TECNOLOGIAS
E ARQUITETURA

Singleton Objects, Functional Objects, Classes, Case Classes, Parametric Fields, Overriding, Traits

Functional Object-Oriented Programming

Algebraic Data Types, Option and Try

Partially based on the book: Functional Programming in Scala. Paul Chiusano, Rúnar Bjarnason. Manning.

Singleton Objects, Functional Objects, Classes, Case Classes, Parametric Fields, Overriding, Traits and Companion

Singleton Objects

- Scala **cannot have static members** instead Scala has **singleton objects**
- Looks like a class definition with *object* keyword

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

Singleton object

One way to **think of singleton objects is as the home for any static methods** you might have written in Java.

How to invoke methods on singleton objects :

```
Circle.calculateArea(2.0)
```

Singleton objects **cannot take parameters**

Cannot instantiate a singleton object

A Scala Application

To run a Scala program, you must supply the name of a **singleton object** with a main method.

Any singleton object with a main method can be used as the **entry point** into an application.

```
object Summer {
  def main(args: Array[String]) {
     println(args)
  }
}
```

Companion object

- When a singleton object shares the **same name with a class**
- You must define both the class and its companion object in the **same source file**.
- The class is called the **companion class**. A class and its companion object can access each other’s private members.

```
class Circle(val radius: Double) {
  def area: Double = Circle.calculateArea(radius)
}

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

Classes and Mutable Objects – will be avoided!

```scala
class ChecksumAccumulator {
  private var sum = 0
  def add(b: Byte): Unit = {
    sum += b
  }
  def checksum(): Int = {
    return ~(sum & 0xFF) + 1
  }
}
```

```scala
val c = new ChecksumAccumulator
c.add(1)
c.checksum()
c.add(1)
c.checksum()
```

Functional Immutable Objects

do **not** have any mutable state

Advantages:

- often easier to reason about
- can pass them around quite freely
- concurrency

Disadvantage:

- require new information to be copied where otherwise, an update could be done

Class Rational

- *rational number* is a number that can be expressed as a ratio **n/d**
  - *n* and *d* are integers, except that *d* cannot be zero

- In mathematics, rational numbers do not have mutable state
  - you can add one rational number to another, but the result will be a **new** rational number.
  - the **original numbers will not change**

- **Immutable Rational** class will have the same property
  - E.g., when adding two Rational objects, a new Rational object will be created to hold the sum

Constructing a *Rational*

class parameters

Scala compiler
get the class parameters and
create a **primary constructor**

```
class Rational(n: Int, d: Int)
```

No body required
No fields required
No copy of constructor parameters to fields

Java: classes have constructors, which can take parameters
Scala: classes can take parameters directly

Constructing a *Rational*

```
class Rational(n: Int, d: Int) {
  println("Created "+ n +"/"+ d)
}
```

Can introduce code that is not part of a method or field

```
scala> new Rational(1, 2)
Created 1/2
res0: Rational = Rational@90110a
```

Adding fields

To keep Rational immutable, the add method **creates and returns a new Rational** that holds the sum.

```
class Rational(n: Int, d: Int) { // This won't compile
  require(d != 0)
  override def toString = n +"/"+ d
  def add(that: Rational): Rational =
    new Rational(n * that.d + that.n * d, d * that.d)
}
```

you can **only access *class parameters* values** on the object on which add method was invoked.
To access *class parameters*, you'll need to **make them into fields**.

Adding fields

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  override def toString = numer +"/"+ denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
```

```
scala> val oneHalf = new Rational(1, 2)
oneHalf: Rational = 1/2

scala> val twoThirds = new Rational(2, 3)
twoThirds: Rational = 2/3

scala> oneHalf add twoThirds
res3: Rational = 7/6
```

You can now **access** the numerator and denominator values from **outside the object**

```
scala> val r = new Rational(1, 2)
r: Rational = 1/2

scala> r.numer
res4: Int = 1

scala> r.denom
res5: Int = 2
```

Parametric Fields

To avoid **redundancy** and **code repetition**

```
class Rational(val numer: Int, val denom: Int)
```

Self references - *this*

```
def lessThan(that: Rational) =
  this.numer * that.denom < that.numer * this.denom
```

```
def max(that: Rational) =
  if (this.lessThan(that)) that else this
```

Auxiliary constructors

Constructors other than the primary constructor are **auxiliary constructors**

```
new Rational(5, 1) <=>
new Rational(5)
```

every auxiliary constructor must invoke another constructor of the same class as its first action, i.e. "… `= this( … )`"

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  def this(n: Int) = this(n, 1) // auxiliary constructor
  override def toString = numer +"/"+ denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
```

Private fields and methods

A rational can be **normalized** by dividing the numerator and denominator by their greatest common divisor.

e.g. 66/42 ⇔ 11/7
(gdc(66/42) is 6)

