Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Data classes & Record

Da switch a match/when & pattern matching

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

LE CLASSI «DATI»

- Nel modello di un sistema vi sono spesso classi che costituiscono semplici agglomerati di dati immodificabili
  - campi dati + costruttore primario + accessor
  - **`equals`**, **`hashCode`** e **`toString`** «standard»
- La loro struttura è «meccanica» e standard, al punto che si fanno generare automaticamente allo strumento
  - vedi generazione automatica in Eclipse
- Ma se sono così standard, ha senso doversi appoggiare allo strumento per fargli generare l'ovvio?
  - *..non potrebbe pensarci il compilatore,* magari a partire da una dichiarazione *molto semplificata..?*

LE CLASSI «DATI»

- In Scala (prima), Kotlin (poi) e ora anche in Java (15+) è stato introdotto a tale scopo un costrutto semplificato
  - IDEA: definire i campi dati e lasciare al compilatore «tutto il resto»
  - Scala: case classes
  - Kotlin: data classes
  - Java 15: record
- L'espressività varia da un linguaggio all'altro
  - in Scala, le case classes lavorano in tandem col costrutto **`match`** (evoluzione dello **`switch`**) che supporta *pattern matching*
  - in Kotlin, le data classes lavorano in tandem col costrutto **`when`** (evoluzione dello **`switch`**) ma fino a un certo punto…
  - in Java 17+, i record beneficiano del *pattern matching* nel costrutto **`switch`** (con qualche limitazione, per ora)

CASE CLASSES IN SCALA

- In Scala, una *case class* – introdotta dalla keyword **`case`** davanti a **`class`** – implica le seguenti specificità:
  - la classe contiene dati immutabili
  - le istanze vengono create *senza usare l'operatore* **`new`** (è definito un factory method implicito)
  - la classe include definizioni implicite e «standard» per **`toString`**, **`equals`**, **`hashCode`** (in particolare, **`equals`** confronta i campi)
  - è fornito un metodo **`copy`** per duplicare un'istanza
  - la classe supporta l'ereditarietà, ma *non su altre case classes* (case-to-case classes inheritance is prohibited)
    - la limitazione si bypassa studiando una diversa architettura software
  - è supportata la de-strutturazione in forma di *pattern matching*
    - si basa sull'uso «behind the scenes» dei metodi accessor

Scala

CASE CLASSES: ESEMPIO

```scala
case class Persona(val nome:String, var anni:Int=0) {}
```

Scala

```scala
def main(args: Array[String]) : Unit = {

   val p1 = Persona("John", 25); println(p1);
   val p2 = Persona("Mary", 22); println(p2);
   val p3 = Persona("John", 25); println(p3);

   print(p1.equals(p3)); print(", "); println(p1 == p3);
   print(p1.hashCode().equals(p3.hashCode()));
       print(", "); println(p1.hashCode() == p3.hashCode());
}
```

Scala

toString generata automaticamente

equals generata automaticamente

hashCode generata automaticamente

```
Persona(John,25)
Persona(Mary,22)
Persona(John,25)
true, true
true, true
```

CASE CLASSES: ESEMPIO

```
case class Persona(val nome:String, var anni:Int=0) {}
```
Scala

```
def main(args: Array[String]) : Unit = {

   val p1 = Persona("John", 25); println(p1);
   val p2 = Persona("Mary", 22); println(p2);
   val p3 = Persona("John", 25); println(p3);

   print(p1.equals(p3)); print(", "); println(p1 == p3);
   print(p1.hashCode().equals(p3.hashCode()));
       print(", "); println(p1.hashCode() == p3.hashCode());

   val Persona(nome, età) = p1
   println(nome); println(età);
}
```
Scala

Pattern matching con *de-strutturazione automatica* dell'oggetto nei suoi componenti! ☺

```
John
25
```

CASE CLASSES: ESEMPIO

Scala

```
case class Persona(val nome:String, var anni:Int=0) {

 def show(): String = {
  this match {
    case Persona(nome, età) if età<18 => s"$nome è minorenne"
    case Persona(nome, età) if età<70 => s"$nome è un lavoratore"
    case other => s"$nome è pensionato"
   }
 }
}
```

In Scala, il costrutto *match* (sostituisce lo switch) è un'espressione, non un'istruzione!
Quindi, può restituire un risultato!

