**Pattern Matching**

## **Índice**

* Sintaxis
* Matching en clases Case
* Clases Sealed
* Pattern guards
* Matching de tipos
* Tipos de pattern
* Tipo Option

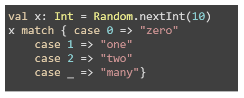
## **Sintaxis**

Pattern matching es un mecanismo que se utiliza, sobretodo, en la programación funcional, y sirve para comparar un valor con diferentes patrones.

Esta función es similar a la utilización de switch en Java. Además, es posible usar el patrón match en lugar de utilizar una sentencia if-else.

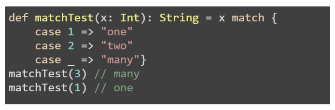
Se deben contemplar todas las soluciones posibles, y en caso de no poder hacerlo, hay que definir una solución genérica para cuando no haya matching con ninguna opción de nuestra lista. Esto debe hacerse, puesto que si no se encuentra una solución, Scala devolverá un error de tipo scala.MatchError.

En el siguiente ejemplo, podemos ver cómo se utiliza la función match:



El valor x es un entero aleatorio entre 0 y 10. Este se convierte en el operador izquierdo, mientras que el operador derecho es la función match con los cuatro casos. El último caso se usa cuando no coincide ninguno de los casos anteriores.

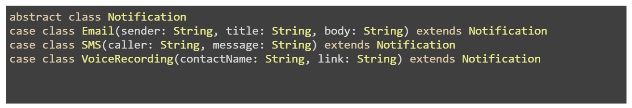
Las expresiones Match tienen un valor:



Esta expresión Match tiene un tipo String porque todos los casos devuelven String. Por lo tanto, la función matchTest devuelve también String.

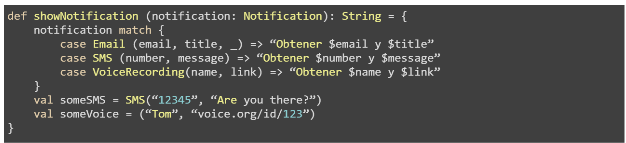
## **Matching en clases Case**

Las clases Case son especialmente útiles para pattern matching:



La clase Notificación es una superclase que tiene tres tipos concretos de notificación implementados con las clases Case Emails, SMS y VoiceRecording.

Tras el ejemplo anterior, ahora podemos hacer pattern matching en estas clases case:



Los siguientes ejemplos muestran cómo llamar a la clase creada anteriormente.

println(showNotification (someSms)) //Obtener 12345 y Are you there?

println(showNotification (someVoice)) //Obtener Tom y voice.org/id/123

La función showNotification se da cuenta de si se trata de un correo electrónico, un SMS o una grabación de voz.

En el caso de case Email (email, title, \_) los campos email y title se usan en el valor de retorno, pero el campo cuerpo se ignora con \_.

## **Clases Sealed**

Cuando se trabaja con pattern matching, es posible que no exista un valor por defecto para cubrir el resto de posibilidades no contempladas, por lo que es posible que se puedan generar errores.

Una solución para evitar este problema sería utilizar las clases Sealed. Estas clases utilizan la ayuda del compilador para poder detectar combinaciones que no se contemplen en el matching.

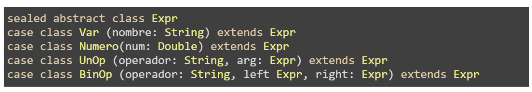
Una clase Sealed no puede tener nuevas subclases agregadas, excepto las que están en el mismo archivo.

El compilador se encargará de comprobar las clases que heredan de una clase sealed.

Si estas subclases no están contempladas para hacer pattern matching, el compilador marcará las combinaciones de patrones que faltan con un mensaje de advertencia.

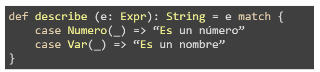
Para poder implementar una clase sellada, se debe utilizar la palabra reservada “sealed” delante de la palabra “class”.

