Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Enumerativi

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

ENUMERATIVI: L'EVOLUZIONE

- Il linguaggio C offre gli enumerativi (keyword **`enum`**), ma sono poco più che *alias per costanti intere*
  - compatibili con gli interi (e quindi mischiabili orrendamente..)
  - essendo "praticamente degli int", *non possono avere comportamento associato* → necessità di helper functions piazzate "da qualche parte"
- Java, Scala, Kotlin (in misura minore C#) offrono un enumerativo ben più completo, basato sul *typesafe enum pattern*
  - stessa keyword **`enum`**, ma sostanza molto diversa
  - un tipo enumerativo è una [quasi] normale classe, di cui però l'utente *non può costruire istanze a piacere*: il costruttore è privato
  - i valori elencati nell'enumerativo sono le *uniche possibili istanze* della classe, esposte come costanti (**`final`**)
  - in quanto classi, questi enumerativi possono avere *stato* e *metodi*

ENUMERATIVI Java

- Nella declinazione più semplice, per dichiarare un tipo enumerativo basta elencarne i valori:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;
}
```

Java

- Questa dichiarazione causa la costruzione di *quattro istanze pubbliche e statiche* della classe **`Direction`**
- Si possono usare semplicemente con il loro nome assoluto:

```
Direction dir = Direction.NORTH;
```

  – MA dentro uno **`switch`** va usato, come vedremo, *il nome relativo*

ENUMERATIVI C#

- Anche in C# nel caso più semplice per dichiarare un tipo enumerativo basta elencare i valori, ma senza '`;`' finale:

C#

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST
}
```

- Però qui le costanti si usano *sempre* con il *nome assoluto*:

```
Direction dir = Direction.NORTH;
```

  – si usa il nome assoluto anche *dentro* uno **`switch`**

ENUMERATIVI Scala

- Scala aveva in origine enumerativi più scomodi
- MA Scala 3 ha adottato lo stesso approccio di Java

```scala
enum Direction {
 case NORTH, SOUTH, EAST, WEST
}
```

Scala

  - è anche possibile evitare le parentesi, secondo la sintassi «quiet»: in tal caso è cruciale andare a capo con un tab nei modi previsti

```scala
enum Direction : // quiet syntax
   case NORTH, SOUTH, EAST, WEST
```

Scala

- Anche qui le costanti si usano *sempre* con il *nome assoluto*:

```
Direction dir = Direction.NORTH;
```

  - si usa il nome assoluto anche *dentro* il costrutto **`match`** (~**`switch`**)

ENUMERATIVI Kotlin

- In Kotlin la situazione è quasi identica a Java, ma la keyword **`enum`** è un «aggettivo» del «sostantivo» **`class`**:

Kotlin

```
public enum class Direction {
    NORTH, SOUTH, EAST, WEST;
}
```

- Come in C# e Scala, le costanti enumerative si usano sempre con il loro nome assoluto:

```
var dir : Direction = Direction.NORTH;
```

  - come in C#, esso va usato anche *dentro* uno **`switch`**
  - MA in Kotlin lo **`switch`** si chiama **`when`** e ha una sintassi innovativa (vedere oltre)

Java: UN SEMPLICE ESEMPIO

- Esempio di uso dentro uno **`switch`**:

Java

```
Direction dir = Direction.NORTH;
switch (dir) {
  case NORTH:System.out.println("North"); break;
  case SOUTH:System.out.println("South"); break;
  case EAST: System.out.println("East");  break;
  case WEST: System.out.println("West");  break;
}
```

- Però, questo approccio non è scalabile: lo **`switch`** diventa rapidamente illeggibile, non manutenibile, ridondante
- Molto meglio *sfruttare il fatto che l'enum sia una classe* per *personalizzarla* e introdurre *metodi appositi.*

LO STESSO ESEMPIO IN C#

- Esempio di uso dentro uno **`switch`** in C#:

C#

```
Direction dir = Direction.NORTH;
switch (dir) {
  case Direction.NORTH: Console.WriteLine("North"); break;
  case Direction.SOUTH: Console.WriteLine("South"); break;
  case Direction.EAST:  Console.WriteLine("East");  break;
  case Direction.WEST:  Console.WriteLine("West");  break;
}
```

L'ESEMPIO IN Scala…

- In Scala, l'antico (e obsoleto..) costrutto **`switch`** del C viene sostituito da un più moderno e potente costrutto **`match`**

Scala

```
dir match {
  case Direction.NORTH => println("North");
  case Direction.SOUTH => println("South");
  case Direction.EAST  => println("East");
  case Direction.WEST  => println("West");
}
```

- Cambia la sintassi, perché cambia la semantica:
  - i rami diventano mutuamente esclusivi → *non occorre più il* ***`break`*** (la cui dimenticanza ha causato negli anni tanti bug…)
  - la struttura «x **`match`** expression» evidenzia l'idea che x possa fare match con tante situazioni, *anche ben oltre le semplici costanti*

…e in Kotlin

- Anche in Kotlin l'antico e obsoleto costrutto **`switch`** viene sostituito da un costrutto più moderno e potente: **`when`**

Kotlin

```
when(dir) {
  Direction.NORTH -> println("North");
  Direction.SOUTH -> println("South");
  Direction.EAST  -> println("East");
  Direction.WEST  -> println("West");
}
```

  - come in Scala, i rami sono mutuamente esclusivi → niente **`break`**
  - estetica: cambia la freccina (da «cicciotta» **`=>`** a «snella» **`->`**)
  - sparisce le keyword **`case`**, ritenuta inutilmente prolissa
  - la struttura è una via di mezzo fra Java (**`when`** iniziale) e Scala (il costrutto fa match con le più varie situazioni, *ben oltre le costanti)*

Dall'antico `switch` a `match`/`when`

- I costrutti **`match`** (Scala) e **`when`** (Kotlin) possono essere usati anche come espressioni che restituiscono un valore

Scala

```
val name = dir match {
  case Direction.NORTH => "North";
  case Direction.SOUTH => "South";
  case Direction.EAST  => "East";
  case Direction.WEST  => "West";
}
```

Mutuamente esclusive, senza **`break`**

Kotlin

```
val name = when(dir) {
  Direction.NORTH -> "North";
  Direction.SOUTH -> "South";
  Direction.EAST  -> "East";
  Direction.WEST  -> "West";
}
```

Mutuamente esclusive, senza **`break`**

`switch` expression in Java 16+

- Questo approccio ha avuto tale successo che da Java 16 è previsto anche in Java come *switch expression*

Java

```
var name = switch(dir) {
  case Direction.NORTH -> "North";
  case Direction.SOUTH -> "South";
  case Direction.EAST  -> "East";
  case Direction.WEST  -> "West";
}
```

Mutuamente esclusive, senza **`break`**

  - a differenza dello **`switch`** classico (che è un'istruzione), la **`switch`** expression *restituisce un valore*
  - si riconosce per la presenza del separatore **`->`** anziché **`:`**
  - nella switch expression i rami diventano mutuamente esclusivi → non occorre più il **`break`**

Java: ENUM COME CLASSE

- L'**`enum`** Java è in realtà una *classe gestita dal compilatore in modo speciale*, aggiungendo automaticamente:
  - un costruttore privato
  - un metodo di utilità **`ordinal`**
    - restituisce *l'indice di quel valore* nell'elenco degli enumerativi (ossia 0 per `NORTH`, 1 per `SOUTH`, etc.)
  - due metodi statici *factory*, **`values`** e **`valueOf`**
    - **`values()`** restituisce *un array* con *tutti i possibili valori* dell'enumerativo
    - **`valueOf(String)`** restituisce *l'oggetto corrispondente* alla stringa passata (ossia da `"NORTH"` restituisce `Direction.NORTH`, etc.)

