"
```

- acc = accumulator
- cur = current element

HIGHER ORDER COLLECTION FUNCTIONS

```
1 val xs = List(1, 2, 3, 4, 5)
2
3 xs.reduce((acc: Int, cur: Int) => acc + cur)          // 15
4
5 xs.foldLeft(5) { (acc: Int, cur: Int) => acc + cur}   // 20
6
7 xs.foldLeft("") { (acc: Int, cur: Int) =>
8     acc + cur.toString }                              // "12345"
```

- acc = accumulator
- cur = current element

HIGHER ORDER COLLECTION FUNCTIONS

```scala
val xs = List(1, 2, 3, 4, 5)

xs.reduce((acc: Int, cur: Int) => acc + cur)          // 15

xs.foldLeft(5) { (acc: Int, cur: Int) => acc + cur}   // 20

xs.foldLeft("") { (acc: Int, cur: Int) =>
    acc + cur.toString }                              // "12345"
```

- acc = accumulator
- cur = current element

HIGHER ORDER COLLECTION FUNCTIONS

```
val xs = List(1, 2, 3, 4, 5)

xs.reduce((acc, cur) => acc + cur)                    // 15

xs.foldLeft(5) { (acc, cur) => acc + cur}             // 20

xs.foldLeft("") { (acc, cur) =>
    acc + cur.toString }                              // "12345"
```

- types are inferred by compiler, no need to specify them

HIGHER ORDER COLLECTION FUNCTIONS

```
val xs = List(1, 2, 3, 4, 5)

xs.reduce(_ + _)                                    // 15

xs.foldLeft(5) { _ + _ }                            // 20

xs.foldLeft("") { _ + _.toString }                  // "12345"
```

- using each parameter exactly once, no need to name them

HIGHER ORDER COLLECTION FUNCTIONS: FLATMAP

```
1 val xs = List(1, 2, 3, 4, 5)
2
3 val ys = xs.map(el => List(el, el))                    // List(List(1,1),List(2,2),List(3,3),List(4,4),List(5,5))
4
5 ys.flatten                                             // List(1,1,2,2,3,3,4,4,5,5)
6
7 xs.flatMap(el => List(el, el))                         // List(1,1,2,2,3,3,4,4,5,5)
```

HIGHER ORDER COLLECTION FUNCTIONS: FLATMAP

```
1 val xs = List(1, 2, 3, 4, 5)
2
3 val ys = xs.map(el => List(el, el))                    // List(List(1,1),List(2,2),List(3,3),List(4,4),List(5,5))
4
5 ys.flatten                                             // List(1,1,2,2,3,3,4,4,5,5)
6
7 xs.flatMap(el => List(el, el))                         // List(1,1,2,2,3,3,4,4,5,5)
```

HIGHER ORDER COLLECTION FUNCTIONS: FLATMAP

```
1 val xs = List(1, 2, 3, 4, 5)
2
3 val ys = xs.map(el => List(el, el))                    // List(List(1,1),List(2,2),List(3,3),List(4,4),List(5,5))
4
5 ys.flatten                                             // List(1,1,2,2,3,3,4,4,5,5)
6
7 xs.flatMap(el => List(el, el))                         // List(1,1,2,2,3,3,4,4,5,5)
```

HIGHER ORDER COLLECTION FUNCTIONS: FLATMAP

```
1 val xs = List(1, 2, 3, 4, 5)
2
3 def even(n: Int): Option[Int] =
4     if (n % 2 == 0) Some(n) else None
5
6 xs map even                                    // List(None, Some(2), None, Some(4), None)
7
8 xs flatMap even                                // List(2, 4)
```

HIGHER ORDER COLLECTION FUNCTIONS: FLATMAP

```scala
val xs = List(1, 2, 3, 4, 5)

def even(n: Int): Option[Int] =
    if (n % 2 == 0) Some(n) else None

xs map even                                     // List(None, Some(2), None, Some(4), None)

xs flatMap even                                 // List(2, 4)
```

HIGHER ORDER COLLECTION FUNCTIONS: FLATMAP

