λ

Introduction to Scala

Programmazione Funzionale

2023/2024

Università di Trento

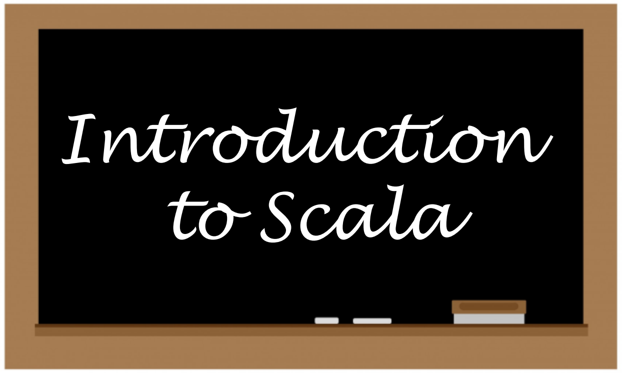
Chiara Di Francescomarino

Next lectures

- Thursday May 23: exam simulation
- Tuesday May 28: last lecture
  - We will have a lab class: bring your laptop
  - We will see results and a solution of the ML mini-challenge

Today

- Introduction to Scala
- The basics of the language
- Data Types
- Control structures
- OOP Domain modeling



Introduction to Scala

A bit of history

- The design of Scala started in 2011 at the EPFL (École Polytechnique Fédérale de Lausanne) by Martin Odersky
- He decided to take some ideas from functional programming and moving them into the Java space.
- The result was in 1996 a language called Pizza.
- It was quite successful: functional language features can be implemented on a JVM platform.

After Pizza

- Martin got in contact with people from Sun interested in generics
- In 1997/1998 Generic Java (GJ) was born, which then became the generics in Java 5.

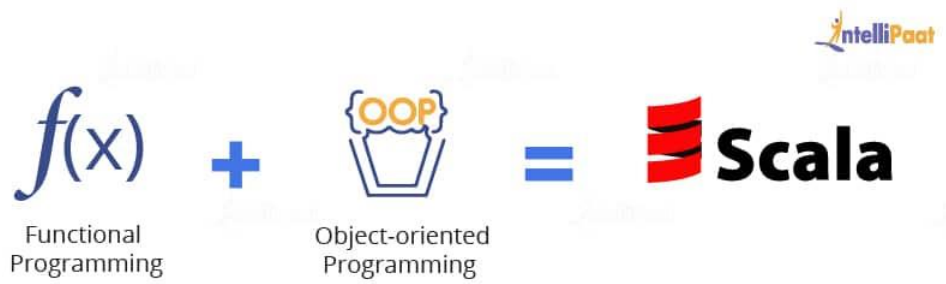
- May 2011 Typesafe Inc.
- Currently named Lightbend Inc.

Why a new programming language?

- Scalable Language
  - Designed to grow with the users' demand
- Multi-paradigm Language
  - Combining object-oriented and functional programming

Scala in few words

- A combination of functional and object-oriented programming



FP is good at isolating state change
Immutability, repeatability, concurrency

OOP is good at structuring code
Interfaces, classes, encapsulation, delegation, singleton

Multi-paradigm

Object-oriented paradigm

- Every value is an object
- Types and behaviour of objects are described by classes
- Mechanisms of class abstractions

Functional paradigm

- Every function is a value (including methods)
- Lightweight syntax for anonymous functions, higher-order functions, nested functions, currying
- Pattern matching

Why Scala?

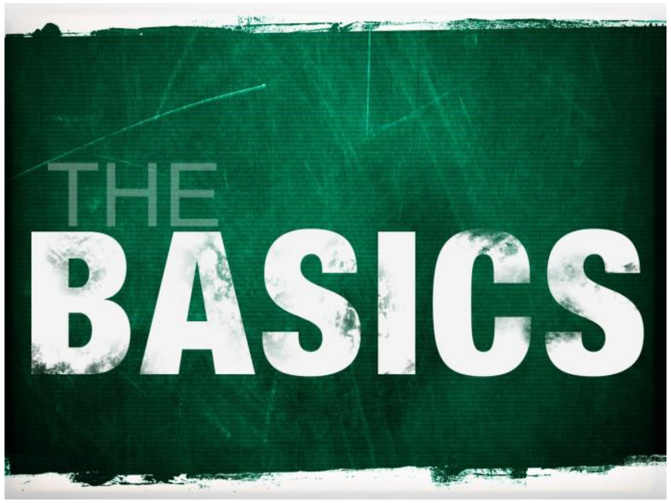
- Concise. Fewer lines of code means less typing and less effort at reading and understanding and less opportunities to make errors
- High-level. OOP and FP let you write more complex programs
- Statically typed. Verbosity is avoided through type inference so it looks like a dynamic language but it is not
- And also
  - Concurrency and parallelism
  - Integration with Java

Scala use cases

- Big data and data science
- Web Application Development, REST API Development
- Distributed System, Concurrency and Parallelism
- Scientific computation like NLP, Numerical Computing and Data Visualization
- Financial applications

Scala users

airbnb
twitter
intel
databricks
LinkedIn
theguardian
UniCredit
coursera
SAMSUNG
Walmart



The basics of the language

Getting started

```
C:\> scala

Welcome to Scala 3.3.1 (11.0.21, Java OpenJDK
64-Bit Server VM).

Type in expressions for evaluation. Or try
:help.


scala> 10 + 5.2

val res0: Double = 15.2

scala> :quit
```

REPL (Read-Evaluate-Print-Loop)

- The REPL is a command-line interpreter used as a “playground” to test the Scala code.

```
user@DESKTOP-UN3PBAM:~$ scala
Welcome to Scala 3.3.1 (11.0.21, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> 1 + 1
val res0: Int = 2

scala> val x = res0 *10
val x: Int = 20

scala> def sum(a: Int, b: Int): Int = a + b
def sum(a: Int, b: Int): Int
```

Compiling and executing scala programs

- You can compile and execute your scala programs from command line by typing `scala`

- `C:\> scala script.scala`

Program Entry point

- The main method is the entry point of a Scala Program
- In Scala 3, we can directly use the `@main` annotation

```
@main def hello() = println("Hello, Scala developer!")
```

Expressions

- You can output the results using `println`:

```
scala> println(10+5.2);
15.2

scala> println("Hello world");
Hello world

scala> println("Hello“ + " world");
Hello world
```

Values

- We can name the results of an expression with the keyword `val`

```
scala> val x = 2+3
val x: Int = 5
scala> println(x)
5
```

- Values, however cannot be reassigned

```
scala> x = 3
-- [E052] Type Error: -------------------------------------------------
--------
1 |x = 3
  |^^^^^
  |Reassignment to val x
  |
  | longer explanation available when compiling with ‘-explain‘
1 error found
```

Variables

- Variables can be defined with the keyword `var`. They are like values, but we can reassign them

```
scala> var x = 2 + 3
var x: Int = 5
scala> println(x)
5
scala> var x = 3
var x: Int = 3
scala> println(x)
3
```

Type can be automatically inferred and it cannot be changed when we reassign the value

Specifying the type

- Both in case of values and variables we can explicitly specify the type – or we can omit it and it will be automatically inferred

```
scala> val x: Int = 2 + 3

val x: Int = 5


scala> var x: Int = 2 + 3

var x: Int = 5
```

Values and variables

```
scala> val ten:Int = 10
//ten is an immutable
value of type Int
val ten: Int = 10
scala> ten = 11
-- [E052] Type Error: ---
-------------------------
-------------------------
-----
1 |ten = 11
  |^^^^^^^^
  |Reassignment to val
ten
```

```
scala> var ten:Int = 10
//ten is a variable whose
value can change
var ten: Int = 10
scala> ten = 11 //ten is
set to 11
ten: Int = 11
```

Blocks

- Expressions can be combined in blocks by surrounding them with { }.
- The result of the last expression is the result of the overall block

```
scala> println({val x = 1+1
     |       x+1
     |   })
3
```

Question 1

- What is the value of `msg` after the two instructions?

```
scala> val msg = "Hello"

scala> msg += "world"
```

A. Hello world
B. The instructions raise an error
C. World
D. Hello

Answer question 1

- What is the value of `msg` after the two instructions?

```
scala> val msg = "Hello"

scala> msg += "world"
```

```
scala> val msg = "Hello"
val msg: String = Hello

scala> msg += "world"
-- [E008] Not Found Error: ------
----------------------------------
--------------
1 |msg += "world"
  |^^^^^^
  |value += is not a member of
String - did you mean msg.!=?
```

A. Hello world
B. The instructions raise an error
C. World
D. Hello

Question 2

- What is the value of `msg` after the two instructions?

```
scala> var msg = "Hello world"

scala> msg = 5
```

A. Hello world
B. The instructions raise an error
C. 5
D. Hello world 5

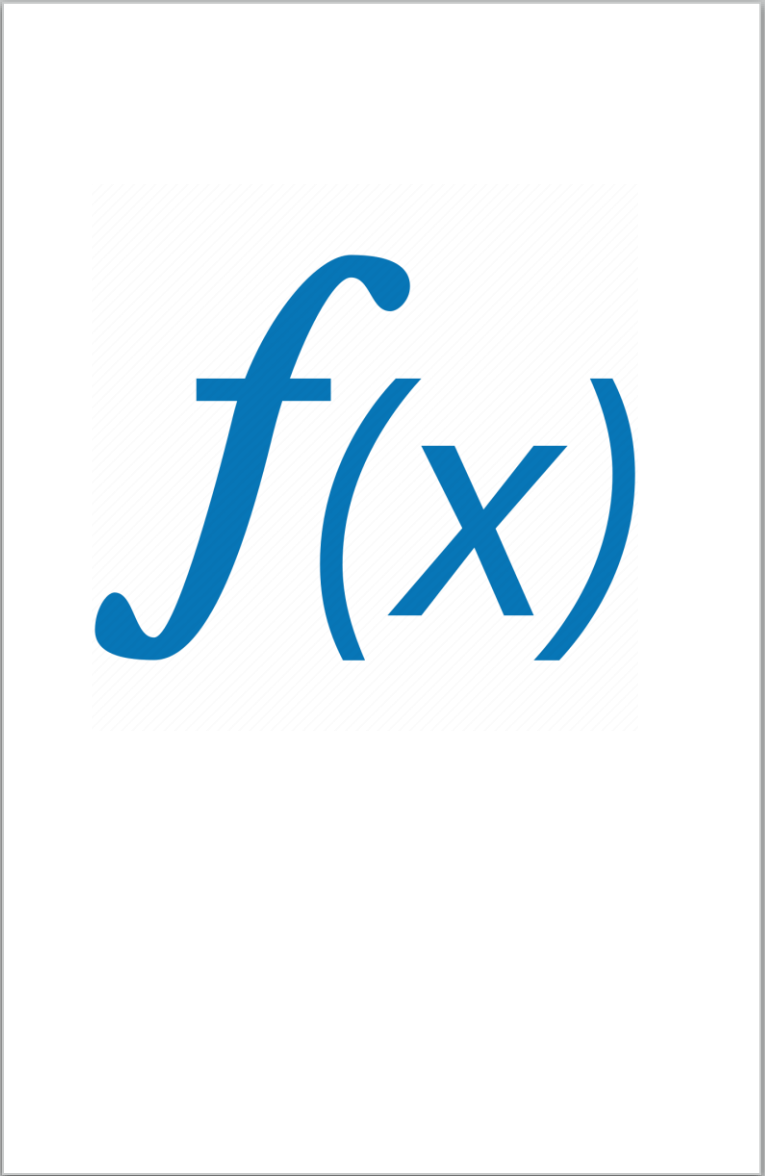
Answer question 2

- What is the value of msg after the two instructions?

```
scala> var msg = "Hello world"

scala> msg = 5
```

```
scala> var msg = "Hello world"
val msg: String = Hello

scala> msg = 5 //compiler error
-- [E007] Type Mismatch Error: --
---------------------------------
--------------
1 |msg = 5
  |      ^
  |      Found:    (5 : Int)
  |      Required: String
```

A. Hello world
B. The instructions raise an error
C. World
D. Hello



Functions

Functions

- Functions have a key role given the functional nature of Scala
- Functions are expressions that have parameters and take arguments
- Functions are defined by using => (similarly to ML)
    - On the left of =>, there is the list of parameters
    - On the right of =>, there is an expression involving the parameters

```
scala> val addOne = (x:Int) => x +1
val addOne: Int => Int =
Lambda$1400/0x00000008007ee040@4efe014f
scala> println(addOne(1))
2
```

Functions

- Functions can also have multiple parameters

```
scala> val add = (x: Int, y: Int) => x + y
val add: (Int, Int) => Int =
Lambda$1420/0x0000000800804040@73a116d
scala> println(add(1,2))
3
```

- Or no parameters

```
scala> val getanumber = () => 5
val getanumber: () => Int =
Lambda$1423/0x0000000800805440@430106cf
scala> println(getanumber())
5
```

Anonymous functions

- We can define anonymous functions, i.e., functions that have no name

```
scala> (x: Int) => x+1

val res0: Int => Int =
Lambda$1387/0x000000008007e5040@1d8dbf10
```