Java: ENUM COME CLASSE

- Quindi la classe è compilata circa così:

Java

```
public final class Direction {
  public static final Direction NORTH = new Direction();
  public static final Direction SOUTH = new Direction();
  public static final Direction EAST  = new Direction();
  public static final Direction WEST  = new Direction();

  private Direction() {...} // costruttore privato
  public static Direction[] values() {
        return un array con tutti i possibili valori dell'enumerativo
  }
  public static Direction valueOf(String s) {
        return l'opportuna istanza dell'enumerativo
  }
  public int ordinal() {
        return l'indice corrispondente all'enumerativo corrente (this)
  }
}
```

ENUM IN Java: ESEMPIO D'USO

- Questo semplice frammento mostra come usarli
  - `Direction.values()` fornisce *i valori su cui iterare*
    - poi, per ogni iterazione si effettua l'operazione (es. si stampa l'indice)
  - `Direction.valueOf()` converte una stringa nel corrispondente oggetto dell'enumerativo

```
for (Direction d : Direction.values())
    System.out.println( d.ordinal() );
System.out.println(Direction.valueOf("EAST"));
```

Java

Output

```
0
1
2
3
EAST
```

ENUM IN C#: ESEMPIO D'USO

- Nel caso C#, i metodi disponibili sono diversi e *meno immediati* nell'uso (richiedono **`typeof`** e a volte cast):
  - **`GetName`** recupera il nome dell'i-esima costante dell'enumerativo
  - **`GetNames`** restituisce un array con tutti i *nomi* delle costanti
  - **`GetValues`** restituisce un array con tutti i *valori* delle costanti
  - **`Parse`** recupera la costante dato il suo nome

Necessario perché i metodi valgono per tutti gli enum..

```
Console.WriteLine(
            Direction.GetName(typeof(Direction),1) );

string[] allNames = Direction.GetNames(typeof(Direction));
foreach(string s in allNames) Console.WriteLine(s);
foreach(int v in Direction.GetValues(typeof(Direction)) …
Direction shouldBeSouth =
  (Direction) Direction.Parse(typeof(Direction), "SOUTH");
Console.WriteLine(shouldBeSouth);
```

Cast necessario

C#

L'ESEMPIO IN Scala

- In Scala 3 la situazione è praticamente identica a Java
  - metodi **`values`**, **`valueOf`**, **`ordinal`**
  - NB: **`ordinal`** non accetta argomenti, quindi va invocato senza **`()`**

```
for (d <- Direction.values )
      println( d.ordinal );
println(Direction.valueOf("EAST"));
```

Scala

```
0
1
2
3
EAST
```

  - metodo aggiuntivo **`fromOrdinal`** per ottenere l'i-esimo valore

```
for (d <- Direction.values )
      println( d.ordinal );
println(Direction.fromOrdinal(2));
```

Scala

```
0
1
2
3
EAST
```

L'ESEMPIO IN Kotlin

- Anche in Kotlin la situazione è pressoché identica a Java
  - unica differenza: **`ordinal`** non è più un metodo, ma un valore intero (read-only) associato alla specifica istanza
  - l'effetto netto è che, come in Scala, si usa *senza parentesi*

Kotlin

```
for (d in Direction.values())
        println( d.ordinal );
println(Direction.valueOf("EAST"));
```

```
0
1
2
3
EAST
```

ESTENDERE L'ENUMERATIVO

- Essendo l'enum una [quasi] normale classe, nulla vieta che *aggiungiamo noi altri metodi,* ad esempio per:
  - **`getOpposite`**: restituire la direzione opposta a quella data
  - …

Non C# !

