Methodologies for Software Processes Lecture 3 - Scala

Content

- **Case classes**
- **Pattern matching**
- **Singleton Objects**
- **Regular Expressions**
- **Extractor Objects**
- **For Comprehensions**
- **Generic Classes: Variances, Upper Type Bounds, Lower Type Bounds**
- **Inner classes**
- **Abstract Type Members**
- **Self Type**
- **Polymorphic Methods**
- **Type Inference**
- **Operators**
- **By-Name Parameters, Default Parameter Values, Named Arguments**
- **Packages, Imports, Package objects**

References

NOTE: The slides are based on the following free tutorials. You may want to consult them too.

1. https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html
2. https://docs.scala-lang.org/tour/tour-of-scala.html
3. https://docs.scala-lang.org/overviews/scala-book/introduction.html

Case Classes

- **Problem:** **a program to manipulate very simple arithmetic expressions composed of sums, integer constants and variables, for instance 1+2 and (x+x)+(7+y)**

- **Problem Representation:** **as a tree, where nodes are operations (here, the addition) and leaves are values (here constants or variables).**
    - **Java representation: an abstract super-class for the trees, and one concrete sub-class per node or leaf.**
    - **functional programming language: an algebraic data-type**
    - **Scala: case classes which is somewhat in between the two**

Case Classes

classes Sum, Var and Const are declared as case classes

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Case Classes

Differences from standard classes:

- the **new** keyword is not mandatory to create instances of these classes (i.e., one can write **Const(5) instead of new Const(5)**)

- getter functions are automatically defined for the constructor parameters (i.e., it is possible to get the value of the **v** constructor parameter of some instance **c** of class Const just by writing **c.v**)

- default definitions for methods **equals** and **hashCode** are provided, which work on the structure of the instances and not on their identity

- a default definition for method **toString** is provided, and prints the value in a "source form" (e.g., the tree for expression **x+1** prints as **Sum(Var(x),Const(1))**)

- instances of these classes can be decomposed through **pattern matching**

Pattern Matching

- is a mechanism for checking a value against a pattern.
- a successful match can also deconstruct a value into its constituent parts.
- it is a more powerful version of the switch statement in Java

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
matchTest(3)  // other
matchTest(1)  // one
```

Pattern Matching – Case Classes

```
abstract class Notification

case class Email(sender: String, title: String, body: String) extends Notification
case class SMS(caller: String, message: String) extends Notification
case class VoiceRecording(contactName: String, link: String) extends Notification


def showNotification(notification: Notification): String = {
  notification match {
    case Email(sender, title, _) =>
      s"You got an email from $sender with title: $title"
    case SMS(number, message) =>
      s"You got an SMS from $number! Message: $message"
    case VoiceRecording(name, link) =>
      s"You received a Voice Recording from $name! Click the link to hear it: $link"
  }
}
val someSms = SMS("12345", "Are you there?")
val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")

println(showNotification(someSms))  // prints You got an SMS from 12345! Message: Are you there?
println(showNotification(someVoiceRecording))  // you received a Voice Recording from Tom! Click the link to hear it: voicerecording.org/id/123
```

Pattern Matching – Guards

```
def showImportantNotification(notification: Notification, importantPeopleInfo:
Seq[String]): String = {
  notification match {
    case Email(sender, _, _) if importantPeopleInfo.contains(sender) =>
      "You got an email from special someone!"
    case SMS(number, _) if importantPeopleInfo.contains(number) =>
      "You got an SMS from special someone!"
    case other =>
      showNotification(other) // nothing special, delegate to our original
showNotification function
  }
}
```

```
val importantPeopleInfo = Seq("867-5309", "jenny@gmail.com")
val importantSms = SMS("867-5309", "I'm here! Where are you?")

println(showImportantNotification(importantSms, importantPeopleInfo))
//prints You got an SMS from special someone!
```

Pattern Matching – on Type

```
abstract class Device
case class Phone(model: String) extends Device {
  def screenOff = "Turning screen off"
}

case class Computer(model: String) extends Device {
  def screenSaverOn = "Turning screen saver on..."
}


def goIdle(device: Device) = device match {
  case p: Phone => p.screenOff
  case c: Computer => c.screenSaverOn
}
```

Pattern Matching – Sealed Classes

Traits and classes can be marked sealed which means all subtypes must be declared in the same file. This assures that all subtypes are known.