Le *guardie* semantiche consentono di *discriminare anche semanticamente* (non solo sintatticamente) i diversi casi

- Il costrutto **`match`** di Scala è *molto più* di uno **`switch`**
  - è un'espressione, non un'istruzione
  - i casi sono mutuamente esclusivi (niente più *fall through*!)
  - supporta pattern matching sintattico e guardie semantiche

CASE CLASSES: ESEMPIO

Scala

```
def main(args: Array[String]) : Unit = {

   val p1 = Persona("John", 25); println(p1);
   val p2 = Persona("Mary", 22); println(p2);
   println(p1.show());
   println(p2.show());

   println(Persona("Henry",55).show());
   println(Persona("Jane",17).show());
   println(Persona("Sergio",76).show());

}
```

```
John è un lavoratore
Mary è un lavoratore
Henry è un lavoratore
Jane è minorenne
Sergio è pensionato
```

DATA CLASSES IN KOTLIN

- In Kotlin, una *data class* – introdotta dalla keyword **`data`** davanti a **`class`** – implica le seguenti specificità:

Kotlin

  - la classe contiene dati immutabili
  - la classe include definizioni implicite e «standard» per **`toString`**, **`equals`**, **`hashCode`** (in particolare, **`equals`** confronta i campi)
  - la classe include *N* metodi accessor **`componentN`**, uno per campo
  - è fornito un metodo **`copy`** per duplicare un'istanza
  - una data class è **`final`** e quindi non può neppure essere astratta
    - si può però derivare una data-class da una classe standard
  - è supportata la de-strutturazione in forma di *pattern matching*
    - si basa sull'uso «behind the scenes» dei metodi **`componentN`**

DATA CLASSES: ESEMPIO

```kotlin
data class Persona(val nome:String, var anni:Int=0) {}
```

Kotlin

```kotlin
fun main(args: Array<String>) : Unit {

   val p1 = Persona("John", 25); println(p1);
   val p2 = Persona("Mary", 22); println(p2);
   val p3 = Persona("John", 25); println(p3);

   print(p1.equals(p3)); print(", "); println(p1 == p3);
   print(p1.hashCode().equals(p3.hashCode()));
       print(", "); println(p1.hashCode() == p3.hashCode());
}
```

Kotlin

toString generata automaticamente

equals generata automaticamente

hashCode generata automaticamente

```
Persona(nome=John, anni=25)
Persona(nome=Mary, anni=22)
Persona(nome=John, anni=25)
true, true
true, true
```

DATA CLASSES: ESEMPIO

Kotlin

```
data class Persona(val nome:String, var anni:Int=0) {}
```

Kotlin

```
fun main(args: Array<String>) : Unit {

   val p1 = Persona("John", 25); println(p1);
   val p2 = Persona("Mary", 22); println(p2);
   val p3 = Persona("John", 25); println(p3);

   print(p1.equals(p3)); print(", "); println(p1 == p3);
   print(p1.hashCode().equals(p3.hashCode()));
       print(", "); println(p1.hashCode() == p3.hashCode());

   val (nome, età) = p1
   println(nome); println(età);
}
```

Pattern matching con *de-strutturazione automatica* dell'oggetto nei suoi componenti! ☺

```
John
25
```

DATA CLASSES: ESEMPIO

Kotlin

```
data class Persona(val nome:String, var anni:Int=0) {

 fun show(): String {
  return when(this.anni) {
    in  0..17 -> "$nome è minorenne"
    in 18..70 -> "$nome è un lavoratore"
    else -> "$nome è pensionato"
   }
 }
}
```

In Kotlin, il costrutto *when* (sostituisce lo switch) è un'espressione, non un'istruzione! Quindi, può restituire un risultato!

