UNIVERSIDAD COMPLUTENSE DE MADRID

ntic master School

Scala

Pattern Matching

Noviembre de 2023

Agenda

- **Pattern Matching**
- **Valores y generadores con patrones**

1 Pattern matching

Expresiones *match*


```
expr match {
 case pattern1 => result1
 case pattern2 => result2
}
```

- El valor generado de la evaluación de una expresión es comparado con un patrón.
- Existe un gran conjunto de “patrones” para comparar.
- *Match* es una expresión, por lo que va a devolver un valor.
- Si no existe coincidencia con ningún patrón se lanza una excepción *MatchError*.

Comparando con patrones


```
case pattern => result
```

- *case* declara una comparación con el patrón.
- *pattern* es la definición del patrón contra el que se compara el valor. Existen varios tipos de patrones. *pattern* no es código arbitrario.
- *result* es una expresión.
- Si se cumple el *pattern*, *result* se evalúa y devuelve su resultado.

Patrón comodín

```
def whatTimeIsIt(any: Any): String = any match {
 case _ => s"$any is no time!"
}
```

```
scala> def whatTimeIsIt(any: Any): String = any match {
     |  case _ => s"$any is no time!"
     | }
whatTimeIsIt: (any: Any)String

scala> whatTimeIsIt("foo")
res13: String = foo is no time!
```

- Se usa _ como comodín para dar por buena la coincidencia del patrón aceptando cualquier valor (*any*) sin ningún tipo de restricción.
- El uso del comodín como última alternativa es un buen mecanismo para evitar *MatchErrors*.

Patrón variable

```
def whatTimeIsIt(any: Any): String = any match {
 case x => s"$x is no time!"
}
```

```
scala> def whatTimeIsIt(any: Any): String = any match {
     |  case x => s"$x is no time!"
     | }
whatTimeIsIt: (any: Any)String

scala> whatTimeIsIt("foo")
res14: String = foo is no time!
```

- Se usa un identificador, en el ejemplo *x*, para:
  - Buscar una coincidencia con todo.
  - Capturar el valor.
- El patrón variable es muy útil combinado con otros patrones.
- A diferencia del patrón comodín, el patrón variable nos permite mantener la referencia al valor que se le pasa al Match.

Patrón tipado

```
def whatTimeIsIt(any: Any): String = any match {
 case time: Time => s"It is ${time.hours}:${time.minutes}"
 case _ => s"$any is no time!"
}
```

```
scala> def whatTimeIsIt(any: Any): String = any match {
     |     case time: Time => s"It is ${time.hours}:${time.minutes}"
     |     case _ => s"$any is no time!"
     |   }
whatTimeIsIt: (any: Any)String

scala> whatTimeIsIt(Time(10, 20))
res15: String = It is 10:20
```

- Se usa la anotación de tipo sólo para buscar coincidencia con determinados tipos: *case variable: <tipo variable>*. Este patrón hace uso del patrón variable.
- Los patrones tipados necesitan ser complementados con el patrón comodín o variable (para evitar *MatchError*).

Patrón constante

```scala
def whatTimeIsIt(any: Any): String = any match {
 case "12:00" => s"High noon"
 case _ => s"$any is no time!"
}
```

```
scala> def whatTimeIsIt(any: Any): String = any match {
     |     case "12:00" => s"Hih noon"
     |     case _ => s"$any is no time!"
     |   }
whatTimeIsIt: (any: Any)String

scala> whatTimeIsIt("12:00")
res16: String = Hih noon

scala> whatTimeIsIt("10:00")
res17: String = 10:00 is no time!
```

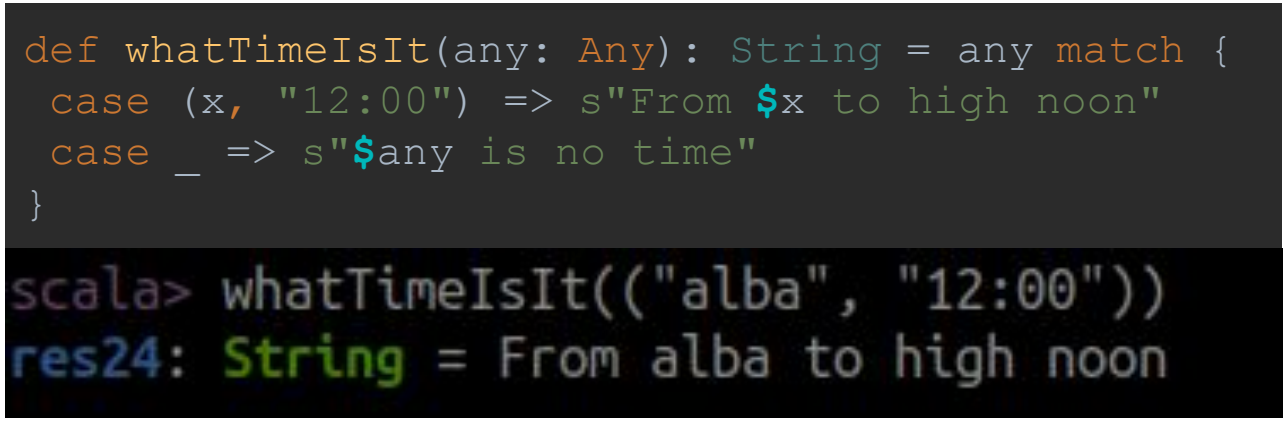
- Se usa un valor fijo para buscar la coincidencia.
- Se pueden usar *identificadores estables* (*Stable Identifiers*)

