Pattern Matching

Scala training, 2016

Outline

Recap

Questions:

- what's a function?

Recap

Questions:

- what's a function?
- functions vs objects?

Recap

Questions:

- what's a function?
- functions vs objects?
- *apply*?

Recap

Questions:

- what's a function?
- functions vs objects?
- *apply*?
- instruction vs expression?

Recap

Questions:

- what's a function?
- functions vs objects?
- *apply*?
- instruction vs expression?
- what are case classes?

Recap

Questions:

- what's a function?
- functions vs objects?
- *apply*?
- instruction vs expression?
- what are case classes?
- why are case classes special?

Outline

Switch on steroids

```
val e = ...
val x = e match {
  case pattern1 => result1
  case pattern2 => result2
  case _ => result3
}
```

Notice:

- the *match* keyword - mandatory
- the cases - not just simple constants (we'll cover them shortly)
- the wildcard (_) case - matches anything - not mandatory
- that the whole construct is an *expression*

Switch on steroids (2): Pattern match expressions

```
val e = ...
val x = e match {
  case pattern1 => result1
  case pattern2 => result2
  case _ => result3
}
```

A few more things:

- the type of each pattern must be a (sub)type of *e*
- first pattern that matches evaluates the corresponding result
- if there is a match
  - the result is the value of the entire expression
  - the next alternatives are *not* evaluated (unlike *switch*)
- if nothing matches, a `MatchError` is thrown

Outline

Patterns: constants

```
val y = x match {
  case 2 => "an integer"
  case true => "the true boolean"
  case "hello" => "the greeting string"
  case MySingleton => "a singleton object"
}
```

A constant pattern matches only itself.

Patterns: constants

```
val y = x match {
  case 2 => "an integer"
  case true => "the true boolean"
  case "hello" => "the greeting string"
  case MySingleton => "a singleton object"
}
```

A constant pattern matches only itself.

Q: what is the return type of the above expression?

Patterns (2): the wildcard

Matches anything:

```
val y = x match {
  case 1 => "the only one"
  case _ => "some other value"
}
```

Patterns (2): the wildcard

Matches anything:

```
val y = x match {
  case 1 => "the only one"
  case _ => "some other value"
}
```

Also works in "deeper" matches (we'll talk shortly).

Patterns (3): variables

- also matches anything
- but also *binds* the value to the variable...
- ...even in "deeper" matches (see tuples next)

```
val y = x match {
  case 1 => "the only one"
  case someValue => "some other value: " + someValue
}
```

Patterns (4): tuples

Recap: how does a tuple look like?

Patterns (4): tuples

Recap: how does a tuple look like?

Matching on tuples example:

```
val y = x match {
  case (1, 1) => "the only ones"
  // variable pattern in tuple
  case (v, 2) => v + " paired with a 2"
  // deep match with wildcards and a variable
  case (_, _, (v, 5)) =>
    "deep tuple with " + v " and 5 in the nested pair"
}
```

Outline

Patterns (5): constructors

Matching **case classes**:

```
trait List
case object Nil extends List
case class Cons(head: Int, tail: List) extends List

val y = x match {
  case Nil => "the empty list"
  case Cons(h, Nil) => "a list with one element"
  case Cons(h, Cons(sh, _)) =>
    "a list starting with " + h + " and " + sh
}
```

Notice:

- matching on case objects = singletons
- variable bindings
- use of wildcards
- all within "deep" matching

Patterns (6): sequences and lists

```
val y = x match {
  case List(1, _, _, _) =>
    "list of 4 elements, first is 1"
  case List(2, _*) =>
    "list of at least 1 element, starting with 2"
  case x :: Nil => "a list with only one element: " + x
  case 1 :+ _ => "nonempty list starting with 1"
  case Nil => "empty list"
}
```

Notice

- the use of List constructors
- the vararg (_*) constructor of List
- the prepend operators (:: and :+)
- the use of wildcards and variables

Outline

Patterns (7): specifying types

Match only the types that you want:

```
val y = x match {
  case l: List => "a list"
  case _: Vector => "a vector"
  case _ => "something else"
}
```

Notice

- every typed case has either a wildcard or a variable name
- the non-typed wildcard can still match (when?)

Patterns (7.1): "generic" types

```
// just read List[Int] as List<Int> in Java
val x: List[Int] = List[Int](1,2,3)
val y = x match {
  case l: List[Double] => "the double list"
  case Nil => "empty list"
  case _ => "something else"
}
```

Q: What's the result of this match?

Patterns (7.1): "generic" types

```
// just read List[Int] as List<Int> in Java
val x: List[Int] = List[Int](1,2,3)
val y = x match {
  case l: List[Double] => "the double list"
  case Nil => "empty list"
  case _ => "something else"
}
```

Q: What's the result of this match?

A: "the double list".

Patterns (7.1): "generic" types

```
// just read List[Int] as List<Int> in Java
val x: List[Int] = List[Int](1,2,3)
val y = x match {
  case l: List[Double] => "the double list"
  case Nil => "empty list"
  case _ => "something else"
}
```

Q: What's the result of this match?