Kotlin supporta solo alcuni *tipi prestabili* di guardie semantiche: **`in range`** è una

- Il costrutto **`when`** di Kotlin è *più* di uno **`switch`** (ma meno del **`match`**)
  - è un'espressione, non un'istruzione
  - i casi sono mutuamente esclusivi (niente più *fall through*!)
  - supporta pattern matching sintattico, ma non molte guardie semantiche

DATA CLASSES: ESEMPIO

Kotlin

```
data class Persona(val nome:String, var anni:Int=0) {

 fun show(): String =
   when(this.anni) {
    in  0..17 -> "$nome è minorenne"
    in 18..70 -> "$nome è un lavoratore"
    else -> "$nome è pensionato"
   }
}
```

In generale, in Kotlin se la funzione è costituita da una *singola expressione* si possono evitare sia il blocco `{}` sia la keyword `return`: basta un `=`

Ulteriori dettagli sul costrutto ***when***

- un'altra guardia possibile è `is tipo`, che verifica che l'oggetto sia del tipo specificato
- usato senza argomenti (`when{..}`), equivale a una catena di `if`

- Il costrutto **`when`** di Kotlin è *più* di uno **`switch`** (ma meno del **`match`**)
  - è un'espressione, non un'istruzione
  - i casi sono mutuamente esclusivi (niente più *fall through*!)
  - supporta pattern matching sintattico, ma non molte guardie semaniche

DATA CLASSES: ESEMPIO

Kotlin

```
fun main(args: Array<String>) : Unit {

   val p1 = Persona("John", 25); println(p1);
   val p2 = Persona("Mary", 22); println(p2);
   println(p1.show());
   println(p2.show());

   println(Persona("Henry",55).show());
   println(Persona("Jane",17).show());
   println(Persona("Sergio",76).show());

}
```

```
John è un lavoratore
Mary è un lavoratore
Henry è un lavoratore
Jane è minorenne
Sergio è pensionato
```

RECORD IN JAVA 15+

- In Java 15+, un *record* – introdotto dalla keyword **`record`** *in sostituzione di* `class` – implica le seguenti specificità:
  - la classe (record) contiene dati immutabili: *«Records are intended to be simple data carriers»*
  - le istanze vengono *create normalmente* con l'operatore **`new`**
  - la classe include definizioni implicite e «standard» per **`toString`**, **`equals`**, **`hashCode`** (in particolare, **`equals`** confronta i campi)
  - la classe include *N* metodi accessor di *nome uguale al campo-dati*
  - è fornito un metodo **`copy`** per duplicare un'istanza
  - un record è **`final`** e non può essere astratto
  - per ora *non è ancora supportata* la de-strutturazione in forma di pattern matching → JEP 420: https://openjdk.java.net/jeps/420 (ma ci stanno lavorando…)

Java

RECORD: ESEMPIO

```
public record Persona(String nome, int anni) {}
```

Java

```
public static void main(String[] args) {
   var p1 = new Persona("John", 25); System.out.println(p1);
   var p2 = new Persona("Mary", 22); System.out.println(p2);
   var p3 = new Persona("John", 25); System.out.println(p3);
   System.out.print(p1.equals(p3));
       System.out.print(", "); System.out.println(p1 == p3);
   System.out.println(p1.hashCode() == p3.hashCode())
   System.out.println(p1.nome());
   System.out.println(p1.anni());
}
```

Java

`toString` generata automaticamente

`equals` generata automaticamente

NB: `equals` non è ==

Accessor generati automaticamente

```
Persona[nome=John, anni=25]
Persona[nome=Mary, anni=22]
Persona[nome=John, anni=25]
true, false
John
25
```

NB: in Java, l'operatore == non è sinonimo di `equals` !

