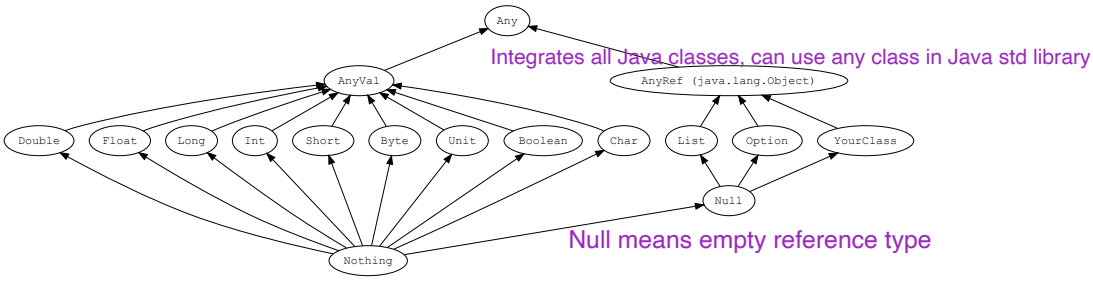
COMS W3137 - Recitation Notes
Week 4: OOP in Scala, Pattern Matching and Case Classes

Daniel Bauer

February 11, 2022

The OOP system in Scala is very similar to the one in Java. Specifically, most class instances are still subtypes of `java.lang.Object`. However, Scala takes this a step further. All types, including basic data types, are integrated into a *unified hierarchy*. Figure **??** illustrates what this type hierarchy looks like.
In Java, the common supertype of all classes is `Object`. Scala adds another level to this with the type `Any`. In addition to `java.lang.Object`, which has the alias `AnyRef` (*reference types*), Scala provides a type `AnyVal`, which is the supertype of all built-in basic data types described above (*value types*). Note that `Null` is also a type in Scala. All Java classes fit under `AnyRef`, as do all programmer defined classes. Maybe most notably, while the Java type hierarchy forms a tree, the Scala types form a lattice, a kind of algebraic structure that has a unique top element (here `Any`) and bottom element (here `Nothing`). This type hierarchy makes it possible to define more expressive type bounds than in Java, for example when defining generics.
Another difference between Java and Scala is that there are no interfaces in Scala. Instead, Scala has *traits*. Unlike interfaces, traits can not inherit from



Integrates all Java classes, can use any class in Java std library

Null means empty reference type

Nothing means nothing at all (like null in Java)

Figure 1: Basic types and reference types in the Scala type hierarchy. Source: `https://docs.scala-lang.org/resources/images/tour/unified-types-diagram.svg`

Scala has no primitive data types. Double/Float/Long/etc are all classes

In Java, each class can only inherit from one parent, but in Scala, classes can inherit from multiple parents, which makes the type hierarchy a lattice (unique top and unique bottom)

If the compiler can't figure out the data type, it will default to Nothing

each other.

1 Basic Data Types as Instances

As mentioned earlier, all objects in Scala, including "basic" value types are class instances. As a result, we can call methods on these instances.

```
scala> 42.toString()
res1: String = 42
```

When calling a method with zero parameters, the parentheses can be ommited.

```
scala> val x = 42
x : Int = 42
scala> x.toString
res1: String = 42
```

1.1 Operators as Methods

In fact, binary operators (such as arithmetic operations +,-,/,*) are implemented as methods on the corresponding data type. When defining your own classes, you can override these operators. In fact, you can make up your own binary operators. Let's first examine how these work for built-in data types.

```
scala> val x = 40
x: Int = 40
scala> x.+(2)
res0: Int = 42
scala> x.==(39+1)
res1: Boolean = true
```

== works for objects in Scala, so .equals isn't really needed

You can also use any method that expects a single parameter as an infix operator.

```
scala> "fortunate".contains("tuna")
res0: Boolean = true
scala> "fortunate" contains "tuna"
res1: Boolean = true
```

ie. x.+(10) = x + (10)

Note that by default the method is called on the left operand and the right operand because the argument. We have already ancountered an example in which the order is reversed: The `::` operator for constructing lists. The general rule is that any operator that ends with `:` corresponds to a method call on the right operand, using the left operand as the argument. For example, `1 :: 2 :: 3 :: Nil` is actually `Nil.::(3).::(2).::(1)`