Java

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;
    public Direction getOpposite(){
        switch (this) {
            case NORTH: return SOUTH;
            case SOUTH: return NORTH;
            case EAST:  return WEST;
            case WEST:  return EAST;
        }
    }
}
```

`getOpposite` è un normale metodo, quindi **`this`** è la specifica istanza dell'enumerativo su cui il metodo è chiamato.

ESEMPIO D'USO:

```
System.out.println(
 dir.getOpposite());
```

Questo non si può fare in C#: il costrutto **`enum`** NON è estendibile come una normale classe

ESTENDERE L'ENUMERATIVO

- Essendo l'enum una [quasi] normale classe, nulla vieta che *aggiungiamo noi altri metodi,* ad esempio per:
  - **`getOpposite`**: restituire la direzione opposta a quella data
  - **`toString`**: ottenere una rappresentazione stringa del valore
- Problema: come progettare **`toString`**?
  1. modo becero e non scalabile:
     adottare una struttura con **`switch`** (come in `getOpposite`)
     → *non pensateci neanche!!*
  2. modo intelligente e flessibile:
     inserire in ogni istanza-costante il valore stringa "corrispondente"
     - in altre parole: associare all'istanza NORTH la stringa "nord", all'istanza SOUTH la stringa "sud", e così via
     - bisogna necessariamente farlo in fase di costruzione
       → *personalizzare il costruttore privato*

ENUM COME CLASSE Java PERSONALIZZATA

- Oltre ad aggiungere metodi, possiamo *aggiungere proprietà* e farle *inizializzare normalmente* dal costruttore (privato)
- Ne definiamo due:
  - **`value`**: la stringa "corrispondente"
  - **`degrees`**: i "gradi" corrispondenti (0°=nord, 90°=est, etc.)

Java

```
private String value;
private int degrees;

private Direction(String value, int degrees) {
        this.value = value;
        this.degrees = degrees;
}
```

- La dichiarazione dell'enumerativo va adeguata di conseguenza, passando a ogni costante le due proprietà corrispondenti.

ENUM COME CLASSE Java PERSONALIZZATA

- Versione revisionata:

Java

```
public enum Direction {

   NORTH("Nord", 0), SOUTH("Sud", 180),
   EAST("Est", 90),  WEST("Ovest", 270);

   private String value;
   private int degrees;

   private Direction(String value, int degrees) {
      this.value = value; this.degrees = degrees;
   }
   public String toString(){
      return value + " a " + degrees + "°";   }
   public Direction getOpposite() { … }
}
```

ENUM Java: ESEMPIO D'USO

- Stampiamo una direzione e la sua opposta:

Java

```
class ProvaEnum  {
  public static void main(String[] args) {
        Direction dir = Direction.NORTH;
        System.out.println(dir);
        System.out.println(dir.getOpposite());
  }
}
```

```
Nord a 0°
Sud a 180°
```

ENUM COME CLASSE Scala PERSONALIZZATA

- Lo stesso esempio in Scala

Scala

```
enum Direction(value: String, degrees:Int) {
  case NORTH extends Direction("Nord", 0);
  case SOUTH extends Direction("Sud", 180);
  case EAST  extends Direction("Est", 90);
  case WEST  extends Direction("Ovest", 270);
  override
  def toString() =  value + " a " + degrees + "°";
  def getOpposite() = this match {
      case NORTH => SOUTH;
      case SOUTH => NORTH;
      case EAST =>  WEST;
      case WEST =>  EAST;
   }
}
```

```
Nord a 0°
Sud a 180°
```

ENUM COME CLASSE Kotlin PERSONALIZZATA

- Lo stesso esempio in Kotlin

```
enum class Direction(val value: String, val degrees:Int) {
  NORTH("Nord", 0),
  SOUTH("Sud", 180),
  EAST("Est", 90),
  WEST("Ovest", 270);
  override fun toString() = value + " a " + degrees + "°";
  fun getOpposite() = when(this) {
      NORTH -> SOUTH;
      SOUTH -> NORTH;
      EAST ->  WEST;
      WEST ->  EAST;
   }
}
```

`val` per poterli usare in toString

Kotlin

Analoghi a Java

```
Nord a 0°
Sud a 180°
```

PERSONALIZZARE GLI ENUM IN C#

- In C# la personalizzazione degli enum è più limitata
- È possibile però *associare un valore* alle costanti all'atto stesso della loro dichiarazione, tramite l'operatore =
  - ad esempio, i "gradi" corrispondenti (0°=nord, 90°=est, etc.)