```
sealed abstract class Furniture
case class Couch() extends Furniture
case class Chair() extends Furniture

def findPlaceToSit(piece: Furniture): String = piece match {
  case a: Couch => "Lie on the couch"
  case b: Chair => "Sit on the chair"
}
```

Example

- **Problem:** **a function to evaluate an expression in some environment.**
  - **The aim of the environment is to give values to variables.**
  - **For example, the expression x+1 evaluated in an environment which associates the value 5 to variable x, written { x -> 5 }, gives 6 as result.**

- **Environment representation:**
  - **some associative data-structure like a hash table**
  - **a function which associates a value to a (variable) name**

  - **Scala: a function which, when given the string "x" as argument, returns the integer 5, and fails with an exception otherwise.**

```
{ case "x" => 5 }
```

Example

- use the type **String => Int** for environments, but it simplifies the program if we introduce a name for this type, and makes future changes easier

- the type **Environment** can be used as an alias of the type of functions from **String to Int**

```
type Environment = String => Int
```

Example

Pattern matching over the tree t:

1. checks if the tree t is a Sum, and if it is, it binds the left sub-tree to a new variable called l and the right sub-tree to a variable called r, and then proceeds with the evaluation of the expression following the arrow;

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Example

Pattern matching over the tree t:

2. if the tree is not a Sum, it goes on and checks if t is a Var; if it is, it binds the name contained in the Var node to a variable n and proceeds with the right-hand expression

```scala
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Example

Pattern matching over the tree t:

3. if the second check also fails, that is if t is neither a Sum nor a Var, it checks if it is a Const, and if it is, it binds the value contained in the Const node to a variable v and proceeds with the right-hand side,

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Example

Pattern matching over the tree t:

4. finally, if all checks fail, an exception is raised to signal the failure of the pattern matching expression; this could happen here only if more sub-classes of Tree were declared

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Example

why we did not define eval as a method of class Tree and its subclasses?

Deciding whether to use pattern matching or methods has important implications on extensibility:

- when using methods: it is easy to add a new kind of node as this can be done just by defining a sub-class of Tree for it; on the other hand, adding a new operation to manipulate the tree is tedious, as it requires modifications to all sub-classes of Tree

- when using pattern matching: the situation is reversed: adding a new kind of node requires the modification of all functions which do pattern matching on the tree, to take the new node into account; on the other hand, adding a new operation is easy, by just defining it as an independent function.

Example

Derivative Example:

1. the derivative of a sum is the sum of the derivatives
2.the derivative of some variable v is one if v is the variable relative to which the derivation takes place, and zero otherwise
3.the derivative of a constant is zero

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

Example

- **the case expression for variables has a guard, an expression following the if keyword. This guard prevents pattern matching from succeeding unless its expression is true**
- **the wildcard, written _, which is a pattern matching any value, without giving it a name**

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

Example

```scala
def main(args: Array[String]): Unit = {
  val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
  val env: Environment = { case "x" => 5 case "y" => 7 }
  println("Expression: " + exp)
  println("Evaluation with x=5, y=7: " + eval(exp, env))
  println("Derivative relative to x:\n " + derive(exp, "x"))
  println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

Example

the output:

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

Singleton Objects

- An object is a class that has exactly one instance. It is created lazily when it is referenced, like a lazy val.
- As a top-level value, an object is a singleton.
- As a member of an enclosing class or as a local value, it behaves exactly like a lazy val.

```
package logging
object Logger {
  def info(message: String): Unit = println(s"INFO: $message")
}

import logging.Logger.info

class Project(name: String, daysToComplete: Int)
class Test {
  val project1 = new Project("TPS Reports", 1)
  val project2 = new Project("Website redesign", 5)
  info("Created projects")  // Prints "INFO: Created projects"
}
```

Companion Objects

- An object with the same name as a class is called a companion object. Conversely, the class is the object’s companion class.

- A companion class or object can access the private members of its companion.

- Use a companion object for methods and values which are not specific to instances of the companion class.

- static members in Java are modeled as ordinary members of a companion object in Scala.

Companion Objects

- The class Circle has a member area which is specific to each instance, and the singleton object Circle has a method calculateArea which is available to every instance.