```
1 val xs = List(1, 2, 3, 4, 5)
2
3 def even(n: Int): Option[Int] =
4     if (n % 2 == 0) Some(n) else None
5
6 xs map even                                    // List(None, Some(2), None, Some(4), None)
7
8 xs flatMap even                                // List(2, 4)
```

EXERCISE 7: TODO.TXT, PART II

EXERCISE 7: TODO.TXT, PART II

- Revisit the todo.txt exercise

EXERCISE 7: TODO.TXT, PART II

- Revisit the todo.txt exercise
- How would you

EXERCISE 7: TODO.TXT, PART II

- Revisit the todo.txt exercise
- How would you
  - Get all tasks in project GarageSale?

EXERCISE 7: TODO.TXT, PART II

- Revisit the todo.txt exercise
- How would you
  - Get all tasks in project GarageSale?
  - Get all priority A phone calls?

EXERCISE 7: TODO.TXT, PART II

- Revisit the todo.txt exercise
- How would you
  - Get all tasks in project GarageSale?
  - Get all priority A phone calls?
  - Turn all priority A tasks into priority B tasks?

MODULE 7 - FOR COMPREHENSIONS

FOR COMPREHENSIONS - SIDE EFFECTS

```
for (i <- 1 to 10) println(i)
```

FOR COMPREHENSIONS - SIDE EFFECTS

```
for (i <- 1 to 10) println(i)

// is equivalent to
(1 to 10) foreach { i => println(i) }

// is equivalent to
(1 to 10) foreach { println(_) }

// is equivalent to
1 to 3 foreach println
```

FOR COMPREHENSIONS - FUNCTIONAL

```
val xs = List(1, 2, 3)

val ys = for{
  x <- xs
} yield (x + 1)

// is equivalent to
val ys = xs.map(_ + 1)
```

FOR COMPREHENSION - HOW IT WORKS

Simplified general form

```
for {
  p1 <- g1
    ...
  pn <- gn
} yield result(p1, ..., pn)
```

FOR COMPREHENSION - HOW IT WORKS

Simplified general form

```
for {
  p1 <- g1
    ...
  pn <- gn
} yield result(p1, ..., pn)
```

- the for comprehension is a sequence of generators and filters

FOR COMPREHENSION - HOW IT WORKS

Simplified general form

```
for {
  p1 <- g1
    ...
  pn <- gn
} yield result(p1, ..., pn)
```

- the for comprehension is a sequence of generators and filters
- `p1 ← g1` is a generator

FOR COMPREHENSION - HOW IT WORKS

Simplified general form

```
for {
  p1 <- g1
    ...
  pn <- gn
} yield result(p1, ..., pn)
```

- the for comprehension is a sequence of generators and filters
- `p1 ← g1` is a generator
- if there are several generators

FOR COMPREHENSION - HOW IT WORKS

Simplified general form

```
for {
  p1 <- g1
    ...
  pn <- gn
} yield result(p1, ..., pn)
```

- the for comprehension is a sequence of generators and filters
- `p1 ← g1` is a generator
- if there are several generators
  - equivalent of a nested loop

FOR COMPREHENSION - HOW IT WORKS

Simplified general form

```
for {
  p1 <- g1
    ...
  pn <- gn
} yield result(p1, ..., pn)
```

- the for comprehension is a sequence of generators and filters
- `p1 ← g1` is a generator
- if there are several generators
  - equivalent of a nested loop
  - the last generator varies faster than the first

FOR COMPREHENSION - HOW IT WORKS

Simplified general form

```
for {
  p1 <- g1
    ...
  pn <- gn
} yield result(p1, ..., pn)
```

- the for comprehension is a sequence of generators and filters
- `p1 ← g1` is a generator
- if there are several generators
  - equivalent of a nested loop
  - the last generator varies faster than the first
  - `result(p1, ... , pn)` computes 1 element of the resulting collection

FOR COMPREHENSION - HOW IT WORKS

Complete general form

```
for {
  p1 <- g1 if f1(p1)
    ...
  pn <- gn if fn(pn)
} yield result(p1, ..., pn)
```

FOR COMPREHENSION - HOW IT WORKS

Complete general form

```
for {
  p1 <- g1 if f1(p1)
    ...
  pn <- gn if fn(pn)
} yield result(p1, ..., pn)
```

- `if f1(p1)` is a filter

FOR COMPREHENSION - HOW IT WORKS

Complete general form