Identificadores estables


```scala
val highNoon = "12:00"
def whatTimeIsIt(any: Any): String = any match {
 case `highNoon` => "High noon"
 case TestData.munich => "Not a time"
 case _ => s"$any is no time!"
}
```

- Los identificadores estables son:
  - Literales
  - Identificadores para vals u objetos singleton que empiecen por:
    - Mayúscula (singletons)
    - Minúscula entre comilla invertida `

Patrón tupla



- Se puede usar la sintaxis de tuplas para buscar coincidencia y descomponer las tuplas.
- El patrón tuplas se puede componer con otros patrones, por ejemplo: el patrón la constante o variable

Patrón constructor


```scala
def whatTimeIsIt(any: Any): String = any match {
 case Time(12, 0) => s"High noon"
 case Time(12, x) => s"it is $x minutes past 12"
 case _ => s"$any is no time"
}
```

```
scala> whatTimeIsIt(Time(12, 15))
res0: String = it is 15 minutes past 12
```

- Se puede usar la sintaxis de constructor para comparar y descomponer *case classes*.
- El patrón constructor se puede componer con otro patrones.
- Se puede construir estructuras anidadas de varios niveles.

Patrón secuencia


```scala
def matchSeq[A](seq: Seq[A]): String = seq match {
 case Seq(1, 2, 3) => "1 to 3"
 case x +: Nil => s"Only element $x"
 case    :+ x => s"Last element is $s"
 case Nil => "Empty sequence"
}
```

```
scala> matchSeq(1 to 3)
res6: String = 1 to 3

scala> matchSeq(Vector(1))
res7: String = Only element 1

scala> matchSeq(Array(1, 2, 3, 4))
res8: String = Last element is 4
```

- Se usa el constructor de secuencia (Seq), los operadores de añadir y anteponer para coincidir y descomponer secuencias.

Patrón combinados


```
def whatTimeIsIt(any: Any): String = any match {
 case "00:00" | "12:00" => "midnight or high noon"
 case Nil => "Empty sequence"
}
```

- Se usa `|` para combinar como alternativas de coincidencia varios patrones.

Patrones entrelazados


```
def whatTimeIsIt(any: Any): String = any match {
 case time @ Time( , 0) => s"$time with 0 minutes"
 case time @ Time(_, m) => s"$time with $m
minutes"
 case Nil => s"$any is no time"
}
```

- Se usa @ para atar una variable a un patrón.
- Útil si un patrón combinado como el patrón constructor o secuencia se quiere tener el valor completo además de las partes.

Patrones con guardias

```scala
def isAfternoon(any: Any) = any match {
 case Time(h, m) if h >= 12 => s"Yes, it is ${h-12}:$m pm"
 case Time(h, m) => s"No, it is $h:$m am"
 case _ => s"$any is no time"
}
```

```
scala> isAfternoon(Time(10, 30))
res9: String = No, it is 10:30 am

scala> isAfternoon(Time(16, 20))
res10: String = Yes, it is 4:20 pm
```

- La composición de patrones permite tener gran control de lo que queremos hacer coincidir.
- Mecanismo extra: guardias:
  - se definen usando la palabra reservada *if*

Ejercicio: Usa match-expression

- Definición:
  - Un viaje entre dos estaciones es breve si:
    - existe una conexión con un mismo tren
    - hay como mucho una estación entre las dos estaciones dadas.
- Añade un método `isShortTrip` a `JourneyPlanner`:
  - añade los parámetros `from` y `to` de tipo `Station`
  - devuelve `true` si existe un `Train` en trains donde entre las `stations` exiten `from` y `to` con como mucho una única `Station` entre ambas.
  - Hint: usa los métodos de colecciones `exists` y `dropWhile` y un `match expression` con el patrón de secuencia.

2 Valores y generadores con patrones

Patrones y excepciones


```
def toInt(s: String): Int =
 try {
   s.toInt
 } catch {
   case _: NumberFormatException => 0
 }
```

- En Scala no es necesario verificar las excepciones.
- Pero si se requiere, se usa la expresión *try-catch* para capturar la excepción.
- En la cláusula catch se puede tener una o más alternativas de coincidencia.
- Cláusula *finally* es opcional.

Patrones y *vals*


```
scala> val (morning, highNoon) = Time(6) -> Time(12)
morning: Time = Time(6,0)
highNoon: Time = Time(12,0)

scala> val (morning, _) = Time(6) -> Time(12)
morning: Time = Time(6,0)

scala> val midnight = Time()
midnight: Time = Time(0,0)

scala> val (morning, `midnight`) = Time(6) -> Time(12)
scala.MatchError: (Time(6,0),Time(12,0)) (of class scala.Tuple2)
  ... 31 elided
```

- Se pueden definir *vals* usando patrones.

Patrones en generadores


```
for (keyAndValue <- Vector(1->"a", 2->"b"))
 yield keyAndValue._1 + keyAndValue._2

for ((key, value) <- Vector(1->"a",
2->"b"))
 yield key + value
```

- Los patrones pueden simplificar la manera de acceder a los elementos de un generador de tuplas.
- Descompone la tupla en elementos con nombre.

Ejercicio: Usa patterns

- Usa las el pattern tupla para acceder a cada campo de la tupla en el método `stopsAt`de la clase JourneyPlanner