```
import scala.math._

case class Circle(radius: Double) {
  import Circle._
  def area: Double = calculateArea(radius)
}

object Circle {
  private def calculateArea(radius: Double): Double = Pi * pow(radius, 2.0)
}

val circle1 = Circle(5.0)

circle1.area
```

Companion Objects

- The companion object can also contain factory methods:

```
class Email(val username: String, val domainName: String)

object Email {
  def fromString(emailString: String): Option[Email] = {
    emailString.split('@') match {
      case Array(a, b) => Some(new Email(a, b))
      case _ => None
    }
  }
}
```

Regular Expressions Patterns

- Regular expressions are strings which can be used to find patterns (or lack thereof) in data.
- Any string can be converted to a regular expression using the .r method.

- Next the numberPattern is a Regex (regular expression) which we use to make sure a password contains a number

```
import scala.util.matching.Regex

val numberPattern: Regex = "[0-9]".r

numberPattern.findFirstMatchIn("awesomepassword") match {
  case Some(_) => println("Password OK")
  case None => println("Password must contain a number")
}
```

Extractor Objects

- An extractor object is an object with an unapply method.
- the apply method is like a constructor which takes arguments and creates an object
- the unapply takes an object and tries to give back the arguments. This is most often used in pattern matching and partial functions.

```
import scala.util.Random

object CustomerID {
  def apply(name: String) = s"$name--${Random.nextLong}"

  def unapply(customerID: String): Option[String] = {
    val stringArray: Array[String] = customerID.split("--")
    if (stringArray.tail.nonEmpty) Some(stringArray.head) else None
  }
}

val customer1ID = CustomerID("Sukyoung")  // CustomerID.apply is called
customer1ID match {
  case CustomerID(name) => println(name)  // CustomerID.unapply is called
  case _ => println("Could not extract a CustomerID")
}
```

Extractor Objects

Pattern matching:

```
val customer2ID = CustomerID("Nico")
val CustomerID(name) = customer2ID
println(name)  // prints Nico
```

FOR Comprehensions

- have the form **for (enumerators) yield e**, where enumerators refers to a semicolon-separated list of enumerators.
- An enumerator is either a generator which introduces new variables, or it is a filter.
- A comprehension evaluates the body e for each binding generated by the enumerators and returns a sequence of these values.

```
def foo(n: Int, v: Int) =
   for (i <- 0 until n;
        j <- 0 until n if i + j == v)
   yield (i, j)

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

Variances

- is the correlation of subtyping relationships of complex types and the subtyping relationships of their component types.
- the lack of variance can restrict the reuse of a class abstraction

```
class Foo[+A] // A covariant class
class Bar[-A] // A contravariant class
class Baz[A]  // An invariant class
```

Covariance

- For some class List[+A], making A covariant implies that for two types A and B where A is a subtype of B, then List[A] is a subtype of List[B].

```
abstract class Animal {
  def name: String
}
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal

object CovarianceTest extends App {
  def printAnimalNames(animals: List[Animal]): Unit = {
    animals.foreach { animal =>
      println(animal.name)
    }
  }
 val cats: List[Cat] = List(Cat("Whiskers"), Cat("Tom"))
  val dogs: List[Dog] = List(Dog("Fido"), Dog("Rex"))

  printAnimalNames(cats) // Whiskers Tom

  printAnimalNames(dogs)// Fido Rex
}
```

Contravariance

- for some class Writer[-A], making A contravariant implies that for two types A and B where A is a subtype of B, Writer[B] is a subtype of Writer[A]

```
abstract class Printer[-A] {
  def print(value: A): Unit
}

class AnimalPrinter extends Printer[Animal] {
  def print(animal: Animal): Unit =
    println("The animal's name is: " + animal.name)
}

class CatPrinter extends Printer[Cat] {
  def print(cat: Cat): Unit =
    println("The cat's name is: " + cat.name)
}
```

Contravariance

for some class Writer[-A], making A contravariant implies that for two types A and B where A is a subtype of B, Writer[B] is a subtype of Writer[A]

```
object ContravarianceTest extends App {
  val myCat: Cat = Cat("Boots")

  def printMyCat(printer: Printer[Cat]): Unit = {
    printer.print(myCat)
  }

  val catPrinter: Printer[Cat] = new CatPrinter
  val animalPrinter: Printer[Animal] = new AnimalPrinter

  printMyCat(catPrinter)
  printMyCat(animalPrinter)
}
```

Invariance

- Generic classes in Scala are invariant by default. This means that they are neither covariant nor contravariant.