El siguiente ejemplo muestra la implementación de una clase Sealed:

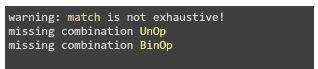


Como puede apreciarse, solo es necesario incluir la palabra reservada “sealed” en la clase padre. Las clases que extienden de la clase sealed no tienen que implementar nada adicional.

Tras definir la clase sealed, ya se puede hacer pattern matching. El siguiente ejemplo de pattern matching solo implementa 2 casos posibles de todos los 4 que se implementaron en el ejemplo anterior:



El compilador lanzará el siguiente mensaje de error con los casos que no estén contemplados:



Como alternativa a los errores, se puede crear un valor por defecto que recoja cualquier error, haciendo que el compilador no muestre constantemente mensajes de advertencia, pero obviando el buen funcionamiento del programa:

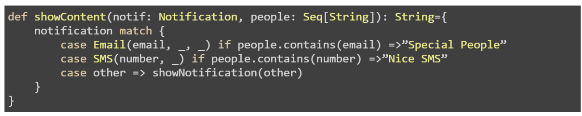


Para evitar esto, existe la posibilidad de utilizar la anotación **@unchecked**.

Esta anotación se utiliza durante el pattern matching y hace que el compilador omita los mensajes de advertencia por no tener todos los casos controlados.

## **Pattern guards**

Las pattern guards son simplemente expresiones booleanas que se usan para hacer los casos más específicos. Simplemente hay que agregar **if <booleanexpression>** después del patrón. En el siguiente código se muestra un ejemplo básico:



El caso other delega los demás resultados al método *showNotification()*

Para poder comprobar el ejemplo anterior, primero hay que crear los valores:



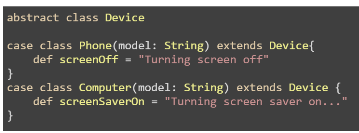
Tras crear los valores, podemos mostrar por pantalla el resultado:



En case Email(email, \_, \_), el patter solo coincide si el email está en la lista que hay en la secuencia People.

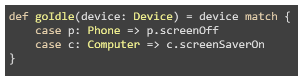
## **Matching de tipos**

En Scala es posible hacer matching en los tipos. En el siguiente ejemplo vamos a definir una clase con dos clases case:



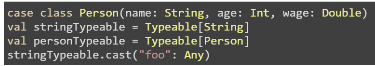
En esta ocasión, se va a definir un método, el cual se va a comportar de una manera concreta dependiendo del tipo de Device.

El siguiente ejemplo muestra que comportamiento se va a otorgar al método según el tipo de dato que reciba como parámetro. Por convenio se escribe la primera letra del tipo como identificador case (“p” y “c” en este caso):

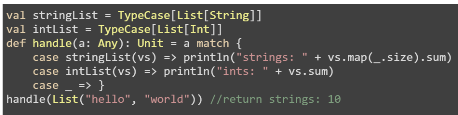


Existe la posibilidad de hacer matching con tipos genéricos. Para este tipo de matching, podemos utilizar los tipos de datos “Typeable” y “TypeCase”.

Typeable es una clase de tipo que proporciona la capacidad de convertir valores de cualquier tipo a un tipo específico.

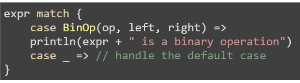


El tipo typeCase es esencialmente un extractor para instancias Typeable. TypeCase y Typeable permiten implementar el ejemplo en la sección anterior de la siguiente manera:



**Wildcard**

El wildcard pattern (\_) hace matching con cualquier objeto.

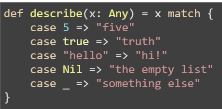


Wildcards también se pueden usar para ignorar partes de un objeto que no nos importa. Por ejemplo, al ejemplo anterior en realidad no le importa cuáles son los elementos de una operación binaria; simplemente verifica si es o no una operación binaria. Por lo tanto, el código también puede usar el pattern wildcard para los elementos de BinOp.