```
public enum Direction {

    NORTH=0, SOUTH=180, EAST=90, WEST=270

}
```

Non in Java

C#

- Il metodo `GetValues` in questo caso restituirebbe 0, 180, 90, 270 anziché i classici 0, 1, 2, 3
- Il metodo `format`, che produce *la stringa corrispondente* alla costante enum, può essere *istruito per restituire il valore associato* anziché il nome della costante stessa

ENUM C#: ESEMPIO D'USO

- Stampiamo la direzione EAST in diversi formati:

C#

```
string value = Direction.Format(
       typeof(Direction), Direction.EAST, fmt);
Console.WriteLine(value);
```

  - il formato `fmt` può essere "G", "F", "X", "D" (anche minuscole)
  - `"G"` produce il nome della costante (es. "EAST") — **EAST**
  - `"F"` produce il nome della costante senza tenere conto di flag
  - `"D"` produce il nome della costante (se non ci sono valori numerici specificati) o il suo valore numerico decimale (se specificato) — **90**
  - `"X"` è identica a `"D"` ma stampa il valore in esadecimale — **0000005A**

UN ALTRO ESEMPIO: DayOfWeek IN JAVA

- **`DayOfWeek`** è un enumerativo del package **`java.time`** che rappresenta i sette giorni della settimana.
- Stampiamo tutti i valori e i rispettivi indici (0-based):

```java
for (DayOfWeek day : DayOfWeek.values())
    System.out.println(day.ordinal() + ": " + day);
```

Java

```
0: MONDAY
1: TUESDAY
2: WEDNESDAY
3: THURSDAY
4: FRIDAY
5: SATURDAY
6: SUNDAY
```

- Ovviamente, un indice 0-based è poco pratico
  - ha senso come rappresentazione interna, non come vista esterna!

UN ALTRO ESEMPIO: DayOfWeek in Java

- Per questo, **`DayOfWeek`** *integra i metodi standard degli enumerativi aggiungendone altri specifici e personalizzati*
- In particolare, **`getValue`** restituisce l'indice 1-based destinato alla vista esterna:

```
for (DayOfWeek day : DayOfWeek.values())

    System.out.println(day.getValue() + ": " + day);
```

Java

- È fornito anche il corrispondente metodo factory di conversione "da indice a oggetto", chiamato **`of`**:

```
DayOfWeek day = DayOfWeek.of(2); // martedì
```

NB: **`DayOfWeek`** e tutta la libreria **`java.time`** sono usati anche da Scala e Kotlin senza modifiche

Kotlin Scala

… DayOfWeek in C#

- Ovviamente, anche C# offre la sua versione di **`DayOfWeek`**
  - ma qui i giorni sono numerati a partire dalla domenica!!
  - *ottimo motivo per usare SEMPRE le COSTANTI, mai i numeri!*

Java

```
0: MONDAY
1: TUESDAY
2: WEDNESDAY
3: THURSDAY
4: FRIDAY
5: SATURDAY
6: SUNDAY
```

C#

```
0: Sunday
1: Monday
2: Tuesday
3: Wednesday
4: Thursday
5: Friday
6: Saturday
```

```
foreach(int v in DayOfWeek.GetValues(typeof(DayOfWeek)))
 Console.WriteLine(v + ": " +
    DayOfWeek.GetName(typeof(DayOfWeek), v));
```

C#

Un caso di studio
Banconote e Portafogli

UN ALTRO ESEMPIO: Banconote

- Vogliamo definire i *tagli* possibili di *banconote* e *monete*
- La prima idea potrebbe essere banalmente questa:

Java