```
scala> new Rational(66, 42)
res7: Rational = 11/7
```

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g
  def this(n: Int) = this(n, 1)
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
  override def toString = numer +"/"+ denom
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

Method overloading

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g
  def this(n: Int) = this(n, 1)
  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
  def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)
```

```
scala> val x = new Rational(1, 2)
x: Rational = 1/2
```

```
scala> x + 2  //now works ⇔ x.+(2)
```

but

```
scala> 2 + x  //does not work
```

Create an implicit conversion:

```
implicit def intToRational(x: Int) = new Rational(x)

scala> 2 + x  // ☺
```

How to override equals

```
//wrong definition
def equals(other: Rational): Boolean =
   this.numer == other.numer &&
   this.denom == other.denom


scala> val x = new Rational(2,3)
scala> val y = new Rational(2,3)
scala> x.equals(y) //true

scala> val y2: Any = y
scala> x.equals(y2) //false!
```

does **not override** the standard method equals (just an overload)

How to override equals

```scala
class Rational(n: Int, d: Int) {
  require(d != 0)
  private val g = gcd(n.abs, d.abs)
  val numer = (if (d < 0) -n else n) / g
  val denom = d.abs / g
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
  override def equals(other: Any): Boolean =
    other match {
      case that: Rational =>
        (that canEqual this) &&
        numer == that.numer &&
        denom == that.denom
      case _ => false
    }
  def canEqual(other: Any): Boolean =
    other.isInstanceOf[Rational]
```

```
scala> val y2: Any = y
scala> x.equals(y2) //true!
```

Quite difficult to implement a correct equality method.

Prefer to define your classes of comparable objects as **case classes**. That way, the Scala compiler will add equals methods with the right properties automatically.

Case Classes

```
case class Book(isbn: String)
val frankenstein = Book("978-0486282114")
```

- keyword ***new*** was **not used** to instantiate the Book
- The compiler adds “natural” implementations of methods **toString** and **equals** to your class.
  - **compared by structure** and not by reference (msg2 and msg3 refer to different objects)

```
case class Message(sender: String, recipient: String, body: String)
val msg2 = Message("joe@cat.es", "gui@bec.ca", “msg")
val msg3 = Message("joe@cat.es", "gui@bec.ca", “msg")
val messagesAreTheSame = msg2 == msg3 // true
```

Case Classes

Good for modeling **immutable data**

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr
```

All arguments in the parameter list of a case class implicitly get a ***val* prefix**, so they are **maintained as fields**.

```
scala> val op = BinOp("+", Number(1), v)
op: BinOp = BinOp(+,Number(1.0),Var(x))
```

```
scala> op.left
res1: Expr = Number(1.0)
```

Case Classes

Case Classes **support pattern matching:**

Constructor patterns

```
Expr match {
 case BinOp("+", e, Number(0)) => println("a deep match")
 case _ =>
}
```

Typed patterns

```
def generalSize(x: Any) = x match {
 case s: String => s.length
 case m: Map[_, _] => m.size
 case _ => -1
}
```

```
scala> generalSize("abc")
res14: Int = 3

scala> generalSize(Map(1 -> 'a', 2 -> 'b'))
res15: Int = 2

scala> generalSize(Math.Pi)
res16: Int = -1
```

Abstract Class

```scala
abstract class Element {
  def contents: Array[String]
}
```

<- Abstract method that has no implementation

Extending Classes

Scala allows the inheritance from just one class only

```scala
class ArrayElement(conts: Array[String]) extends Element {
  def contents: Array[String] = conts
}
```

```
scala> val ae = new ArrayElement(Array("hello", "world"))
ae: ArrayElement = ArrayElement@d94e60
```

Traits

- used to **share interfaces and fields** between classes
- similar to Java 8's interfaces
- Classes and objects can extend <u>many</u> traits, but traits **cannot be instantiated** and therefore have no parameters.

```
trait HairColor

trait Iterator[A] {
   def hasNext: Boolean
   def next(): A
}
```

Traits

```
trait Philosophical {
  def philosophize() {
    println("I consume memory, therefore I am!")
  }
}
```

- Like Java interfaces **with concrete methods** but they can do much more (<u>out of scope</u>):
  - declare field and maintain state
  - work as stackable modifications

Trait – *extends*

```scala
class Frog extends Philosophical {
  override def toString = "green"
}
```

- *extends* keyword to **mix** in a trait; in that case you implicitly inherit the trait's superclass

```
scala> val frog = new Frog
frog: Frog = green

scala> frog.philosophize()
I consume memory, therefore I am!
```

Trait – defines a type

```
scala> val phil: Philosophical = frog
phil: Philosophical = green

scala> phil.philosophize()
I consume memory, therefore I am!
```

The type of *phil* is Philosophical, a trait.

Variable *phil* could have been initialized with any object whose class mixes in Philosophical.

Trait – *with* keywords

Only allowed to inherit from one class, but as many traits as you'd like.

- Use ***extends*** to extend the first trait
- Use ***with*** to extend subsequent traits

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "green"
}
```

```
class Animal
trait HasLegs

class Frog extends Animal with Philosophical with HasLegs {
  override def toString = "green"
}
```

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "green"
  override def philosophize() {
    println("It ain't easy being "+ toString +"!")
  }
}
```

```
scala> val phrog: Philosophical = new Frog
phrog: Philosophical = green

scala> phrog.philosophize()
It ain't easy being green!
```

A Scala Application - *App* Trait

can be used to **quickly turn objects into executable** programs. Here is an example:

```
object Main extends App {
   Console.println("Hello World: " + (args mkString ", "))
 }
```

No explicit main method is needed. Instead, the whole class body becomes the “main method”. *args* returns the current command line arguments as an array.

Enumeration

```
object Fingers extends Enumeration {
  type Finger = Value
  val Thumb, Index, Middle, Ring, Little = Value
}
```

The Enumeration class provides a type called **Value** to represent each of the enumeration values.