**Patrón de constante**

Un patrón constante solo hace matching consigo mismo. Cualquier literal se puede usar como una constante.

Por ejemplo, 5, true y "hola" son todos patterns constants. Además, cualquier objeto val o singleton se puede usar como una constante. Por ejemplo, Nil, un objeto singleton, es un pattern que solo hace matching con la lista vacía.



**Patrón variable**

Un patrón variable hace matching con cualquier objeto, como un Wildcard. Pero a diferencia de un Wildcard, Scala vincula la variable al objeto que sea. Por ejemplo, el fragmento siguiente muestra un pattern matching que tiene un caso especial para cero y un caso predeterminado para todos los demás valores. El caso predeterminado utiliza un pattern variable para que tenga un nombre para el valor (somethingElse), sin importar de que se trate.



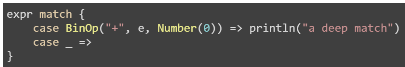
**Patrón constructor**

Los constructores son donde el pattern matching se vuelve realmente potente.

Un pattern constructor es como el ejemplo anterior "BinOp (" + ", e, Number (0))". Consiste en un nombre (BinOp) y luego una serie de patrones entre paréntesis: "+", e, y Number (0).

Suponiendo que el nombre designa una clase case, dicho patrón verifica que el objeto pertenece a la clase case, y luego verifica que los parámetros del constructor del objeto coinciden con los patrones adicionales pasados. Los propios parámetros adicionales pueden tener patrones de constructor. Dependiendo de esto se puede determinar la profundidad del pattern matching.

Por ejemplo, el patrón que se muestra a continuación comprueba que el objeto de nivel superior es un BinOp, que su tercer parámetro de constructor es un número y que el campo de valor de ese número es 0. Este patrón tiene una línea de longitud pero tiene tres niveles de profundidad.



**Patrón secuencia**

Se puede hacer coincidir los tipos de secuencia, como List o Array, del mismo modo que hace coincidir las clases case. Se usa la misma sintaxis, pero ahora se puede especificar cualquier cantidad de elementos dentro del patrón. El siguiente fragmento muestra un patrón que busca una lista de tres elementos que comienza con cero.

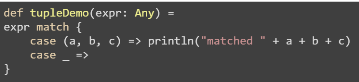


Si queremos hacer matching una secuencia sin especificar la longitud de esta, se puede especificar \_ \* como último elemento del patrón. Este diseño coincide con cualquier cantidad de elementos dentro de una secuencia, incluidos cero elementos.

El siguiente fragmento muestra un ejemplo que hace matching con cualquier lista que comience con cero, independientemente de la longitud de la lista.

**Patrón de tupla**

Se puede hacer matching de tuplas también de manera similar al patrón anterior. En el siguiente fragmento se muestra un patrón (a, b, c) que hace matching con una tupla arbitraria de 3 elementos.



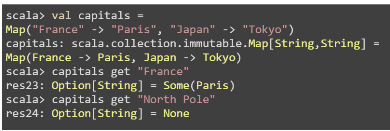
Si se llama al método tupleDemo que se muestra en el fragmento anterior en el intérprete y se le pasan tres elementos se verá lo siguiente:



## **Tipo option**

Scala tiene un tipo estándar llamado Option para valores opcionales. Tal valor puede ser de dos formas: Some (x), donde x es el valor real, o el objeto None, que representa que un valor falta.

Los valores opcionales son producidos por algunas de las operaciones estándar en las colecciones de Scala. Por ejemplo, el método get de Scala's Map produce Some (value) si se ha encontrado un valor correspondiente a una clave determinada, o None si la clave dada no es definido en Map. Como en el ejemplo siguiente:



La forma más común de separar los valores opcionales es a través de un pattern matching.

Por ejemplo:

