FUNCTIONAL PROGRAMMING IN SCALA

d.jureczko@gmail.com
@DamianJureczko

AGENDA

- Scala - a quick intro
- Functional programming
- Functional constructions in Scala

SCALA

- Created by Martin Odersky
- First release in 2004

SUPPORTS OBJECT-ORIENTED PROGRAMMING

Everything is an object

```
val x = 10

x.toString
```

Operators are functions

```
val y = x + 10

val z = x.+(4)
```

Functions like operators (infix notation)

```
names.mkString(",")

names mkString ","
```

Define your own types

```
abstract class Vehicle {
  def move(): Unit
}

class Car extends Vehicle {
  override def move(): Unit = {
    println("drive")
  }
}
```

SUPPORTS FUNCTIONAL PROGRAMMING

First-class functions, and more...

```
// function type
(Int, Int) => Int

// anonymous function
(x: Int, y: Int) => x + y
```

RUNS ON JVM

Interoperates with Java libraries

```
import java.time.Instant

val now = Instant.now()

import com.google.common.cache.CacheBuilder

val usersCache: Cache[String, User] = CacheBuilder.newBuilder()
                                        .maximumSize(100).build()
```

STATICALLY TYPED

Find errors early

```
var name = "John"
name = "Mark"

name = 2 // compilation error !!!

def abs(x: Int): Int

abs("Adam") // compilation error !!!
```

SCALA IS ABOUT CONCISENESS

Express more with less code

```
class User(email: String, password: String)
```

TYPE INFERENCE

Compiler already knows the type

```
val firstName: String = "John"

val lastName = "Smith" // better

val age = 30
```

It's everywhere

```
val add: (Int, Int) => Int = (x: Int, y: Int) => x + y

// we know the type of x and y
val mul: (Int, Int) => Int = (x, y) => x * y

val sum: Int = add(2, 3)

// we know the type of the result
val product = mul(2, 3)
```

SCALA GOODIES

- Case classes
- Pattern matching
- Collections

CASE CLASSES

Immutable and easy to use data structures

```
// declare
case class User(email: String, password: String)

// create
val admin = User("admin@company.com", "buddy")

// access fields
val adminEmail = admin.email

// create copy
val otherAdmin = admin.copy(email = "admin2@company.com")

// compare
assert(admin != otherAdmin)
```

PATTERN MATCHING

Powerful matching technique

```
something match {
  case "value" => println("it's String equal to 'value'")

  case 10 => println("it's Int equal to 10")

  case s: String => println("it's String with value: " + s)

  case _ => println("it's something else")
}
```

Does magic with case classes

```
user match {
  case User("admin@company.com", "buddy") =>
    println("it's administrator")

  case User(email, password) =>
    println("it's " + email + ", his password is: " + password)
}
```

COLLECTIONS

- lists, sets, maps, ...
- immutable
- mutable
- functional API

Easy to create

```
val students = List("Adam", "John", "Andrew")

val studentGrades = Map("Adam" -> 4, "John" -> 3, "Andrew" -> 5)

val cities = Set("London", "Berlin", "Warsaw")
```

Immutable - for elegance

```
val students = List("Adam", "John", "Andrew")

// new collection
val moreStudents = "Michael" :: students
```

Mutable - for speed

```
val students = mutable.ArrayBuffer("Adam", "John", "Andrew")

students += "Michael"
```

Easy to use functional API

```
val students = List("Adam", "John", "Andrew")

students.find(_ == "John") // Some(John)

students.filter(_.length > 4) // List(Andrew)

students.map(_.length) // List(4, 4, 6)
```

FUNCTIONAL PROGRAMMING

Programming paradigm

Imperative	**vs.**	**Functional**
statements		functions/expressions
side effects		no side effects
mutable program state		immutable data

IMPERATIVE

How things should be done

```
List<Student> students;

int counter = 0;
for (Student student : students) {
    if (student.getGrade() > 3) {
        counter++;
    }
}
```

FUNCTIONAL

What should be done

```
val counter = students.count(_.grade > 3)
```

FIRST-CLASS FUNCTIONS

function is a first-class citizen

- can be assigned to a variable
- can be passed as an argument of a function
- can be returned from a function

HIGH-ORDER FUNCTIONS

- take other functions as argument
- return functions as result

Assign function to a variable

```
case class Student(name: String, grade: Int)

// function object
val goodStudent: Student => Boolean = student => student.grade > 3

assert(goodStudent(Student("Smith", 3)) == false)

assert(goodStudent(Student("Jones", 4)) == true)
```

Pass function to a high-order function

```
// List high-order funtion
def count(predicate: Student => Boolean): Int

val students = List(Student("Smith", 3), Student("Jones", 4))

val counter = students.count(goodStudent)

assert(counter == 1)
```

Return function from a high-order function

```
// high-order function
def gradeHigherThen(threshold: Int): Student => Boolean =
  student => student.grade > threshold

val above3 = gradeHigherThen(3)

val above4 = gradeHigherThen(4)

val counter = students.count(above3)
```

PURE FUNCTIONS

No side effects

Computes a result based of its inputs

```
def add(x: Int, y: Int) = x + y
```

Referential transparency

expression is referentially transparent if it can be replaced with is corresponding value without changing the program's behavior

Possible replacement

```
val sum1 = add(2, 2)

val sum2 = add(2, 2)
```

```
val sum1 = 4

val sum2 = 4
```

Impure function

```
var acc = 0

def impureAdd(x : Int, y: Int) = {
  acc = acc + x + y
  acc
}
```

Not referentially transparent

```
val sum1 = impureAdd(2, 2)

val sum2 = impureAdd(2, 2)
```

```
val sum1 = 4 // here it's 4

val sum2 = 4 // here it's 8 and the order matters
```

Pure functions simplify

- reasoning about program behavior
- composing programs
- proving correctness - testing
- optimizing - caching, parallelization, reordering

FUNCTIONAL DATA STRUCTURES

Immutable

Operated by pure functions

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

val oddNumbers = numbers.filter(_ % 2 == 0) // new list
```

Quick copy

```
val list1 = 1 :: Nil

assert(list1.head == 1)
assert(list1.tail == Nil)

val list2 = 2 :: list1

assert(list2.head == 2)
assert(list2.tail == List(1))
```

Thread safety

Immutable data can be safely shared between threads

```
def changeGrade(student: Student): Future[Student] = Future {
  student.copy(grade = 5)
}
```

FUNCTIONAL CONSTRUCTIONS IN SCALA

OPTIONAL VALUE

```
Option[+A]
```

Has two subclasses

```
val someName: Option[String] = Some("John")

val noneName: Option[String] = None
```

Optional result of a function

```
def findUser(email: String): Option[User]
```

better then

```
def findUser(email: String): User

val user = findUser("john.smith@company.com")

if (user != null) {
  ...
}
```

Pattern matching Options

```
val user = findUser("john.smith@company.com")

user match {
  case Some(User(email, password)) =>

  case None =>
}
```

MONADS

Containers

```
trait M[T] {
  def map(f: T => S): M[S]

  def flatMap(f: T => M[S]): M[S]
}
```

Option is a monad

```
Option[+A] {
  def map[B](f: (A) ⇒ B): Option[B] // only if nonempty

  def flatMap[B](f: (A) ⇒ Option[B]): Option[B] // only if nonempty
}
```

Map Option

```
val user: Option[User] = findUser("john.smith@company.com")

val password: Option[String] = user.map(u => u.password)
```

FlatMap Option

```
val user: Option[User] = findUser("john.smith@company.com")

val password: Option[String] = user.flatMap { u =>
  if (u.password.nonEmpty) Some(u.password) else None
}
```

Transforming collections

```
val names = List("John", "Mark", "Andrew", "Micheal")

names.map(_.length) // List(4, 4, 6, 7)
```

ERROR HANDLING

Try - catch, the non functional way

```
try {
  val user = findUser("john.smith@company.com") // may fail
  user.password // possible NullPointerException
} catch {
  case t: Throwable => // handle error
}
```

Try

```
Try[+T]
```

```
def findUser(email: String): Try[User]

findUser("john.smith@company.com") match {
  case Success(User(email, password)) =>

  case Failure(exception) =>
}
```

Try is a monad

```
val user = findUser("john.smith@company.com")

val password  = user.map(user => user.password)
```

Either error or correct value

```
Either[+A, +B]

case class Error(message: String)

def findUser(email: String): Either[Error, User]
```

Pattern matching Either

```
findUser("john.smith@company.com") match {
  case Right(User(email, password)) =>

  case Left(Error(message)) =>
}
```

Right-biased Either (coming soon in Scala 2.12.0)

```
val user = findUser("john.smith@company.com")

val password = user.map(u => u.password) // only if user is Right
```

TAKEAWAYS

- Scala is a powerful programing language
- Functional programming simplifies your life

THANK YOU