- In the context of the following example, Container class is invariant. A Container[Cat] is not a Container[Animal], nor is the reverse true.

```
class Container[A](value: A) {
  private var _value: A = value
  def getValue: A = _value
  def setValue(value: A): Unit = {
    _value = value
  }
}
```

Upper Type Bounds

- An upper type bound T <: A declares that type variable T refers to a subtype of type A

```
abstract class Animal {
 def name: String
}

abstract class Pet extends Animal {}

class Cat extends Pet {
  override def name: String = "Cat"
}

class Dog extends Pet {
  override def name: String = "Dog"
}

class Lion extends Animal {
  override def name: String = "Lion"
}
```

Upper Type Bounds

An upper type bound T <: A declares that type variable T refers to a subtype of type A

```
class PetContainer[P <: Pet](p: P) {
  def pet: P = p
}

val dogContainer = new PetContainer[Dog](new Dog)
val catContainer = new PetContainer[Cat](new Cat)




// this would not compile
val lionContainer = new PetContainer[Lion](new Lion)
```

Lower Type Bounds

- declare a type to be a supertype of another type.
- term B >: A expresses that the type parameter B or the abstract type B refer to a supertype of type A.
- In most cases, A will be the type parameter of the class and B will be the type parameter of a method.

```
trait Node[+B] {
  def prepend[U >: B](elem: U): Node[U]
}

case class ListNode[+B](h: B, t: Node[B]) extends Node[B] {
  def prepend[U >: B](elem: U): ListNode[U] = ListNode(elem, this)
  def head: B = h
  def tail: Node[B] = t
}

case class Nil[+B]() extends Node[B] {
  def prepend[U >: B](elem: U): ListNode[U] = ListNode(elem, this)
}
```

Lower Type Bounds

```
trait Bird
case class AfricanSwallow() extends Bird
case class EuropeanSwallow() extends Bird


val africanSwallowList = ListNode[AfricanSwallow](AfricanSwallow(), Nil())
val birdList: Node[Bird] = africanSwallowList
birdList.prepend(EuropeanSwallow())
```

Inner Classes

- In Scala it is possible to let classes have other classes as members. As opposed to Java-like languages where such inner classes are members of the enclosing class, in Scala such inner classes are bound to the outer object.

```
class Graph {
  class Node {
    var connectedNodes: List[Node] = Nil
    def connectTo(node: Node): Unit = {
      if (!connectedNodes.exists(node.equals)) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
  var nodes: List[Node] = Nil
  def newNode: Node = {
    val res = new Node
    nodes = res :: nodes
    res
  }
}
```

Inner Classes

```
val graph1: Graph = new Graph
val node1: graph1.Node = graph1.newNode
val node2: graph1.Node = graph1.newNode
node1.connectTo(node2)     // legal
```

```
val graph2: Graph = new Graph
val node3: graph2.Node = graph2.newNode
node1.connectTo(node3)     // illegal! But in Java is correct
```

Inner Classes

For nodes of both graphs, Java would assign the same type Graph.Node; i.e. Node is prefixed with class Graph. In Scala such a type can be expressed as well, it is written Graph#Node

```
class Graph {
  class Node {
    var connectedNodes: List[Graph#Node] = Nil
    def connectTo(node: Graph#Node): Unit = {
      if (!connectedNodes.exists(node.equals)) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
  var nodes: List[Node] = Nil
  def newNode: Node = {
    val res = new Node
    nodes = res :: nodes
    res
  }
}
```

Abstract Type Members

- Abstract types, such as traits and abstract classes, can in turn have abstract type members.
- the concrete implementations define the actual types

```
trait Buffer {
  type T
  val element: T
}
```

```
abstract class SeqBuffer extends Buffer {
  type U
  type T <: Seq[U]
  def length = element.length
}
```

```
abstract class IntSeqBuffer extends SeqBuffer {
  type U = Int
}
```

Compound Types

- Sometimes it is necessary to express that the type of an object is a subtype of several other types.
- In Scala this can be expressed with the help of compound types, which are intersections of object types.
- General form: A with B with C ...

```
trait Cloneable extends java.lang.Cloneable {
  override def clone(): Cloneable = {
    super.clone().asInstanceOf[Cloneable]
  }
}
trait Resetable {
  def reset: Unit
}

def cloneAndReset(obj: Cloneable with Resetable): Cloneable = {
  val cloned = obj.clone()
  obj.reset
  cloned
}
```

Self Type

- are a way to declare that a trait must be mixed into another trait, even though it doesn't directly extend it. That makes the members of the dependency available without imports.

```
trait User {
  def username: String
}

trait Tweeter {
  this: User =>  // reassign this
  def tweet(tweetText: String) = println(s"$username: $tweetText")
}

class VerifiedTweeter(val username_ : String) extends Tweeter with User {
// We mixin User because Tweeter required it
   def username = s"real $username_"
}

val realBeyoncé = new VerifiedTweeter("Beyoncé")
realBeyoncé.tweet("Just spilled my glass of lemonade")
// prints "real Beyoncé: Just spilled my glass of lemonade"
```

