Pattern Matching

Simple and Compound Pattern Matches, Case Classes, Custom Extractors

Agenda

1. Simple Constant Patterns
2. Variable Loading and Binding
3. Guards
4. Options, Collections, Tuples, Try
5. Case Classes
6. Constructor Patterns
7. Type Patterns
8. Pattern matching in `vals` and `fors`
9. Partial functions reprise
10. Custom Extractors

Simple Constant Patterns

- In its simplest usage, Scala's `match` is like Java's `switch`:

```
def matchIt(x: Any): Unit = x match {
  case 10         => println("The number 10")
  case true       => println("This is the truth")
  case 2.0        => println("Double precision 2.0")
  case "hello"    => println("Well, hi there")
  case ()         => println("Unit")
  case _          => println("It's something else")
}

matchIt(10)       // The number 10
matchIt(2.0)      // Double precision 2.0
matchIt("hello")  // Well, hi there
matchIt(())       // Unit
matchIt(3)        // It's something else
```

- `case _` is the equivalent of `default` in Java
- There is no automatic fall-through, no break keyword, and no need for `{}`s (next `case` keyword is the demarcation between handlers)

`match` is an expression

- The cases are checked in order, and the first successful match consumes the match event
- Unlike Java, Scala's `match` is also an expression

```
def pair(s: String): String = s match {
  case "fish"     => "chips"
  case "bacon"    => "eggs"
  case "tea"      => "scones"
  case "horse"    => "carraige"
}

pair("fish")      // chips
pair("tea")       // scones

pair("universe")  // MatchError!
```

- Also this example has no default `case _` which means a non-matching result will throw a `MatchError` exception

Variable Loads

- As well as simply matching constants, Scala can load a value for use in the code block:

```
def opposite(s: String): String = s match {
  case "hot"          => "cold"
  case "full"         => "empty"
  case "cool"         => "square"
  case "happy"        => "sad"
  case anythingElse   => s"not $anythingElse"
}

opposite("cool")    // square
opposite("happy")   // sad
opposite("sane")    // not sane
```

- While the constants will be matched first, anything that doesn't match those constants will be put into the value `anythingElse` and the `s"not $anythingElse"` will be returned (this also makes the pattern match complete for all inputs)

Binding vs Loading

- A variable identifier in a pattern match is loaded with the value
- But there is an alternative, useful for multiple-matches and in other situations, binding:

```
def opposite2(s: String): String = s match {
  case "hot"          => "cold"
  case "full"         => "empty"
  case "cool"         => "square"
  case "happy"        => "sad"
  case inWord @ ("sane" | "edible" | "secure") => s"in$inWord"
  case anythingElse => s"not $anythingElse"
}
opposite2("happy")   // sad
opposite2("sane")    // insane
opposite2("edible")  // inedible
opposite("fish")     // not fish
```

- `("sane" | "edible" | "secure")` matches any of those words, and the `@` binds the result into `inWord`
- More generally, `@` binds the pattern match on the right of it to the variable on the left of it

Case Matters!

- Identifiers starting with lower case are treated as **variables**
- Identifiers starting with upper case are treated as **constants**

```
val MaxLimit = 10  // constants start with upper case
val minLimit = 1

def isALimit(x: Int) = x match {
  case MaxLimit => true  // constant match works as expected
  case _ => false
}

isALimit(10)  // true
isALimit(3)   // false
```

But!

```
def isALimit(x: Int) = x match {
  case MaxLimit => true  // constant match works as expected
  case minLimit => true  // this is treated as a load and will always match!
  case _ => false
}

isALimit(10)  // true
isALimit(1)   // true
isALimit(3)   // true!
```

- If you must use lower case constants, put them in backticks in the match:

```
def isALimit(x: Int) = x match {
  case MaxLimit => true
  case `minLimit` => true  // backticks make this work as constant match
  case _ => false
}

isALimit(10)  // true
isALimit(1)   // true
isALimit(3)   // false
```