```
for {
  p1 <- g1 if f1(p1)
    ...
  pn <- gn if fn(pn)
} yield result(p1, ..., pn)
```

- `if f1(p1)` is a filter
- A value is produced by the generator only if the filter evaluates to `true`

FOR COMPREHENSION - TRANSLATION RULES

The for comprehension is called "syntactic sugar", because it is converted into other Scala constructs before compiling

```
for (x <- e1) yield e2

// is translated to

e1.map(x=> e2)
```

FOR COMPREHENSION - TRANSLATION RULES

The for comprehension is called "syntactic sugar", because it is converted into other Scala constructs before compiling

```
for (x <- e1) yield e2

// is translated to

e1.map(x=> e2)
```

- A for-comprehension looks like a traditional for-loop, but works differently

FOR COMPREHENSION - TRANSLATION RULES

The for comprehension is called "syntactic sugar", because it is converted into other Scala constructs before compiling

```
for (x <- e1) yield e2

// is translated to

e1.map(x=> e2)
```

- A for-comprehension looks like a traditional for-loop, but works differently
- You can use it with your own types, as long as they provide `map`, `flatMap`, and `filter`

FOR COMPREHENSION - TRANSLATION RULES

The for comprehension is called "syntactic sugar", because it is converted into other Scala constructs before compiling

```
for (x <- e1) yield e2

// is translated to

e1.map(x=> e2)
```

- A for-comprehension looks like a traditional for-loop, but works differently
- You can use it with your own types, as long as they provide `map`, `flatMap`, and `filter`
  - With the proper behaviour that is expected from these methods

FOR COMPREHENSION - TRANSLATION RULES

The for comprehension is called "syntactic sugar", because it is converted into other Scala constructs before compiling

```
for (x <- e1 if f) yield e2

// is translated to

for (x <- e1.filter(x => f)) yield e2

// is translated to
(e1.filter(x => f)).map(x => e2)
```

FOR COMPREHENSION - TRANSLATION RULES

The for comprehension is called "syntactic sugar", because it is converted into other Scala constructs before compiling

```
for (x <- e1; y <- e2) yield e3

// is translated to

e1.flatMap(x => for (y <- e2) yield e3)
```

FOR COMPREHENSION - TRANSLATION RULES

The for comprehension is called "syntactic sugar", because it is converted into other Scala constructs before compiling

```
for (x <- e1; y <- e2) yield e3

// is translated to

e1.flatMap(x => for (y <- e2) yield e3)
```

- and then the inner `for` is translated again...

FOR COMPREHENSION - TRANSLATION RULES

Listing all combinations of numbers x and y, where x is drawn from 1 to m, and y is drawn from 1 to n

```
for (x <- 1 to m; y <- 1 to n) yield (x, y)

// is equivalent to

(1 to m).flatMap(x => (1 to n) map (y => (x, y)))
```

EXERCISE 8: SHAKESPEARE, PART II

Implement the functions marked `???` in `functions.scala`

EXERCISE 8: SHAKESPEARE, PART II

Implement the functions marked `???` in `functions.scala`

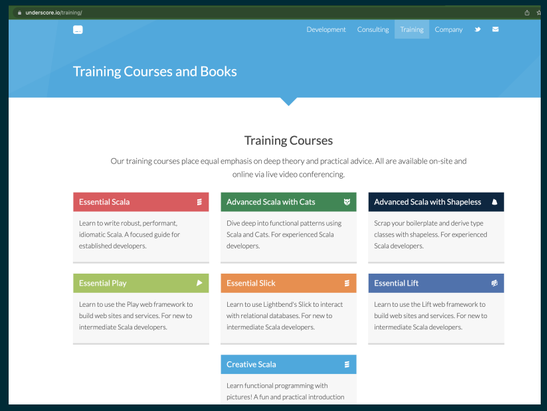
Open `appHamlet/src/main/scala/words/fuinctions.scala` and add/update the code to make the program work as expected.

Exercises available at https://github.com/code-star/scala101-full-course.git

WRAP UP

RESOURCES FOR FURTHER STUDY

- Twitter's Scala School
  - http://twitter.github.io/scala_school/
- Free online courses at Coursera
  - https://www.coursera.org/specializations/scala
- Book: Functional Programming in Scala
  - https://www.manning.com/books/functional-programming-in-scala
- Udemy training
  - https://www.udemy.com/course/rock-the-jvm-scala-for-beginners/
- Underscore.io - free eBooks
  - https://underscore.io/training/



THE END

QUESTIONS?