Polymorphic Methods

- methods in Scala can be parameterized by type as well as value. The syntax is similar to that of generic classes.

```
def listOfDuplicates[A](x: A, length: Int): List[A] = {
  if (length < 1)
    Nil
  else
    x :: listOfDuplicates(x, length - 1)
}
println(listOfDuplicates[Int](3, 4))  // List(3, 3, 3, 3)
println(listOfDuplicates("La", 8))  // List(La, La, La, La, La, La, La, La)
```

Type Inference

```
val businessName = "Montreux Jazz Café" // infer the value type as String

def squareOf(x: Int) = x * x // infer the result type as Int

def fac(n: Int) = if (n == 0) 1 else n * fac(n – 1)
//for recursive functions Scala cannot infer the result type


case class MyPair[A, B](x: A, y: B)
val p = MyPair(1, "scala") // type: MyPair[Int, String]

def id[T](x: T) = x
val q = id(1)             // type: Int
```

Operators

- In Scala, operators are methods.
- Any method with a single parameter can be used as an infix operator.

```
case class Vec(x: Double, y: Double) {
  def +(that: Vec) = Vec(this.x + that.x, this.y + that.y)
}

val vector1 = Vec(1.0, 1.0)
val vector2 = Vec(2.0, 2.0)

val vector3 = vector1 + vector2
vector3.x  // 3.0
vector3.y  // 3.0
```

By Name Parameters

- are only evaluated when used (in contrast to by-value parameters).
- to make a parameter called by-name, simply prepend => to its type.
- By-name parameters have the advantage that they are not evaluated if they aren't used in the function body.
- On the other hand, by-value parameters have the advantage that they are evaluated only once.

```
def whileLoop(condition: => Boolean)(body: => Unit): Unit =
  if (condition) {
    body
    whileLoop(condition)(body)
  }

var i = 2

whileLoop (i > 0) {
  println(i)
  i -= 1
}  // prints 2 1
```

Default Parameter Values

```
def log(message: String, level: String = "INFO") = println(s"$level: $message")
log("System starting")  // prints INFO: System starting
log("User not found", "WARNING")  // prints WARNING: User not found
```

```
class Point(val x: Double = 0, val y: Double = 0)
val point1 = new Point(y = 1)
```

Named Arguments

```
def printName(first: String, last: String): Unit = {
  println(first + " " + last)
}

printName("John", "Smith")  // Prints "John Smith"
printName(first = "John", last = "Smith")  // Prints "John Smith"
printName(last = "Smith", first = "John")  // Prints "John Smith"
```

Packages

- One convention is to name the package the same as the directory containing the Scala file.
- Scala is agnostic to file layout.
- The directory structure of an sbt project for package users might look like this:

```
package users
class User
```

- the structure:

```
- ExampleProject
  - build.sbt
  - project
  - src
    - main
      - scala
        - users
          User.scala
          UserProfile.scala
          UserPreferences.scala
    - test
```

Packages

```
package users {

  package administrators {
    class NormalUser
  }

 package normalusers {
    class NormalUser
  }

}
```

Imports

```
import users._  // import everything from the users package
import users.User  // import the class User
import users.{User, UserPreferences}  // Only imports selected members
import users.{UserPreferences => UPrefs}  // import and rename for convenience
```

```
def sqrtplus1(x: Int) = {
  import scala.math.sqrt
  sqrt(x) + 1.0
}
```

Package objects

- a convenient container shared across an entire package
- can contain arbitrary definitions, not just variable and method definitions.
- can even inherit Scala classes and traits.
- the source code for a package object is usually put in a source file named package.scala

```
// in file gardening/fruits/Fruit.scala
package gardening.fruits

case class Fruit(name: String, color: String)
object Apple extends Fruit("Apple", "green")
object Plum extends Fruit("Plum", "blue")
object Banana extends Fruit("Banana", "yellow")
```

Package objects

- Now assume we want to place a variable planted and a method showFruit directly into package gardening.fruits

```
// in file gardening/fruits/package.scala
package gardening
package object fruits {
  val planted = List(Apple, Plum, Banana)
  def showFruit(fruit: Fruit): Unit = {
    println(s"${fruit.name}s are ${fruit.color}")
  }
}

// in file PrintPlanted.scala
import gardening.fruits._
object PrintPlanted {
  def main(args: Array[String]): Unit = {
    for (fruit <- planted) {
      showFruit(fruit)
    }
  }
}
```