```
def isShortest(finger: Finger) = {
  finger == Little
}

def twoLongest() = {
  Fingers.values.toList.filter(finger => finger == Middle || finger == Index)
}
```

Functional OOP

Functional OOP

As we dive into OOP and immutable variables an interesting question arises:

"Why would we have an object if we are never going to change it?"

Answer: An Object is **no longer something that "acts"** instead it **"contains"** data (containers that encapsulate data).

"But how does the work get done?"

Answer: By **using static functions** that take our objects. (Remember: Scala **cannot have static members** instead Scala has **singleton objects)**

Example - Static Encapsulation with Singleton objects

```
class Contact(val contact_id: Int,
              val firstName: String,
              val lastName: String,
              val email: String,
              val enabled: Boolean) {

   def sendEmail(subject:String, body:String) = {
     println("To:" + email + "\nSubject:" + subject + "\nBody:" + body)
   }
}
```

How to send an email using an Email object (to be reused by each Contact object) that "contains" instead of "acts"?

Example - Static Encapsulation with Singleton objects

```
case class Email(address: String,
                 val subject: String,
                 val body: String) {

   def send():Boolean = Email.send(this)
}

object Email { //singleton where we keep our static methods

   def send(msg: Email): Boolean = {
      println(“To:” + msg.address + “\nSubject: ” + msg.subject +
      “\nBody: “ + msg.body)
      true
   }
}
```

Example - Static Encapsulation with Singleton objects

We can now modify the `sendEmail` method in `Contact` to create a new `Email` object and then call its `send()` method:

```
class Contact(val contact_id: Int,
              val firstName: String,
              val lastName: String,
              val email: String,
              val enabled: Boolean) {

   def sendEmail(subject:String, body:String) = {
      new Email(email, subject, body).send()
   }
}
```

`Email` class has become a container of the data itself.
We are calling into the **`Email` singleton to perform the `email` functionality**.

Algebraic Data Type - Tree, Option and Try

Algebraic Data Types – The Tree Example

Tree

the + in front of the type parameter A is a variance annotation that signals that A is a covariant parameter of Tree

```
sealed trait Tree[+A]
case object Empty extends Tree[Nothing]
case class Node[A](value: A,
                   left: Tree[A],
                   right: Tree[A]) extends Tree[A]

val tree = Node(42, Node(0, Empty, Empty), Empty)
print(tree) // prints Node(42,Node(0,Empty,Empty),Empty)
```

Option

Option represents optional values. Instances of Option are either an instance of **scala.Some** or the object **None**

```
val abc = new java.util.HashMap[Int, String]
abc.put(1, "A")
bMaybe = Option(abc.get(2))
bMaybe match {
  case Some(b) =>
    println(s"Found $b")
  case None =>
    println("Not found")
}
```

Java uses ***null*** to indicate no value. If a variable is allowed to be *null*, then you **must remember** to check it for null every time you use it. In Scala **must check** (otherwise type error)

Weakness: it doesn't tell you anything about why something failed

Try (contains the reason why something failed)

```
import scala.util.{Try, Success, Failure}

def makeInt(s: String): Try[Int] = Try(s.trim.toInt)

scala> makeInt("1")
res0: scala.util.Try[Int] = Success(1)

scala> makeInt("foo")
res1: scala.util.Try[Int] =
Failure(java.lang.NumberFormatException: For input
string:
```

Try with *match*

```
makeInt("hello") match {
 case Success(i) => println(s"Success, value is: $i")
 case Failure(s) => println(s"Failed, message is: $s")
}
```

Pattern Matching over Algebraic Data Types - Exercise

Complete the `size` function that counts the number of nodes in a tree:

```scala
//Example.scala
sealed trait MyTree[+A]
case object Empty extends MyTree[Nothing]
case class Node[A](value: A, left: MyTree[A], right: MyTree[A]) extends MyTree[A]

case class Example(myField: MyTree[Int]){

  def size() = Example.size(this.myField)
}

object Example{

  def size[A](t: MyTree[A]): Int = t match {
        ???
  }

  def main(args: Array[String]): Unit = {

    val tree1 = Node(42, Node(4, Empty, Empty), Node(84, Empty, Empty))
    val e = Example(tree1)
    println(s"Number of nodes of the tree: ${e.size()}")
  }
}
```

“Object oriented programming makes code understandable by encapsulating moving parts. Functional programming makes code understandable by minimizing moving parts.”

- Michael Feathers

TECNOLOGIAS
E ARQUITETURA