Guards

- Anything on the left of the => is part of the pattern match, anything on the right is what to do
- `if` expressions can be used on the left of the =>:

```
def describeNumber(x: Int): String = x match {
  case 0                       => "zero"
  case n if n > 0 && n < 100   => "smallish positive"
  case n if n > 0              => "large positive"
  case n if n < 0 && n > -100 => "smallish negative"
  case _                       => "large negative"
}

describeNumber(-99)  // smallish negative
describeNumber(99)   // smallish positive
describeNumber(0)    // zero
describeNumber(101)  // large positive
describeNumber(-101) // large negative
```

- Remember that the first full match stops the attempt going any further

The Wrong Way to Guard

- It's easy to forget that the if goes before the =>

```
def badDescribeNumber(x: Int) = x match {
  case 0 =>                          "zero"
  case n => if (n > 0 && n < 100)  "smallish positive"
  case n => if (n > 0)             "large positive"
  case n => if (n < 0 && n > -100) "smallish negative"
  case _ =>                          "large negative"
}  // badDescribeNumber[](val x: Int) => Any!

badDescribeNumber(-99)  // ()
badDescribeNumber(99)   // smallish positive
badDescribeNumber(0)    // zero
badDescribeNumber(101)  // ()
badDescribeNumber(-101) // ()
```

- Remember, guards go on the left of the =>

Matching Options

```
def matchOption(o: Option[Int]) =  o match {
  case Some(n) if n > 10 => "It's a number above 10"
  case Some(_)          => "It's a number 10 or less"
  case None             => "No number given"
}

matchOption(Some(50))  // It's a number above 10
matchOption(Some(5))   // It's a number 10 or less
matchOption(None)      // No number given
```

- There are only two states for `Option`, `Some(x)` and `None`
- Can unpack variables from inside the option (and use them)
- This is common usage, but there are often more idiomatic ways of dealing with Option (e.g. `map`, `getOrElse`)

Matching Tuples

```
def matchTuple3(tup: (Int, Boolean, String)): String = tup match {
  case (1, flag, string) => s"a 1 followed by $flag and $string"
  case (i, true, "Fred") => s"a true Fred with int $i"
  case (a, b, c)         => s"Some other tuple int $a, flag $b, string $c"
}

matchTuple3((1, false, "Sally"))
// a 1 followed by false and Sally
matchTuple3((1, true, "Harry"))
// a 1 followed by true and Harry
matchTuple3((2, true, "Fred"))
// a true Fred with int 2
matchTuple3((2, false, "Fred"))
// Some other tuple int 2, flag false, string Fred
```

- `true` is a keyword, so there is no confusion about load or constant match there, it's a constant

Matching Lists

- For Lists specifically, you can use `::` (cons) notation for matches:

```
def matchList(xs: List[Int]): String = xs match {
  case 1 :: 2 :: rest => s"A 1, 2 list followed by $rest"
  case a :: b :: _ => s"A list of at least 2 items, starting with $a, $b"
  case a :: Nil => s"A single element list of $a"
  case Nil => "The empty list"
}

matchList(List(1,2,3))
// A 1, 2 list followed by List(3)
matchList(List(1,2))
// A 1, 2 list followed by List()
matchList(List(1,3,4))
// A list of at least 2 items, starting with 1, 3
matchList(List(4))
// A single element list of 4
matchList(Nil)
// The empty list
```

- very common to see `case head :: tail =>` in recursive functions

Other Collections

- You can do similar matches for other collections (but not with cons notation):

```
def matchSeq(xs: Vector[Int]): String = xs match {
  case 1 +: 2 +: rest => s"A 1, 2 vector followed by $rest"
  case Vector(a, b, _*) => s"A vector of at least 2 items, starting with $a, $b"
  case Vector(a) => s"A single element vector of $a"
  case Vector() => "The empty vector"
}
```

- `+:` stands in for `::`
- Can also use expansion operator `_*` to match remainder in "constructor" style
- And bindings, so `Vector(1, 2, rest @ _*) =>` is equivalent to `1 +: 2 +: rest =>`
- This syntax also works for `Lists` (but with `List` replacing `Vector` of course)

Matching Try

```
import scala.util._

def matchTry(t: Try[_]): String = t match {
  case Success(x) => s"It worked, result is $x"
  case Failure(e) => s"It failed with $e"
}

matchTry(Try(4/2))  // It worked, result is 2
matchTry(Try(4/0))  // It failed with java.lang.ArithmeticException: / by zero
```

- Other core libraries often have pattern match support too, like `Future`, `Either`, etc.

