CS4215: Programming Language Implementation
2018/9, Sem 2

Introduction to Scala

Lecture 2

Scala: Functional Programming for the JVM

- Combines functional and object-oriented programming elegantly
- Completely interchangeable with Java
  - Can use all Java libraries
  - A Java coder would not know whether an API was written in Java or Scala
- Modern, powerful collections framework
- Great IDE support
  - Use IntelliJ Community Edition with Scala plugin

Scala Use Cases

- Big Data and Data Science (Kafka, Flink, Spark)
- Web Application Development, REST API Development (Lift, Play)
- Distributed Systems, Concurrency, and Parallelism (AKKA)
- Scientific Computation
- NLP
- Data Visualization

Hello World

```
object MyApp {
   def main(args: Array[String]): Unit = {
     println("Hello World!")
   }
 }

 object MyApp extends App {
   println("Hello World!")
 }
```

void

Define Some Variables

```
val msg = "Hello, world!"

msg = "Goodbye cruel world!" // error: reassignment to val

var greeting = "Hello, world!"

greeting = "Leave me alone, world!" // OK

val msg2: java.lang.String = "Hello again, world!"

val msg3: String = "Hello again, world!"
```

Simple Method

```
def getFullName(
    firstName: String,
    lastName: String
): String = {
    firstName + " " + lastName
}
```

optional

Real World Method

```
def getFullName(firstName: String, lastName: String): String = {
   val result: StringBuilder = new StringBuilder

   if(!firstName.trim.isEmpty) {
       result append firstName
   }

   if(!lastName.trim.isEmpty) {
       if(!result.isEmpty) {
           result append " "
       }

       result append lastName
   }

   result.toString
 }
```

Infix/Dot Notation

```
1 + 2
```

```
(1).+(2)
```

```
result.append(a).append(b).append(c)
result append a append b append c
```

Real World Method

```
def getFullName(firstName: String, lastName: String): String = {
   val result: StringBuilder = new StringBuilder

   if(!firstName.trim.isEmpty) {
       result append firstName
   }

   if(!lastName.trim.isEmpty) {
       if(!result.isEmpty) {
           result append " "
       }

       result append lastName
   }

   result.toString
}
```

Nice

```
def getFullName(firstName: String, lastName: String) =
   List(firstName, lastName) filterNot (_.trim.isEmpty) mkString " "
```

lambda
argument used only once

Everything is an Expression

```
val color = if(user.isBlocked) "red" else "green"


val p =  {
  case e1: NumberFormatException => extractInt(e)
  case e2: IllegalArgEx => … e2 ..
}


val number = try "123".toInt catch p
```

Default Arguments

```
def getUser(
    firstName: String = "John",
    lastName: String = "Doe",
    age: Int = -1
) = {
    // ...
}
```

Named Arguments

- How many times you saw this?

```
createUser(user, true, false, false, true, false, false)
```

- Isn’t this better?

```
createUser(
   user = user,
   encryptPassword = true,
   admin = false,
   ldapAuth = false,
   suspicious = true,
   blocked = false,
   visible = false
)
```

Class User - Java 1/4

```
public class User {
   private String firstName;
   private String lastName;
   private int age;

   public User(String firstName, String lastName, int age) {
     this.firstName = firstName;
     this.lastName = lastName;
     this.age = age;
   }

   public User(String firstName, String lastName) {
     this.firstName = firstName;
     this.lastName = lastName;
   }

   public User() {}
```

Class User - Java 2/4

```
public String getFirstName() { return firstName; }
 public void setFirstName(String firstName) { this.firstName = firstName;}

 public String getLastName() { return lastName; }
 public void setLastName(String lastName) { this.lastName = lastName; }

 public int getAge() { return age; }
 public void setAge(int age) { this.age = age; }
```

Class User - Java 3/4

```
@Override
 public int hashCode() {
   final int prime = 31;
   int result = 1;
   result = prime * result + age;
   result = prime * result + ((firstName == null) ? 0 : firstName.hashCode());
   result = prime * result + ((lastName == null) ? 0 : lastName.hashCode());
   return result;
 }

 @Override
 public boolean equals(Object o) {
   if (this == o) return true;
   if (o == null || getClass() != o.getClass()) return false;
   User other = (User) o;
   return age == other.age && firstName.equals(other.firstName) &&
       lastName.equals(other.lastName);
 }
```

Class User - Java 4/4

```
    @Override
     public String toString() {
       return "User(" +
            firstName + ", " +
            lastName + ", " +
            age + ")";
     }
}
```

Class User - Scala

```
case class User(
    var firstName: String,
    var lastName: String,
    var age: Int
)
```

Option[A]

- Container for an optional value of type A, i.e. values that may be present or not
- If the value of type A is present, the Option[A] is an instance of Some[A]
- If the value is absent, the Option[A] is the object None
- By stating that a value may or may not be present on the type level, you and any other developers who work with your code are forced by the compiler to deal with this possibility
- There is no way you may accidentally rely on the presence of a value that is really optional

Creating an option

```
val greeting: Option[String] = Some("Hello world")
```

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

Working with optional values

```
case class User(id: Int, firstName: String, lastName: String,
       age: Int, gender: Option[String])

object UserRepository {
  private val users = Map(
  1 -> User(1, "John", "Doe", 32, Some("male")),
  2 -> User(2, "Johanna", "Doe", 30, None))

  def findById(id: Int): Option[User] = users.get(id)
  def findAll = users.values
}
```