2 Classes

Class definitions look almost like in Java, but there are some significant differences. Because Scala allows us to skip empty blocks, we do not even need { } to define a basic (empty) class. You can then immediately instantiate such a class using the `new` keyword, like in Java.

```
scala> class Test
defined class Test
scala> val test = new Test
test: Test = Test@4ae5ddc0
```

2.1 Methods

We have already seen method definitions. When defined inside a class, methods behave like methods in Java.

```
scala> class Test {
     |   def square(x : Int) : Int = x*x
     | }
defined class Test

scala> val t : Test = new Test()
t: Test = Test@28bd5015

scala> t.square(3)
res4: Int = 9
```

2.2 Constructors

One difference between Java and Scala is how constructors are handled. In Scala, the code for the *primary* constructor is simply the body of the class. This includes declarations for any instance variables.

```
scala> class Test {
     |     val foo : Int = 23;
     |     println("created a Test instance")
     | }
defined class Test

scala> val test = new Test
created a Test instance
test: Test = Test@c414eb3

scala> test.foo
res0: Int = 23
```

classes in Scala have a default constructor

Parameters passed to the primary constructor are specified in the class definition. They also automatically become instance variables.

```
scala> class Greeter(msg : String) {
     |       def greet : Unit = println(msg);
     | }
defined class Greeter

scala> val greeter = new Greeter("hello")
greeter: Greeter = Greeter@619c93ca

scala> greeter.greet // This is a method call
hello
```

A class can have auxiliary constructors (constructor overloading), which are always called `this`. These additional constructors always call the primary constructor, which is also referred to as `this`.

```
scala> class Greeter(msg : String) {   If error, try val msg
     |
     |      def this()  = {
     |           this("hello");
     |       }
     |
     |      def greet : Unit = println(msg);
     | }
scala> val greeter = new Greeter
greeter: Greeter = Greeter@10e9a5fe

scala> greeter.greet
hello
```

The parameters are `vals` by default, so they cannot be modified once the instance has been created. They are also *private*, so you cannot access them from outside the class.

```
scala> greeter.msg
<console>6: error: value msg is not a member of Greeter
```

You can declare instance variables, including class parameters to be `vars`, but this is *strongly* discouraged, as it would result in mutable objects.

2.3 Access Modifiers

Regular instance variables and methods are *public* by default, which makes them visible anywhere. This is a reasonable default for `vals`, because there is no risk that they could be modified from outside the class – they cannot be modified at all. The only two access modifiers in scala are `private`, which limits access to the class (and the companion object, see below) and `protected`, which

additionally grants access from subclasses.

```
scala> class TestClass {
     |     private def fact_rec(n : Int, result : Int) : Int = {
     |         if (n==1)
     |             result
     |         else
     |             fact_rec(n-1, result*n)
     |     }
     |
     |     def factorial(n : Int) : Int  = fact_rec(n,1);
     |}

scala> val test = new TestClass()
test: TestClass = TestClass@6371c5ec

scala> test.factorial(5)
res0: Int = 120
```

3 Singleton and Companion Objects

Unlike Java, which uses the *static* keyword, there are no class methods or class variables in Scala. Instead, Scala supports *singleton* objects. A singleton object behaves like a class instance, but only one instance of this class exists. [redacted] new instances can be created using the `new` keyword.

No

```
scala> object TestObject{
     |     private def fact_rec(n : Int, result : Int) : Int = {
     |                 if (n==1)
     |                     result
     |                 else
     |                     fact_rec(n-1, result*n)
     |             }
     |
     |     def factorial(n : Int) : Int  = fact_rec(n,1);
     | }

scala> TestObject.factorial(5)
res0: Int = 120
```

Singleton objects are also used to write Scala applications by definining a `main` method.

Everything in Scala applications must go into a Class or an Object.

```
object TestObject{
    private def fact_rec(n : Int, result : Int) : Int = {
       if (n==1)
           result
       else
           fact_rec(n-1, result*n)
    }

    def factorial(n : Int) : Int  = fact_rec(n,1);

    def main(args : String[]) = {
        println(factorial(5))
    }
```

```
$ scalac TestObject.scala
$ scala TestObject
120
```

Another important use case of singleton objects are *companion* objects. A companion object is simply a singleton object with the same name as a class. Such objects can be used to store methods that would be declared *static* in Java or variables that are shared between different instances of the class. A companion object may also define an `apply` method, which allows the class to be 'called'. These are very useful to avoid the `new` keyword.