Case Classes

- When you define a `case class` you get a bunch of things, including pattern matching

```
case class Address(street: String, city: String, postCode: Option[String])
case class Person(name: String, phone: Option[String], address: Option[Address])
```

- Factory methods for easy construction

```
val harry = Person("Harry", None, Some(Address(
  "123 Little Whinging way", "Purley", Some("PN22 6RT")
)))

val sally = Person("Sally", Some("321-222-3344"), None)
```

- Built in useful default `toString`

```
harry
// Person(Harry,None,Some(Address(123 Little Whinging way,Purley,Some(PN22 6RT))))

sally
// Person(Sally,Some(321-222-3344),None)
```

Case Classes

- You also get... `equals` and `hashCode` that work

```
sally == harry                                          // false
sally == sally                                          // true
sally == Person("Sally", Some("321-222-3344"), None) // true
sally == Person("Sally", Some("321-234-3344"), None) // false

sally.hashCode                                          // -171467737
Person("Sally", Some("321-222-3344"), None).hashCode // -171467737
harry.hashCode                                          // 1544670842
```

- Public parametric fields

```
harry.name                // Harry
harry.address.map(_.city) // Some(Purley)
harry.phone               // None
sally.phone               // Some(321-222-3344)
```

Case Classes

- And, a `copy` method

```
val sally2 = sally.copy(address = harry.address, phone = Some("321-333-2211"))
// Person(Sally,Some(321-333-2211),
//   Some(Address(123 Little Whinging way,Purley,Some(PN22 6RT))))
val harry2 = harry.copy(phone = sally2.phone)
// Person(Harry,Some(321-333-2211),
//   Some(Address(123 Little Whinging way,Purley,Some(PN22 6RT))))
```

- `case classes` are immutable by default, but `copy` makes them easy to work with in a functional way
- And, you get pattern matching...

Compound Pattern Matches

```
def postCodeForHarry(person: Person) = person match {
  case Person("Harry", _, Some(Address(street, city, Some(postcode)))) =>
    println("Harry found with postcode")
    println(s"City $city")
    println(s"Street $street")
    postcode
  case _ => ""
}

postCodeForHarry(harry)  // PN22 6RT
postCodeForHarry(harry2) // PN22 6RT
postCodeForHarry(sally)  // ""
postCodeForHarry(sally2) // ""
```

- Mix and match constants, `case` patterns, `Options` and anything matchable
- Could also get the harry match as a whole in the above with:

```
  case harry @ Person("Harry", _, Some(Address(street, city, Some(postcode)))) =>
```

- Because they look like the constructors for case classes, these are called *constructor* patterns

Typed Pattern Matches

```
def describeType(x: Any) = x match {
  case i: Int if i > 0 => s"Int ${i * i}"
  case d: Double => s"Double $d"
  case s: String => s"String ${s.reverse}"
  case p: Person => s"Person, name = ${p.name}"
  case _ => "Some other type"
}

describeType(3)       // Int 9
describeType(3.4)     // Double 3.4
describeType("Hello") // String olleH
describeType(harry)   // Person Harry
describeType(true)    // Some other type
```

- Once matched, the variable is typed on both the left and right of the =>
- This is idiomatic and favored over the form:

```
val s: Any = "Hello"
if (s.isInstanceOf[String]) {
  s.asInstanceOf[String].reverse
}
```

Beware Type Erasure!

```
def withIntStringMap(x: Any): Int = x match {
  case m: Map[Int, String] => m.head._1 * m.head._1
  case _ => 0
}

// Warning: non-variable type argument Int in type pattern
// scala.collection.immutable.Map[Int,String] (the underlying of Map[Int,String])
// is unchecked since it is eliminated by erasure
//  case m: Map[Int, String] => m.head._1 * m.head._1
//          ^
```

- Scala will match the `Map` vs not, but will **believe** you on the inner erased types, so you can get:

```
withIntStringMap(Map(2 -> "two"))  // 4 - as expected
withIntStringMap(List(2))          // 0 - not a match
withIntStringMap(Map("One" -> 1))  // ClassCastException!
```

- The safe way to match erased type parameters is `case m: Map[_, _]`
- Alternatively, type-tags can be used - see advanced course

`val` and pattern matching

- `val` is a pattern-match

```
val Person(name, phone, Some(Address(_, _, postCode))) = harry
// name: String = Harry
// phone: Option[String] = None
// postCode: Option[String] = Some(PN22 6RT)
```

- Which means it can fail...

```
val Person(name2, phone2, Some(Address(_, _, postCode2))) = sally
// scala.MatchError: Person(Sally,Some(321-222-3344),None)
```

- This fails because sally has no Address recorded
- Be aware of this if you use a val with a pattern match - you may get a match error

`for` and pattern matching

- Generators in a `for` block are also pattern matches

```
val numbersMap = Map(1 -> "one", 2 -> "two", 3 -> "three")

for ((k, v) <- numbersMap) {  // unpack the key -> value tuples
  println(s"$k is $v")
}
```

- A non-match will just short-circuit the `for` so there's no exception if no match

```
val people = List(harry, harry2, sally, sally2)

for {
  Person(name, phone, _) <- people
  if phone.isDefined
} yield name -> phone.get
// List((Harry,321-333-2211), (Sally,321-222-3344), (Sally,321-333-2211))
```

Partial functions and pattern matches

- Remember a `PartialFunction[T, R]` extends `Function1[T, R]`
- This means that a partial function (which is a pattern match) can substitute for any `Function1`

```
numbersMap.map {
  case (1, w) => s"It's 1 and the word is $w"
  case (k, v) => s"Not 1 but ($k, $v)"
}
//  List(It's 1 and the word is one, Not 1 but (2, two), Not 1 but (3, three))
```

- If you use a partial function that is incomplete in a function expecting a `Function1`, you may end up with a `MatchError`

Sealed Class Hierarchies

- Sometimes you want to control what different types may be in a hierarchy
- The `sealed` keyword gives you this, and pattern matches can then give warnings about incomplete matches

```
sealed class AccountType
case object Checking extends AccountType
case object Savings extends AccountType


def checking(at: AccountType): Boolean = at match {
  case Checking => true
}
// Warning:(6, 43) match may not be exhaustive.
// It would fail on the following inputs: AccountType(), Savings
// def checking(at: AccountType): Boolean = at match {
```

- `sealed` means that the only sub-types of the sealed class or trait must be defined in the same source file

Extractors and Unapply

- How does it all work?

```
case class Person(first: String, last: String, age: Int)

val p1 = Person("Fred", "Frederickson", 28)

Person.unapply(p1)
// res1: Option[(String, String, Int)] = Some((Fred,Frederickson,28))
```

- `unapply` is auto-generated for case classes in the companion object

```
val xs = List(1,2,3,4)

List.unapplySeq(xs)
// res3: Some[List[Int]] = Some(List(1, 2, 3, 4))
```

- `unapplySeq` allows for matching repeated, var-arg matches in collections
- And we can write our own

Custom Extractors

```
val coordsStr = "-121.432, 34.002"

object Coords {
  def unapply(coordsStr: String): Option[(Double, Double)] = Try {
    val fields = coordsStr.split(",").map(_.trim.toDouble)
    (fields(0), fields(1))
  }.toOption
}

coordsStr match {
  case Coords(x, y) =>
    println(s"x = $x")
    println(s"y = $y")
}
// x = -121.432
// y = 34.002
```

Custom Seq Extractors

```
object CoordSeq {
  def unapplySeq(coordsStr: String): Option[Seq[Double]] = Try {
    coordsStr.split(",").toList.map(_.trim.toDouble)
  }.toOption
}

coordsStr match {
  case CoordSeq(c @ _*) =>
    c foreach println
}
// -121.432
// 34.002

coordsStr match {
  case CoordSeq(x, y, _*) =>
    println(x)
    println(y)
}
// -121.432
// 34.002
```

Exercises for Module 12

- Find the `Module12` class and follow the instructions to make the tests pass
- `Module12` is under `module12/src/test/scala/koans`, but there are other classes in that source file as well (part of the testing)