Higher order functions

- Function `map`

```
val salaries = List(20000, 70000, 40000)
val doubleSalary = (x: Int) => x * 2
val newSalaries = salaries.map(doubleSalary) // List(40000,
140000, 80000)
```

- We can also write it as

```
val newSalaries = salaries.map(x => x * 2) // List(40000,
140000, 80000)
```

- When we have a single parameter, and it appears only once in your anonymous function, we can replace it with _.

```
val newSalaries = salaries.map(_ * 2)// List(40000, 140000,
80000)
```

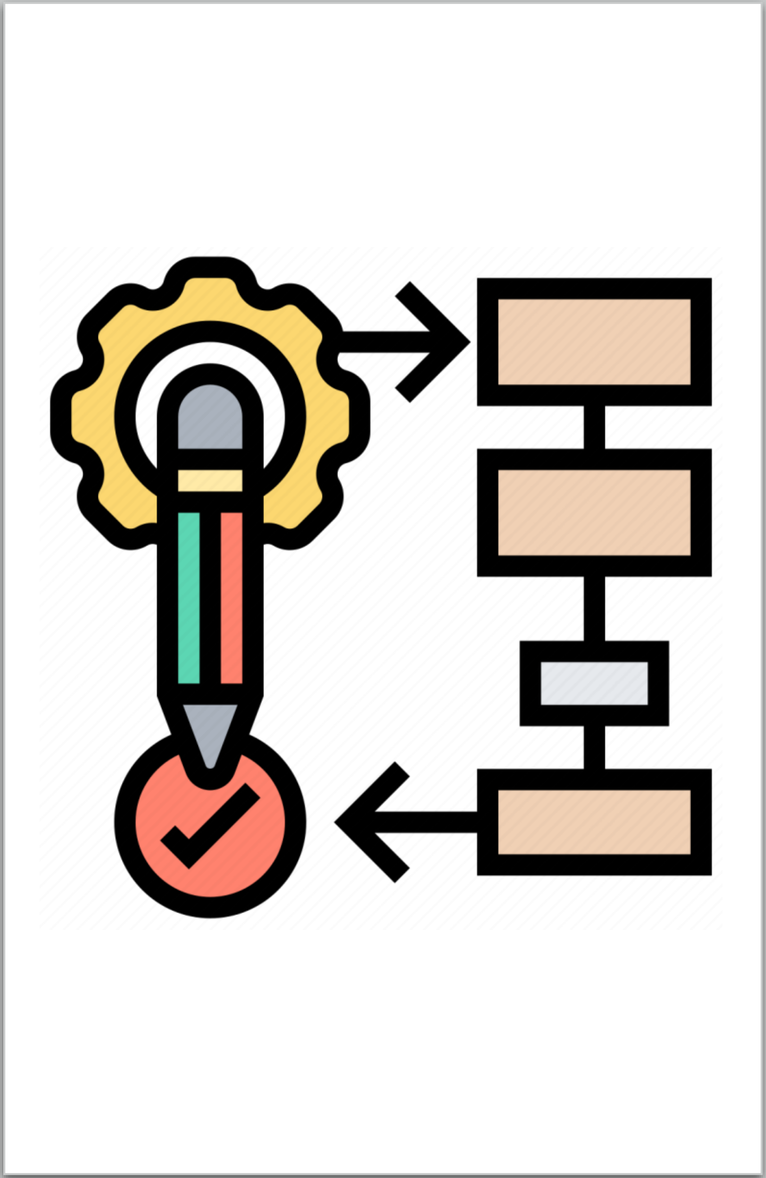
Higher-order functions

- Function `filter`

```
scala> val l = List ("aaa","bbbb","cc")
val l: List[String] = List(aaa, bbbb, cc)

scala> l.filter(s => s.length == 4)
val res: List[String] = List(bbbb)
```

- Alternative syntax

```
scala> l.filter(_.length ==4)
val res2: List[String] = List(bbbb)
```



Methods

Methods

- Methods look and behave very similar to functions, but there are some differences: methods are defined with the keyword `def`

```
def methodName(arg1: Type1, arg2: Type2, …):returnType =
{
    lines of code
    result
}
```

In Scala 3.0 you do not need to use curly brackets

- Historically methods have been part of the definition of a class. However, in Scala 3, using eta-expansion it is possible to have methods outside of classes

Method examples

```
scala> def add(x: Int, y: Int): Int = x + y
def add(x: Int, y: Int): Int
scala> println(add(1, 2))
3

scala> def addThenMultiply(x: Int, y:
Int)(multiplier: Int): Int = (x + y) * multiplier
def addThenMultiply(x: Int, y: Int)(multiplier:
Int): Int
scala> println(addThenMultiply(1, 2)(3))
9
```

Currying

```
scala> def nDividesM(m:Int)(n:Int) = (n%m ==0)
def nDividesM(m: Int)(n: Int): Boolean
scala> nDividesM(4)(2)
val res: Boolean = false

scala> val isEven = nDividesM(2)
val isEven: Int => Boolean =
Lambda$1736/0x0000000800918840@56a6aadb
scala> println(isEven(4))
true
scala> println(isEven(5))
false
```

Question 3

- What does the following code print?

```
scala> def triple(x: Int): Int = x * 3
scala> val tripleCopy: (Int) => Int = triple
scala> println(tripleCopy(5))
```

A. 15 15 15
B. The instructions raise an error
C. 5 5 5
D. 15

Answer question 3

- What does the following code print?

```
scala> def triple(x: Int): Int = x * 3

scala> val tripleCopy: (Int) => Int = triple

scala> println(tripleCopy(5))
```

```
def triple(x: Int): Int = x * 3
def triple(x: Int): Int

scala> val tripleCopy: (Int) =>
Int = triple
val tripleCopy: Int => Int =
Lambda$1722/0x00000008008e4040@59
20c4ed

scala> println(tripleCopy(5))
15
```

A. 15 15 15
B. The instructions raise an error
C. 5 5 5
D. 15

Question 4

- What does the following code print?

```
scala> val play=
(thing:String) =>  s"Let’s
play with $thing"

scala> def funify(thing:
String, f: String => String):
String = {
  f(thing) + " and have fun"
}
scala> println(funify("cats",
play))
```

A. “Let’s play with”
B. The instructions raise an error
C. “Let’s play with cats and have fun”
D. “Let’s play with and have fun”

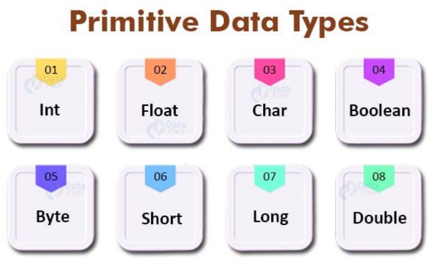
Answer question 4

- What does the following code print?

```
scala> val play=
(thing:String) =>  s"Let’s
play with $thing“

scala> def funify(thing:
String, f: String => String):
String = {
  f(thing) + " and have fun"
}
scala> println(funify("cats",
play))
```

