TOUR OF SCALA

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 and it can likewise be used in place of a series of if/else statements.

Syntax

A match expression has a value, the `match` keyword, and at least one `case` clause.

```
import scala.util.Random

val x: Int = Random.nextInt(10)

x match {
  case 0 => "zero"
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
```

The `val x` above is a random integer between 0 and 10. `x` becomes the left operand of the `match` operator and on the right is an expression with four cases. The last case `_` is a "catch all" case for any other possible `Int` values. Cases are also called *alternatives*.

Match expressions have a value.

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
```

```
  case _ => "other"
}
matchTest(3)  // prints other
matchTest(1)  // prints one
```

This match expression has a type String because all of the cases return String. Therefore, the function `matchTest` returns a String.

Matching on case classes

Case classes are especially useful for pattern matching.

```
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
```

`Notification` is an abstract super class which has three concrete Notification types implemented with case classes `Email`, `SMS`, and `VoiceRecording`. Now we can do pattern matching on these case classes:

```
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: Ar

println(showNotification(someVoiceRecording))  // prints You received a Voice Record
```

The function `showNotification` takes as a parameter the abstract type `Notification` and matches on the type of `Notification` (i.e. it figures out whether it's an `Email`, `SMS`, or `VoiceRecording`). In the `case Email(sender, title, _)` the fields `sender` and `title` are used in the return value but the `body` field is ignored with `_`.

Pattern guards

Pattern guards are simply boolean expressions which are used to make cases more specific. Just add `if <boolean expression>` after the pattern.

```
def showImportantNotification(notification: Notification, importantPeopleInfo: Seq[S
  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 showNotif
  }
}

val importantPeopleInfo = Seq("867-5309", "jenny@gmail.com")

val someSms = SMS("123-4567", "Are you there?")
val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")
val importantEmail = Email("jenny@gmail.com", "Drinks tonight?", "I'm free after 5!"
val importantSms = SMS("867-5309", "I'm here! Where are you?")

println(showImportantNotification(someSms, importantPeopleInfo)) // prints You got a
println(showImportantNotification(someVoiceRecording, importantPeopleInfo)) // print
println(showImportantNotification(importantEmail, importantPeopleInfo)) // prints Yo

println(showImportantNotification(importantSms, importantPeopleInfo)) // prints You
```

In the `case Email(sender, _, _) if importantPeopleInfo.contains(sender)`, the pattern is matched only if the `sender` is in the list of important people.

Matching on type only

You can match on the type like so:

```
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
}
```

`def goIdle` has a different behavior depending on the type of `Device`. This is useful when the case needs to call a method on the pattern. It is a convention to use the first letter of the type as the case identifier (`p` and `c` in this case).

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"
}
```

This is useful for pattern matching because we don't need a "catch all" case.

Notes

Scala's pattern matching statement is most useful for matching on algebraic types expressed via case classes. Scala also allows the definition of patterns independently of case classes, using `unapply` methods in extractor objects.

More resources

- More details on match expressions in the Scala Book

← previous next →

Contributors to this page:

 ckipp01  BaeJi77  mlachkar  mehuled  ashawley  puorc

 asakaev  bkellerman  Michael Kinsey  heathermiller