A: "the double list".

Q: Why?

Patterns (7.1): "generic" types

```
// just read List[Int] as List<Int> in Java
val x: List[Int] = List[Int](1,2,3)
val y = x match {
  case l: List[Double] => "the double list"
  case Nil => "empty list"
  case _ => "something else"
}
```

Q: What's the result of this match?

A: "the double list".

Q: Why? Hint: how are generics/type params stored in Java?

Patterns (7.1): "generic" types

```
// just read List[Int] as List<Int> in Java
val x: List[Int] = List[Int](1,2,3)
val y = x match {
  case l: List[Double] => "the double list"
  case Nil => "empty list"
  case _ => "something else"
}
```

Q: What's the result of this match?

A: "the double list".

Q: Why? Hint: how are generics/type params stored in Java?

A: Trick question - they're *not stored*!

Patterns (7.1): "generic" types

```
// just read List[Int] as List<Int> in Java
val x: List[Int] = List[Int](1,2,3)
val y = x match {
  case l: List[Double] => "the double list"
  case Nil => "empty list"
  case _ => "something else"
}
```

Q: What's the result of this match?

A: "the double list".

Q: Why? Hint: how are generics/type params stored in Java?

A: Trick question - they're *not stored*!

Type erasure

- a "feature" added in Java 5 along with generics...
- ...to keep backward compatibility with Java 1

Patterns (7.2): "generics" in arrays

```
// Recall Any is the ultimate supertype
// Also, an Array[Int] maps onto an int[] in Java
val x: Any = Array(1,2,3)
val y = x match {
  case l: Array[Double] => "array of double"
  case _: Array[Int] => "array of integers"
  case _ => "something else"
}
```

Q: What's the result of this match?

Patterns (7.2): "generics" in arrays

```
// Recall Any is the ultimate supertype
// Also, an Array[Int] maps onto an int[] in Java
val x: Any = Array(1,2,3)
val y = x match {
  case l: Array[Double] => "array of double"
  case _: Array[Int] => "array of integers"
  case _ => "something else"
}
```

Q: What's the result of this match?

A: "array of integers".

Patterns (7.2): "generics" in arrays

```
// Recall Any is the ultimate supertype
// Also, an Array[Int] maps onto an int[] in Java
val x: Any = Array(1,2,3)
val y = x match {
  case l: Array[Double] => "array of double"
  case _: Array[Int] => "array of integers"
  case _ => "something else"
}
```

Q: What's the result of this match?

A: "array of integers".

Q: Why?

Patterns (7.2): "generics" in arrays

```
// Recall Any is the ultimate supertype
// Also, an Array[Int] maps onto an int[] in Java
val x: Any = Array(1,2,3)
val y = x match {
  case l: Array[Double] => "array of double"
  case _: Array[Int] => "array of integers"
  case _ => "something else"
}
```

Q: What's the result of this match?

A: "array of integers".

Q: Why?

Q: Hint: how are array types stored in Java?

Patterns (7.2): "generics" in arrays

```
// Recall Any is the ultimate supertype
// Also, an Array[Int] maps onto an int[] in Java
val x: Any = Array(1,2,3)
val y = x match {
  case l: Array[Double] => "array of double"
  case _: Array[Int] => "array of integers"
  case _ => "something else"
}
```

Q: What's the result of this match?

A: "array of integers".

Q: Why?

Q: Hint: how are array types stored in Java?

A: The type *is stored* along with the array itself!

Patterns (8): multiple patterns in one case

Use the pipe | operator:

```
val y = x match {
  case patern1 | pattern2 | ... => result
  ...
}
```

Patterns (9): bind variables

Use @ to give names to (parts of) a pattern:

```
trait List
case object Nil extends List
case class Cons(head: Int, tail: List) extends List

val y = x match {
  case n@Nil =>
    "the empty list: " + n
  case c@Cons(h, Nil) =>
    "a list with one element:" + c
  case Cons(h, p@Cons(sh, _)) =>
    "non empty list with tail = " + p
}
```

Notice

- names can be given to *nested* parts
- captured (named) parts can be then used in the result

Outline

Patterns (10): conditional patterns

Use *if*:

```
val x: List[Int] = List(1,2,3)
val y = x match {
  case List(_, h, _) if h % 2 == 0 =>
    "List with 3 elements with second element even"
  case _ => "something else"
}
```

Notice

- you must be able to use parts of the pattern in the condition...
- ...so name them!

Patterns in "regular" code: exceptions

```
try {
  // code
} catch {
  case e: IllegalArgumentException =>
    println("illegal arg exception")
  case _: Exception =>
    println("something else")
} finally { // finally is optional, as in Java
  // code
}
```

A bit more on catching exceptions:

- there are no checked exceptions in Scala

Patterns in "regular" code: exceptions

```
try {
  // code
} catch {
  case e: IllegalArgumentException =>
    println("illegal arg exception")
  case _: Exception =>
    println("something else")
} finally { // finally is optional, as in Java
  // code
}
```

A bit more on catching exceptions:

- there are no checked exceptions in Scala

  Q: checked exceptions?