A. “Let’s play with”
B. The instructions raise an error
C. “Let’s play with cats and have fun”
D. “Let’s play with and have fun”

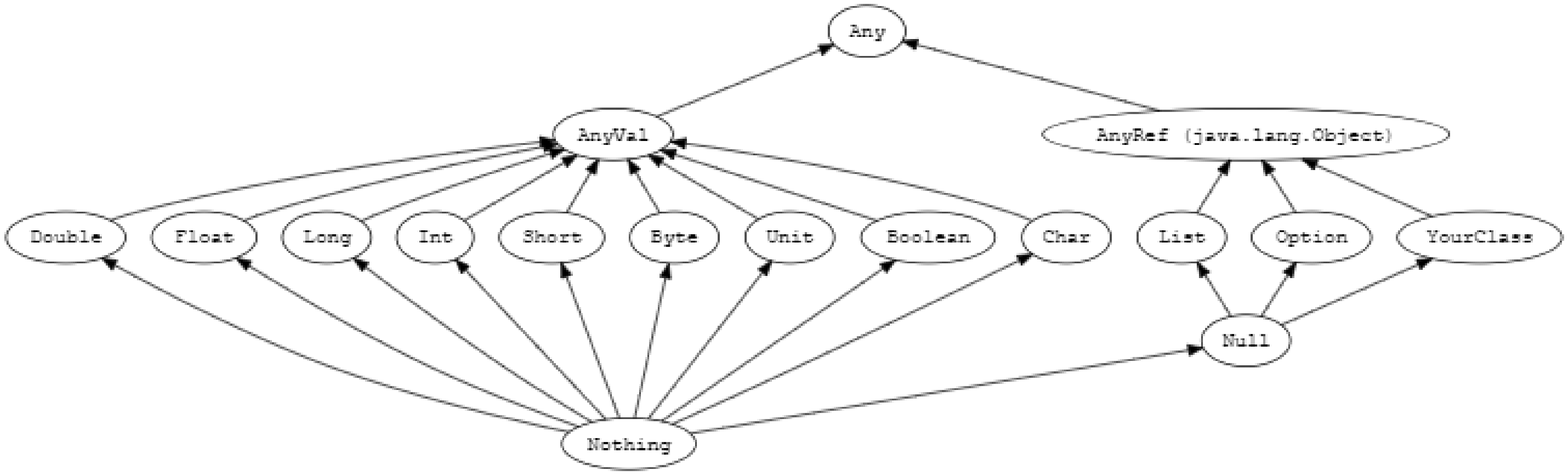
Data types



Unified type hierarchy

Scala types

- In Scala, all values have a type, including functions



Data types

- Byte
- Short
- Int
- Long
- Float
- Double
- Boolean
- String
- Char
- Unit

```
val b: Byte = 1
val i: Int = 1
val l: Long = 1
val s: Short = 1
val d: Double = 2.0
val f: Float = 3.0

val i = 123   // defaults to Int
val j = 1.0   // defaults to Double
val x = 1_000L   // val x: Long = 1000
val y = 2.2D     // val y: Double = 2.2
val z = 3.3F     // val z: Float = 3.3

val bool = true

val name = "Bill"   // String
val c = 'a'         // Char
```

String interpolation & multiline

- String interpolation with s and $
  - precede the string with the letter s and put a $ symbol before variable names

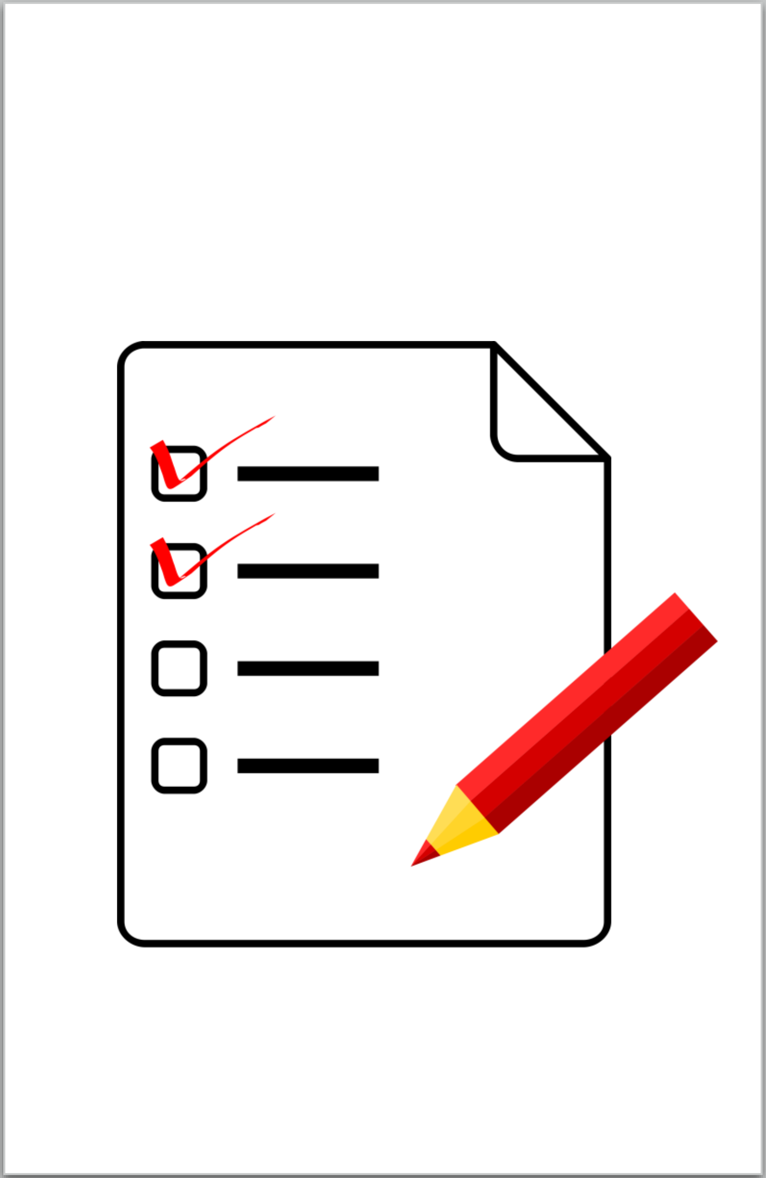
```
val firstName = "Donald"
val lastName = "Duck"
println(s"Name: $firstName $lastName")   // "Name: Donald Duck"
```