.apply = factory method

```
class Greeter(msg : String)  {
    def greet : Unit = println(msg);
}

object Greeter {
    def apply(msg : String) : Greeter = new Greeter(msg);
}

object TestGreeter {
    def main(args : String[]) = {
        val g = Greeter("hello");
        g.greet()
    }
}
```

No need to specify new Greeter because it's implicitly calling the object, Greeter.apply("hello")

```
$ scalac TestGreeter.scala
$ scala TestGreeter
"hello"
```

::(1, Nil) calls the apply method of the :: companion object which then returns the created list

4 Exercise

Write a class to represent rational numbers (i.e. fractions). Your class should support the multiplication and addition operation (using the * and + operator), which should both return *new* class instances. The resulting fraction shoudl be reduced to the lowest term. Make sure to only use `val` instance variables. To implement addition, you will need to rewrite the fractions so that the denominators are equal. You will find it useful to write functions to compute the greatest common divisor (GCD) and least common multiple (LCM). The GCD of two positive integers is

$$GCD(a,b) = \begin{cases} a & \text{if } b == 0 \\ GCD(b, a \bmod b) & \text{otherwise} \end{cases}$$

The LCM of two positive integers is

$$LCM(a,b) = \frac{a \cdot b}{GCD(a,b)}.$$

Write a companion object with corresponding *apply* method that allows you to create new rational numbers without the `new` keyword.

5 Inheritance

Inheritance works almost like in Scala. One difference is that the primary constructor of the parent class is automatically called and we can specify how parameters of the default constructor are passed on to the super constructor. Another difference is that in order to override an instance variable or concrete method of the parent class, we need to add the `override` keyword to the members of the child class. You can specify members as `final` to prevent it from being overriden.

```
class Rectangle (w : Double, h : Double) {
    def area = w * h
    val description = "Rectangle"
    override def toString = description + " , size : " + area
}

class Square(w : Double ) extends Rectangle(w, w) {
    override val description = "Square"
}
```

Be default, all classes inherit from `scala.AnyRef`. Implementing methods defined on `AnyRef`, such as `toString` in the previous example, also requires the override keyword.
Scala also supports abstract classes, like Java. Such classes cannot be instantiated. They contain abstract methods that need to be filled in by a child class. When implementing an abstract method, the `override` keyword is not required.

```
abstract class Shape {
    def area : Double // abstract method

    final val description : String = " Shape "

    override def toString = description + " , size : " + area
}

class Rectangle (w : Double , h : Double) extends Shape {
    def area = w * h // implement abstract method

    val description = " Rectangle "
}

class Square (w : Double) extends Rectangle (w, w) {
    override val description = " Square "
}
```

Just like in Java, inheritance enables polymorphism and dynamic binding. For example, consider a list of `Shape`s. The `toString` method defined in the abstract class `Shape` will use the `description` field and `area` method of the specific class that it is being called on.

```
scala > val x = new Rectangle (2 ,3)
x : Rectangle = Rectangle , size : 6.0

scala > val y = new Square (5)
y : Square = Square , size : 25.0

scala > val z = new Blob
z : Blob = Blob , size : 12.0

scala > val l : List [ Shape ] = List (x, y, z)
l : List [ Shape ] = List ( Rectangle , size : 6.0 , Square , size
    : 25.0 , Blob , size : 12.0)

scala > for (x <- l) println(x)
Rectangle 6.0
Square 25.0
Blob 12.0
```

5.1 Pattern Matching and Case Classes

Pattern matching, in its simplest form, works like the switch statement in Java.

```
scala > val x = 1

scala > val result : String = x match {
    case 1 => "A"
    case 2 => "B"
    case _ => "C"
}

scala> result
result: String = A
```

Like everything else, `match` is an expression. Depending on the value of x, the `match` expression evaluates to the string `"A"`, `"B"`, or `"C"`. The _ pattern is the default case that matches if none of the other cases do. So passing in 3 for x would return `"C"`. Unlike the `switch` statement in Java, the `match` expression will immediately evaluate to the expression on the right hand side of the `=>`. No `break` statement is necessary. Even though it doesn't matter for this simple example, the cases are checked one-by-one in the order in which they are specified. If the _ case is specified first, it will match everything.
Pattern matching is a lot more powerful than this. It allows automatic unpacking of tuples and lists, as well as capturing parts of these tuples and lists in a variable that can be used on the right hand side of the `=>`.

If no default case is specified and there is no match, an exception will be thrown: scala.MatchError