Per ora, Java *non ha* pattern di de-strutturazione

`hashCode` generata auto-maticamente

RECORD: ESEMPIO

Java

```
public record Persona(String nome, int anni) {


 public String show() {
  return
    switch(this.anni) {
      case 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17 ->
            this.nome + " è minorenne";
      case 71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,
           86,87,88,89,90,91,92,93,94,95,96,97,98,99,100 ->
            this.nome + " è un pensionato";
      default -> this.nome + "è un lavoratore";
    };
 }
}
```

Da Java 13, la *switch expression* – che può restituire un risultato – si affianca al tradizionale switch statement (che resta).

Nella *switch expression* l'operatore **->** sostituisce il tradizionale **:**

Grazie al suo *break implicito,* evita il fall-through fra i casi.

Inoltre, tutti i casi devono essere coperti (niente buchi!)

Fino a Java 16, lo *switch* di Java **non** supportava pattern sintattici (es. range di valori) né, a maggior ragione, guardie semantiche

JAVA 17: PATTERN MATCHING EVOLUTO

Java

```
public record Persona(String nome, int anni) {

 public String show() {
  return
    switch(Integer.valueOf(this.anni)) {
      case Integer i && i<18 ->
            this.nome + " è minorenne";
      case Integer i && i>70 ->
            this.nome + " è un pensionato";
      default -> this.nome + "è un lavoratore";
    };
 }
}
```

Le guardie semantiche di tipo *richiedono tipi non primitivi*
→ necessario convertire **`int`** in **`Integer`**

Da Java 17, la *switch expression supporta il pattern matching* (first preview; second preview prevista in Java 18, release in Java 19) *con guardie semantiche*

ATTENZIONE: essendo una preview feature
- compilazione con **`--enable-preview --source 17`**
- esecuzione con **`--enable-preview`**

NB: allo stato **non** è previsto il supporto per pattern di destrutturazione (tuttavia, sono allo studio)

RECORD: ESEMPIO

Java

```
public static void main(String[] args) {
  var p1 = new Persona("John", 25); System.out. println(p1);
  var p2 = new Persona("Mary", 22); System.out. println(p2);
  System.out. println(p1.show());
  System.out. println(p2.show());
  System.out. println(new Persona("Henry",55).show());
  System.out. println(new Persona("Jane",17).show());
  System.out. println(new Persona("Sergio",76).show());
}
```

```
John è un lavoratore
Mary è un lavoratore
Henry è un lavoratore
Jane è minorenne
Sergio è pensionato
```

RECORD IN JAVA 15+
ALCUNI DETTAGLI

Java

- Un *record* può:
  - introdurre nuovi metodi (dopo tutto, è una normale classe)
  - prevedere campi statici e metodi statici (dopo tutto è una classe)
  - implementare interfacce (capiremo a suo tempo..)
  - definire costruttori ausiliari, ma solo col corpo della forma **`this(…)`**
  - ridefinire il costruttore primario, anche in forma shortcut senza argomenti: questo schema si usa spesso per introdurre una sezione iniziale di verifica delle precondizioni
- Un *record* non può:
  - aggiungere ulteriori campi dati di istanza (avrebbero dovuto essere dichiarati nell'intestazione)
  - modificare i campi dati di istanza (sono **`final`**)
  - essere esteso in ereditarietà (è una classe **`final`**)

RECORD & COSTRUTTORI: ESEMPIO

Java

```
public record Persona(String nome, int anni) {
    public Persona {
     // NB: non è un default constructor, non ha lista di argomenti!
     // E' il preludio del costruttore/2 generato automaticamente
     java.util.Objects.requireNonNull(nome, "deve esserci un nome");
     java.util.Objects.requireNonNull(anni, "deve esserci una età");
    }

    public Persona() { // costruttore ausiliario classico
         this("Pippo",333); // no assegnamenti diretti this.xx=xx
    }
  ...
}
```

Questo codice *si aggiunge* a quello del costruttore canonico generato automaticamente

NB: la classe di libreria **Objects** offre molti metodi di verifica di precondizioni, che consentono anche di specificare azioni alternative in caso esse non siano verificate.

ESPLORARE!