  - enclose arbitrary expressions in curly brackets

```
println(s"2 + 2 = ${2 + 2}")   // prints "2 + 2 = 4"
val x = -1
println(s"x.abs = ${x.abs}")   // prints "x.abs = 1“
```

- Multiline strings with three double quotes

```
val quote = """The essence of Scala:
               Fusion of functional and object-oriented
               programming in a typed setting."""
```



Lists and arrays

Lists

- Lists are immutable (content cannot be changed)
- List [String] contains Strings

```
scala> val l = List ("a","b","c")
val l: List[String] = List(a, b, c)

scala> l.head
val res1: String = a

scala> l.tail
val res2: List[String] = List(b, c)
```

Cons operator and concatenation

- As in ML
  - The cons operator `::` prepend an element
  - The concatenation operator `:::` concatenate two lists

```
scala> val l2 = "a"::l
val l2: List[String] = List(a, a, b, c)

scala> val l3 = 1::2::3::Nil
val l3: List[Int] = List(1, 2, 3)

scala> val l4 = List(1,2,3):::List (4,5)
val l4: List[Int] = List(1, 2, 3, 4, 5)
```

List of union types and any type

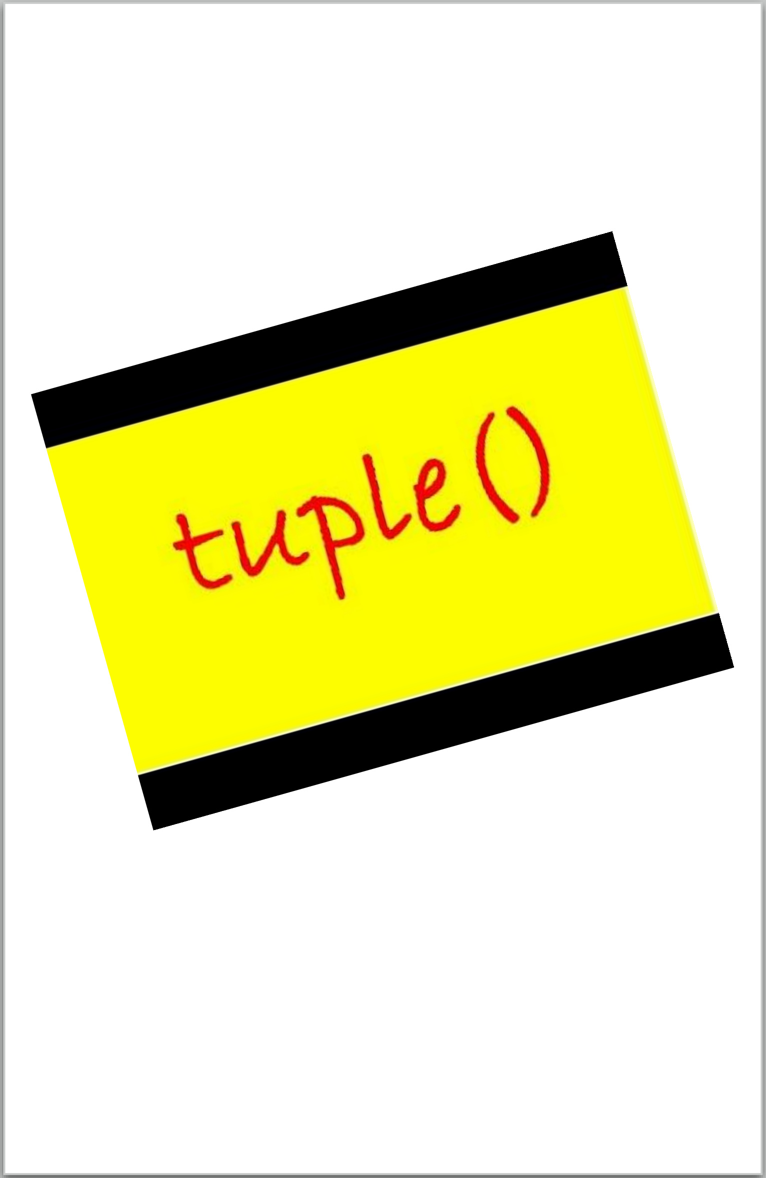
- It is possible to have mixed types in a list or a list of any type

```
val things: List[String | Int | Double] = List(1,
"two", 3.0) // with union types
val list: List[Any] = List(
     |   "a string",
     |   732,  // an integer
     |   'c',  // a character
     |   true, // a boolean value
     |   () => "an anonymous function returning a
string"
     | ) // with any
```

Arrays

- Lists are immutable, arrays are mutable

```
scala> val a = Array ("Java","Python")
val a: Array[String] = Array(Java, Python)
scala> a(0) = “Scala“

scala> val greets = new Array[String](2)
val greets: Array[String] = Array(null, null)
scala> greets(0) = "Hello"
scala> greets(1) = "world!“
```



Tuples

Tuples

- Sequence of (fixed number of) elements with different types

```
scala> (10, List(’a’,’b’), "string")
val res: (Int, List[Char], String) =
(10,List(a, b),string)
```

Tuples

- Tuples are immutable

```
scala> val ingredient = ("Sugar", 25)
val ingredient: (String, Int) = (Sugar,25)
```

- For accessing elements

```
scala> println(ingredient(0))
Sugar
scala> println(ingredient(1))
25
```

In Scala 2.0 you should use ._X, e.g., println(ingredient._1)

Tuple usage

- Tuples are useful in particular for returning multiple values from a method

```
def divMod(x: Int, y: Int) = (x/y,x%y)
divMod(x: Int, y: Int): (Int, Int)

scala> val dm = divMod(10, 3)
val dm: (Int, Int) = (3,1)
scala> dm(1)
val res1: Int = 3
scala> dm(2)
val res2: Int = 1

scala> val (d,m) = divMod(10, 3)
val d: Int = 3
val m: Int = 1
```

Pattern matching on tuples

- A tuple can also be built using pattern matching

```
scala> val (name, quantity) = ingredient
val name: String = Sugar
val quantity: Int = 25
scala> println(name)
Sugar
scala> println(quantity)
25
```

Pattern matching on tuples

- A tuple can be used with pattern matching

```
scala> val planets =
     |   List(("Mercury", 57.9), ("Venus", 108.2), ("Earth",
149.6), ("Mars", 227.9), ("Jupiter", 778.3))
     | planets.foreach {
     |   case ("Earth", distance) =>
     |     println(s"Our planet is $distance million kilometers
from the sun")
     |   case _ =>
     | }
Our planet is 149.6 million kilometers from the sun
val planets: List[(String, Double)] = List((Mercury,57.9),
(Venus,108.2), (Earth,149.6), (Mars,227.9), (Jupiter,778.3))
```

Question 5

- What does the following code print?

```
val myList = List(1, "hello", "world")
println(myList.head)
```

```
A. "hello"
B. Error
C. 1
D. List(1, "hello")
```

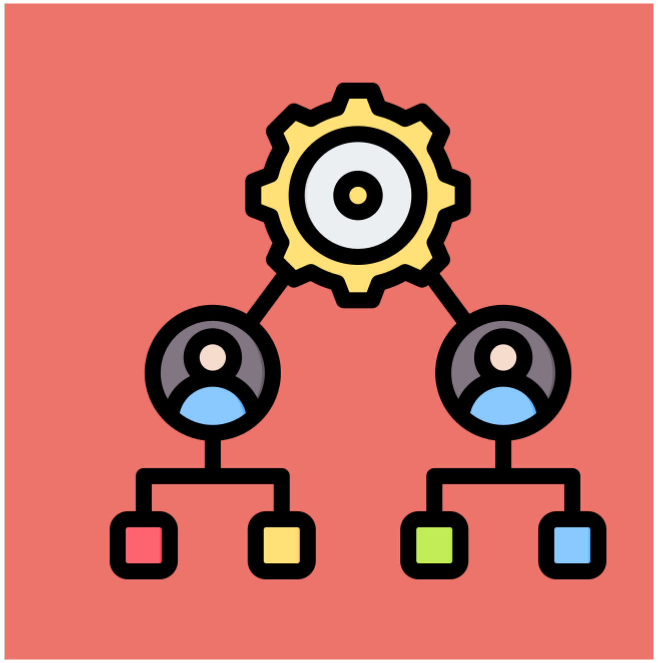
Answer question 5

- What does the following code print?

```
val myList = List(1, "hello", "world")
println(myList.head)
```

A. "hello"
B. Error
C. 1
D. List(1, "hello")

Control structures



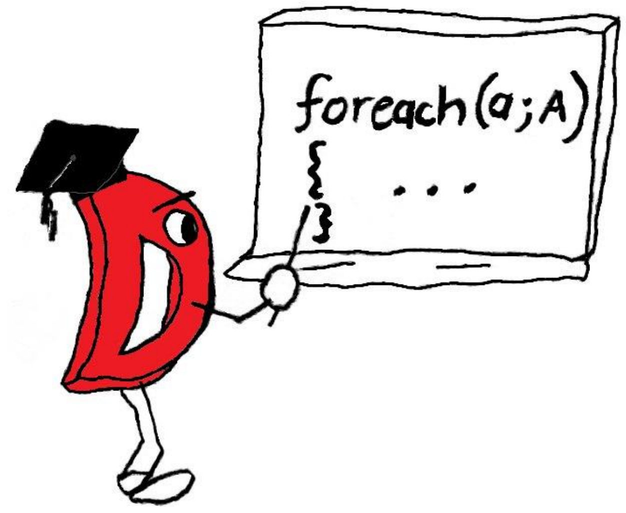
If/else

- Very similar to other languages

```
if x < 0 then
  println("negative")
else if x == 0 then
  println("zero")
else
  println("positive")
```

- As in ML this is a value and not a statement

```
val x = if a < b then a else b
```



For loops and for comprehension

For loops

- The `for` keyword is used to create a for loop – do can also be omitted

```
scala> val ints = List(1, 2, 3, 4, 5)
val ints: List[Int] = List(1, 2, 3, 4, 5)
```

$$\texttt{scala> for i }\underbrace{\texttt{<- ints}}_{\text{generator}}\texttt{ do }\underbrace{\texttt{println(i)}}_{\text{body of the loop}}$$

```
1
2
3
4
5
```

For loops: alternative syntax

-
  ```
  scala> for (i<-0
  until 10)
        | print(s"$i ")
  ```

-
  ```
  scala> for (i<-
  Range(0,20))
        | print(s"$i ")
  0 1 2 3 4 5 6 7 8 9
  ```

-
  ```
  scala> for (i<-0
  until 20 by 2)
        | print(s"$i ")
  ```

-
  ```
  scala> for (i<-
  Range(0,20,2))
        | print(s"$i ")
  0 2 4 6 8 10 12 14 16
  18
  ```

Guards

- We can use one or more `if` inside a for loop

```
scala> for
     |   i <- ints
     |   if i > 2
     | do
     |   print(s"$i ")
3 4 5
```

- We can also have multiple generators and guards

```
scala> for
     |   i <- 1 to 3
     |   j <- ’a’ to ’c’
     |   if i == 2
     |   if j == ’b’
     | do
     |   println(s"i = $i, j = $j")
i = 2, j = b
```

Foreach

- Given

```
scala> val list1 = List ("s1","s2","s3")
val list1: List[String] = List(s1, s2, s3)
```

- The following 3 calls are equivalent
  - `list1.foreach((s:String)=>println(s))`
  - `list1.foreach(s => println(s))`
  - `list1.foeach(println)`

```
s1
s2
s3
```

For expressions (for comprehension)

- We can use `for` followed by `yield` (instead of do) to create for expressions used to calculate and yield results

```
scala>  val doubles = for i <- ints yield i * 2
val doubles: List[Int] = List(2, 4, 6, 8, 10)

scala> val names = List("chris", "ed", "maurice")
val names: List[String] = List(chris, ed, maurice)

scala> val capNames = for name <- names yield
name.capitalize
val capNames: List[String] = List(Chris, Ed, Maurice)
```

For comprehensions

```
scala> val userBase = List(
     |   User("Travis", 28),
     |   User("Kelly", 33),
     |   User("Jennifer", 44),
     |   User("Dennis", 23))
val userBase: List[User] = List(User(Travis,28), User(Kelly,33),
User(Jennifer,44), User(Dennis,23))

scala> val twentySomethings =
     |   for user <- userBase if user.age >=20 && user.age < 30
     |   yield user.name  // i.e., add this to a list
     |
val twentySomethings: List[String] = List(Travis, Dennis)

scala> twentySomethings.foreach(println)
Travis
Dennis
```

For comprehensions

```
scala> def foo(n: Int, v: Int) =
     |    for i <- 0 until n
     |        j <- 0 until n if i + j == v
     |    yield (i, j)
     |
def foo(n: Int, v: Int): IndexedSeq[(Int, Int)]

scala> foo(10, 10).foreach {
     |   (i, j) => println(s"($i, $j) ")
     | }
(1, 9)
(2, 8)
(3, 7)
(4, 6)
(5, 5)
(6, 4)
(7, 3)
(8, 2)
(9, 1)
```

Pattern matching

Pattern matching

- Like switch statement but much more powerful. It can be used in place of a series of if/else statements

```
value match
        case x
        case y
```

```
scala> val x: Int = Random.nextInt(10)
val x: Int = 7

scala> x match
     |    case 0 => "zero"
     |    case 1 => "one"
     |    case 2 => "two"
     |    case _ => "other"
     |
val res4: String = other
```

Pattern matching with types

- A method that flattens a nested list

```
scala> def flatten(list: List[Any]): List[Any] =
       list match{
           case (x: List[Any])::xs =>
flatten(x)::flatten(xs)
           case x::xs => x::flatten(xs)
           case Nil => Nil
       };
def flatten(list: List[Any]): List[Any]

scala> val nested = List(1,List(2,3),4)
val nested: List[Int | List[Int]] = List(1, List(2, 3),
4)
```

Question 6

- What does the following code print?

```
val odds = List(3, 5, 7)
var result = 1
odds.foreach( (num: Int) => result *= num )
println(result)
```

A. Error
B. 15
C. 1
D. 105

Answer question 6

- What does the following code print?

```
val odds = List(3, 5, 7)
var result = 1
odds.foreach( (num: Int) => result *= num )
println(result)
```

A. Error
B. 15
C. 1
D. 105

Question 7

- What does the following code print?

```
def fizzBuzz(num: Int) = (num % 3, num % 5) match {
  case (0, 0) => "FizzBuzz"
  case (0, _) => "Fizz"
  case (_, 0) => "Buzz"
  case _ => ""
}
println(fizzBuzz(3))
```

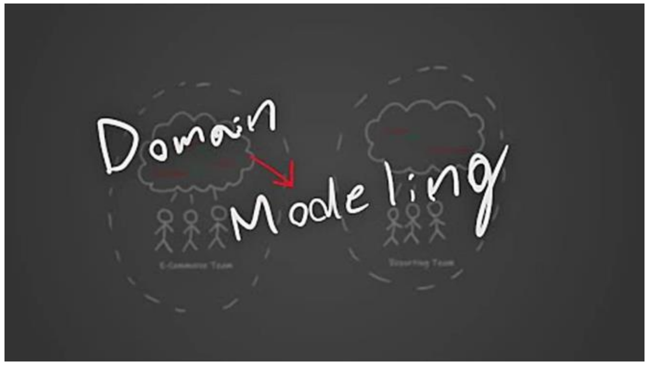
A. "Fizz“
B. "FizzBuzz"
C. Error
D. “Buzz"

Answer question 7

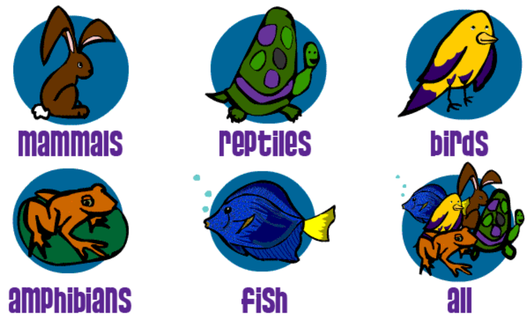
- What does the following code print?

```
def fizzBuzz(num: Int) = (num % 3, num % 5) match {
  case (0, 0) => "FizzBuzz"
  case (0, _) => "Fizz"
  case (_, 0) => "Buzz"
  case _ => ""
}
println(fizzBuzz(3))
```

A. "Fizz“
B. "FizzBuzz"
C. Error
D. “Buzz"



OOP Domain modeling



Classes

Classes

- We can define classes with the keyword `class`

```
class className(par1: Type1, par2: Type2, …):{
   Class definition
   }
```

In Scala 3.0 you do not need to use curly brackets

```
scala> class Greeter(prefix: String, suffix:
String):
     |   def greet(name: String): Unit =
     |     println(prefix + name + suffix)
     |
// defined class Greeter
```

Class instantiation

- An instance of the class can be done calling the constructor

```
val name = new class (arg1, arg2, ... )


scala> val greeter = Greeter("Hello, ", "!")

val greeter: Greeter = Greeter@107ebdad

scala> greeter.greet("Scala developer") //
Hello, Scala developer!

Hello, Scala developer!
```

In Scala 3.0 you do not need to use new

Class examples

```
class Point(var x: Int, var y: Int):
  def move(dx: Int, dy: Int): Unit =
    x = x + dx
    y = y + dy
  override def toString: String =
    s"($x, $y)"
end Point // defined class Point
```

- The class `Point` has four members: the variables `x` and `y` and the methods `move` and `toString`.

```
scala> val point1 = Point(2, 3)
val point1: Point = (2, 3)
scala> println(point1.x)
2
scala> println(point1)
(2, 3)
```

Constructors

- Constructors can have optional parameters by providing default values

```
scala> class Point(var x: Int = 0, var y: Int = 0)
// defined class Point

scala> val origin = Point()
val origin: Point = Point@6669cba
scala> val point1 = Point(1)
val point1: Point = Point@a1b7549
scala> println(point1.x)
1
scala> println(origin.x)
0
scala> val point2 = Point(y=2)
val point2: Point = Point@1f57666b
scala> println(point2.x)
0
scala> println(point2.y)
2
```

Private members and getter/setter syntax

```
scala> class Point:
     |   private var _x = 0
     |   private var _y = 0
     |   private val bound = 100
     |   def x: Int = _x
     |   def x_=(newValue: Int): Unit =
     |     if newValue < bound then
     |       _x = newValue
     |     else
     |       printWarning()
     |   def y: Int = _y
     |   def y_=(newValue: Int): Unit =
     |     if newValue < bound then
     |       _y = newValue
     |     else
     |       printWarning()
     |   private def printWarning(): Unit =
     |     println("WARNING: Out of bounds")
     | end Point
// defined class Point
```

- Members are public by default. We can use the `private` access modifier to hide them from outside
- Here data is stored in private variables `_x` and `_y` and `def  x` and `def  y` are used for accessing private data

Case classes

Case classes

- Case classes are a special type of classes whose instances are immutable
- Differently from class instantiations that are compared by reference, they are compared by value
- We can define case classes with the keyword `case class`

```
case class name (par1: type1, par2: type2, ...)

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

Case class instantiations

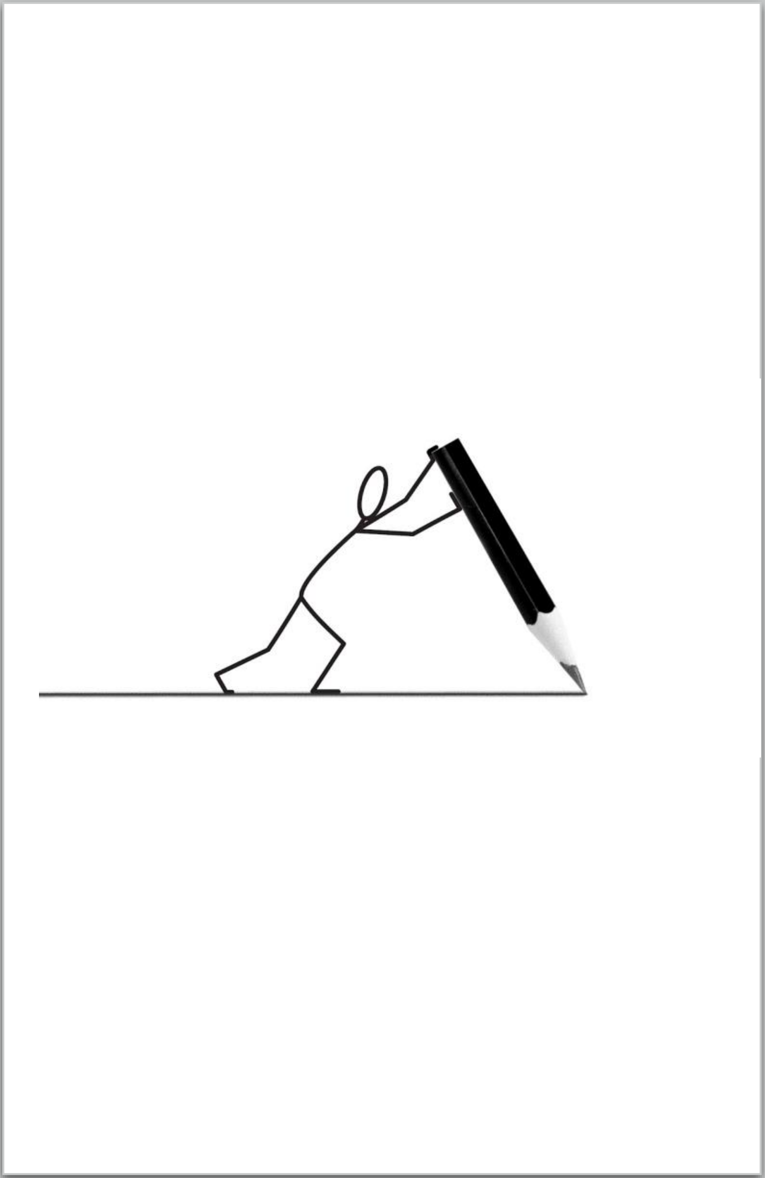
- Case classes can be instantiated

```
val name = case_class (arg1, arg2, ... )

scala> val point1 = Point(1, 2)
val point1: Point = Point(1,2)

scala> val point2 = Point(1, 2)
val point2: Point = Point(1,2)

scala> val yetAnotherPoint = Point(2, 2)
val yetAnotherPoint: Point = Point(2,2)
```

Case class instance comparisons

- Case class instances are compared by value and not by reference

```
scala> if point1 == point2 then
     |   println(s"$point1 and $point2 are the same.")
     | else
     |   println(s"$point1 and $point2 are different.")
Point(1,2) and Point(1,2) are the same.


scala> if point1 == yetAnotherPoint then
     |   println(s"$point1 and $yetAnotherPoint are the same.")
     | else
     |   println(s"$point1 and $yetAnotherPoint are different.")
Point(1,2) and Point(2,2) are different.
```



Traits

Traits

- Traits are abstract data types containing certain fields and methods. In Scala, a class can only extend one other class, but it can extend multiple traits
- We can specify a trait through the keyword `trait`

```
trait traitName:
      trait method definition

scala> trait Greeter:
     |   def greet(name: String): Unit
     |
// defined trait Greeter
```

Traits

```
scala> trait Greeter:
     |   def greet(name: String): Unit
     |
// defined trait Greeter
```

- Traits can also have default implementations

```
scala> trait Greeter:
     |   def greet(name: String): Unit =
     |     println("Hello, " + name + "!")
     |
// defined trait Greeter
```

Trait extensions

- Traits can be extended through the keyword `extends`

```
class className extends traitName

scala> class DefaultGreeter extends Greeter
// defined class DefaultGreeter
scala> class CustomizableGreeter(prefix: String, postfix: String) extends Greeter:
     |   override def greet(name: String): Unit =
     |     println(prefix + name + postfix)
// defined class CustomizableGreeter

scala> val greeter = DefaultGreeter()
val greeter: DefaultGreeter = DefaultGreeter@519e14f6
scala> greeter.greet("Scala developer")
Hello, Scala developer!

scala> val customGreeter = CustomizableGreeter("How are you, ", "?")
val customGreeter: CustomizableGreeter = CustomizableGreeter@23f9d0ce
scala> customGreeter.greet("Scala developer")
How are you, Scala developer?
```

DefaultGreeter can extend even more than one class

Traits with generic types and abstract methods

- Traits become useful with generic types and with abstract methods

```
scala> trait Iterator[A]:
     |   def hasNext: Boolean
     |   def next(): A
     |
// defined trait Iterator
```

Extending traits

- Extending `trait Iterator[A]` requires type `A` and implementations of the two methods

```
scala> class IntIterator(to: Int) extends Iterator[Int]:
     |   private var current = 0
     |   override def hasNext: Boolean = current < to
     |   override def next(): Int =
     |     if hasNext then
     |       val t = current
     |       current += 1
     |       t
     |     else
     |       0
     | end IntIterator
// defined class IntIterator

scala> val iterator = new IntIterator(10)
val iterator: IntIterator = IntIterator@343727b5
scala> iterator.next()  // returns 0
val res0: Int = 0
scala> iterator.next()  // returns 1
val res1: Int = 1
```

Question 8

- What does the following code print?

```
class Kitchen(color: String, floorType: String = "tile") {
  def describe = {
    s"The kitchen has $floorType floors"
  }
}
var myKitchen = new Kitchen("purple")
println(myKitchen.describe)
```

```
A. The kitchen has tile floors
B. Error
C. The kitchen has purple floors
D. The kitchen has $floorType floors
```

Answer question 8

- What does the following code print?

```
class Kitchen(color: String, floorType: String = "tile") {
  def describe = {
    s"The kitchen has $floorType floors"
  }
}
var myKitchen = new Kitchen("purple")
println(myKitchen.describe)
```

```
A. The kitchen has tile floors
B. Error
C. The kitchen has purple floors
D. The kitchen has $floorType floors
```

Summary

- Introduction to Scala
- The basics of the language
- Data Types
- Control structures
- OOP Domain modeling