Working with optional values

```
val user1 = UserRepository.findById(1)
if(user1.isDefined) {
  println(user1.get.firstName)
} // will print "John"


val user2 = User(2, "Johanna", "Doe", 30, None)
println("Gender: " + user.gender.getOrElse("not specified"))
// will print "Gender: not specified"
```

Pattern matching

Java

```
switch (whatIsThis) {
  case 8:
  case 10:
    doSomething();
    break;
  case 12:
    doSomethingElse();
    break;
  default:
    doDefault();
}
```

Scala

```
whatIsThis match {
  case 8 | 10 => something
  case 12 => somethingElse
  case _ => defaultValue
}
```

Wildcard

```
whatIsThis match {
    case _ => "anything!"
}
```

Constant pattern

```
whatIsThis match {
    case 42 => "a magic no."
    case "hello!" => "a greeting"
    case math.Pi => "another magic no."
    case _ => "something else"
}
```

Variable pattern

```
whatIsThis match {
   case 0 => "zero"
   case somethingElse => "not zero: " + somethingElse
}
```

Typed pattern

```
whatIsThis match {
   case n: Int => "aah, a number!?"
   case c: Character => "it's" + c.name
 }
```

Constructor pattern

there are no other subclasses except in this file

```
sealed abstract class Shape
case class Circle( radius : Double ) extends Shape
case class Rectangle( width : Double, height : Double ) extends Shape
case class Triangle( base : Double, height : Double ) extends Shape
```

we can create an exhausted match

```
whatIsThis match {
    case Circle( radius ) => Pi * ( pow( radius, 2.0 ) )
    case Rectangle( 1, height ) => height
    case Rectangle( width, 1 ) => width
    case Rectangle( width, height ) => width * height
    case Triangle( 0, _ ) | Triangle( _, 0 ) => 0
    case Triangle( base, height ) => height * base / 2
}
```

Tuple patterns

```
val whatIsThis = (first, second, third, fourth) // a tuple, i.e. an object of type Tuple

whatIsThis match {
    case (a, b) => "Tuple2"
    case (42, math.Pi, _) => "magic numbers + anything"
    case (s: String, _, _) => "matched string on first position " + s
    case (a, b, c) => "matched " + a + b + c
    case _ => "no match"
  }
```

Pattern Matching on Option type

```
val user = User(2, "Johanna", "Doe", 30, None)

val gender = user.gender match {
    case Some(gender) => gender
    case None => "not specified"
}

println("Gender:" + gender)
```

Functional Programming

- No assignment
- No side effects; the following are avoided/restricted:
  - Modifying a variable
  - Modifying a data structure in place
  - Setting a field on an object
  - Throwing an exception
  - Printing to the console or reading user input
  - Reading from or writing to a file
- Functions are first class citizens
- Higher-order functions

Function Literal

lambda

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

(a: Int, b: Int, c: Int) => {
     val aSquare = a * a
     val bSquare = b * b
     val cSquare = c * c

     aSquare + bSquare == cSquare
}
```

Function Value

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

val isRightTriangle = ( a: Int, b: Int, c: Int ) => {
     val aSquare = a * a
     val bSquare = b * b
     val cSquare = c * c

     aSquare + bSquare == cSquare
}
```

Function Type

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

 val isPythagoras: (Int, Int, Int) => Boolean =
 ( a: Int, b: Int, c: Int ) => {
    val aSquare = a * a
    val bSquare = b * b
    val cSquare = c * c

    aSquare + bSquare == cSquare
 }
```

Function Application

```
add(3, 8) // 11

isPythagoras(1,2,3) // false

isPythagoras(3,4,5) // true
```

Traversing - Java

```
public List<User> findUserByFirstName(List<User> users, String firstName) {
  List<User> foundUsers = new ArrayList<User>();
  for(User user: users) {
    if(user.getFirstName().contains(firstName)) {
      foundUsers.add(user);
    }
  }
  return foundUsers;
}

public List<User> findUserByLastName(List<User> users, String lastName) {
  List<User> foundUsers = new ArrayList<User>();
  for(User user: users) {
    if(user.getLastName().contains(lastName)) {
      foundUsers.add(user);
    }
  }
  return foundUsers;
}
```

Traversing - Scala

```
users.filter((user: User) => user.firstName.contains("o"))
users.filter((user: User) => user.lastName.contains("Mar"))

users.filter(user => user.firstName.contains("o"))
users.filter(user => user.lastName.contains("Mar"))

users.filter(_.firstName.contains("o"))
users.filter(_.lastName.contains("Mar"))
```

Call by Name (a la Haskell)

```
def delayed(t :=> Long) = {
  println("In delayed method")
  println("Param:" + t)
  t
}

delayed({println("Parameter evaluation"); 100})

//Output
In delayed method
Parameter evaluation
Param: 100
```

Traits

- Partial classes
- Can contain implementations and fields
- Linking traits into classes is done dynamically, at runtime (tricky).

```
trait Pet {
  var age: Int = 0
  def greet(): String = {
    return “Hi”
  }
}
```

```
class Dog extends Pet {
  override def greet() = “Woof”
}

trait ExclamatoryGreeter extends
Pet {
  override def greet() =
                super.greet() + “!”
}
```

Mixin Traits

```
val pet = new Dog with ExclamatoryGreeter

println(pet.greet()) // Woof!
```

Resources

- https://scala-lang.org
- https://stackoverflow.com/tags/scala/info
- #scala on Slack
- https://www.coursera.org/specializations/scala
- https://code.corp.indeed.com/scala/skeleton-play-daemon