Patterns in "regular" code: exceptions

```
try {
  // code
} catch {
  case e: IllegalArgumentException =>
    println("illegal arg exception")
  case _: Exception =>
    println("something else")
} finally { // finally is optional, as in Java
  // code
}
```

A bit more on catching exceptions:

- there are no checked exceptions in Scala
  Q: checked exceptions?
- the *catch* block is a *match* expression...

Patterns in "regular" code: exceptions

```
try {
  // code
} catch {
  case e: IllegalArgumentException =>
    println("illegal arg exception")
  case _: Exception =>
    println("something else")
} finally { // finally is optional, as in Java
  // code
}
```

A bit more on catching exceptions:

- there are no checked exceptions in Scala
  Q: checked exceptions?
- the *catch* block is a *match* expression...
- ...so it returns a value!

Patterns in "regular" code: exceptions

```
try {
  // code
} catch {
  case e: IllegalArgumentException =>
    println("illegal arg exception")
  case _: Exception =>
    println("something else")
} finally { // finally is optional, as in Java
  // code
}
```

A bit more on catching exceptions:

- there are no checked exceptions in Scala
  Q: checked exceptions?
- the *catch* block is a *match* expression...
- ...so it returns a value!
- the entire *try/catch* is an expression

Patterns in "regular" code (2): generators

Generators are based on pattern matchers:

```
val l = List((1,2), (2,3), (3,5))

// named/captured variable
for (pair <- l) yield (pair._1, pair._2 + 1)

// captured variables + tuple match
for ((first, second) <- l) yield (first, second + 1)

// wildcard, captured variable, tuple and conditional
for ((_, second) <- l if second > 4) yield second * 2
```

- all pattern matchers are available in generators
- including the handy ones with operators e.g. ::, :+

Patterns in "regular" code (3): value definitions

Teaser: Python sequence unpacking

```
a, (b, c) = (1, (2, 3))
# a = 1, b = 2, c = 3
```

Patterns in "regular" code (3): value definitions

Teaser: Python sequence unpacking

```
a, (b, c) = (1, (2, 3))
# a = 1, b = 2, c = 3
```

Scala too, via pattern matching:

```
val (a, (b, c)) = (1, (2, 3))
// a = 1, b = 2, c = 3
```

Patterns in "regular" code (3): value definitions

Teaser: Python sequence unpacking

```
a, (b, c) = (1, (2, 3))
# a = 1, b = 2, c = 3
```

Scala too, via pattern matching:

```
val (a, (b, c)) = (1, (2, 3))
// a = 1, b = 2, c = 3
```

but Scala rules:

```
val list = List((1,2), (2,3), (3,5))

val head :: tail = list
println(head) // (1, 2)
println(tail) // List ((2,3), (3,5))
```

Pattern matching is powerful!

Patterns in "regular" code (4): partial functions

A bit advanced:

Patterns in "regular" code (4): partial functions

A bit advanced:

```
val list = List(1,2,4)
val x = list map {
  case v if v % 2 == 0 => v + " even"
  case 1 => "the one"
}
// List(the one, 2 even, 4 even)
```

Patterns in "regular" code (4): partial functions

A bit advanced:

```
val list = List(1,2,4)
val x = list map {
  case v if v % 2 == 0 => v + " even"
  case 1 => "the one"
}
// List(the one, 2 even, 4 even)
```

Notice

- the special syntax for an anonymous "function"
  - nothing special - just pattern match

Patterns in "regular" code (4): partial functions

A bit advanced:

```
val list = List(1,2,4)
val x = list map {
  case v if v % 2 == 0 => v + " even"
  case 1 => "the one"
}
// List(the one, 2 even, 4 even)
```

Notice

- the special syntax for an anonymous "function"
  - nothing special - just pattern match
- the "function" is *not* applicable for any Int!

Patterns in "regular" code (4): partial functions

A bit advanced:

```
val list = List(1,2,4)
val x = list map {
  case v if v % 2 == 0 => v + " even"
  case 1 => "the one"
}
// List(the one, 2 even, 4 even)
```

Notice

- the special syntax for an anonymous "function"
  - nothing special - just pattern match
- the "function" is *not* applicable for any Int!
- it's not a proper "function" in the math sense...

Patterns in "regular" code (4): partial functions

A bit advanced:

```
val list = List(1,2,4)
val x = list map {
  case v if v % 2 == 0 => v + " even"
  case 1 => "the one"
}
// List(the one, 2 even, 4 even)
```

Notice

- the special syntax for an anonymous "function"
  - nothing special - just pattern match
- the "function" is *not* applicable for any Int!
- it's not a proper "function" in the math sense...
- ...and not even in the Scala sense:
  - it's not a *Function1[Int, String]*
  - but a *PartialFunction[Int, String]*

Recap

From previous episodes

Pattern matching

- Basic syntax
- Basic patterns
- Complex patterns
- Captures and type specifiers
- And more power

Patterns in "regular" code