```
public enum Taglio  {
   CINQUECENTO, DUECENTO, CENTO, CINQUANTA,
   VENTI, DIECI, CINQUE, DUE, UNO;
}
```

- Peccato che così facendo diventi un incubo *ricavare il valore dal nome*, quando serve per fare calcoli
  - ad esempio, calcolare il valore di un *portafoglio* (insieme di tagli)
  - uno **`switch`**? non pensateci neppure!!
- La soluzione, di nuovo, passa dal *personalizzare* l'enumerativo *associando anche il valore numerico* al nome del taglio.

L'ESEMPIO Banconote in Java

- Versione riveduta e ampliata:

Java

```
public enum Taglio  {
   CINQUECENTO(500), DUECENTO(200), CENTO(100),
   CINQUANTA(50), VENTI(20), DIECI(10), CINQUE(5),
   DUE(2), UNO(1);
   private Taglio(int valore){
        this.valore = valore;
   }
   private int valore;
   public int getValore() { return valore; }
}
```

C#: facile associare le costanti in fase di inizializzazione

  - così, da ogni costante è immediato ricavare il valore tramite l'apposito metodo **`getValore`** : altro che **`switch`**!

CASO DI STUDIO (1/14)
Calcolo del valore di un portafoglio

- Se l'enumerativo **`Taglio`** è definito in modo basico:

```java
public enum Taglio  {
   CINQUECENTO, DUECENTO, CENTO, CINQUANTA,
   VENTI, DIECI, CINQUE, DUE, UNO;
}
```

Java

- Un portafoglio assume la forma, altrettanto basica, di *array di (opportune quantità di) tali valori enumerativi:*

```java
Taglio[] portafoglio = new Taglio[] {
   Taglio.CINQUANTA, Taglio.VENTI, Taglio.DIECI,
   Taglio.DUE, Taglio.DUE, Taglio.UNO
};
```

Java

Osserva: il portafoglio non esiste in quanto tale (ossia, come classe), ma solo "a run time", come *mera variabile "con un nome evocativo"*

CASO DI STUDIO (2/14)
Calcolo del valore di un portafoglio

- Per calcolare il valore del portafoglio occorre quindi *scorrere l'array*, accumulando in una variabile i valori parziali:

```
static int valore(Taglio[] portafoglio){
   int sum = 0;
   for (Taglio t : portafoglio ) sum += valore(t);
   return sum;
}
```

Java

Funzione ausiliaria

RIFLETTI: sono funzioni statiche *perché non esiste davvero* una entità portafoglio

- e analogamente per stamparne il contenuto:

```
static String contenuto(Taglio[] portafoglio){
   StringBuilder sb = new StringBuilder();
   for (Taglio t : portafoglio) sb.append(t + ", ");
   return sb.toString();
}
```

Java

CASO DI STUDIO (3/14)
Calcolo del valore di un portafoglio

- MA per estrarre il valore del generico taglio, è inevitabile uno **`switch`** che *discrimini uno per uno tutti i possibili valori:*

Java

```
static int valore(Taglio t){
   switch(t){
        case CINQUECENTO:    return 500;
        case DUECENTO:       return 200;
        case CENTO:          return 100;
        case CINQUANTA:      return 50;
        case VENTI:          return 20;
        case DIECI:          return 10;
        case CINQUE:         return 5;
        case DUE:            return 2;
        case UNO:            return 1;
        default:             return 0; // unreachable
   }
}
```

AARGH! È codice cablato, verboso e *fragile!*

Se l'enumerativo cambia (si aggiunge/toglie un valore), questo *non funziona più correttamente*

MA nessuno ti avvisa!

CASO DI STUDIO (4/14)
Calcolo del valore di un portafoglio

- Un piccolo main di prova

Java

```
public static void main(String[] args)   {
 Taglio[] portafoglio = new Taglio[] {
        Taglio.CINQUANTA,
        Taglio.VENTI,
        Taglio.DIECI,
        Taglio.DUE, Taglio.DUE,
        Taglio.UNO
 };
 System.out.println("Contenuto portafoglio: " +
                                contenuto(portafoglio));
 System.out.println("Valore portafoglio: " +
                                valore(portafoglio));
}
```

```
Contenuto portafoglio: CINQUANTA, VENTI, DIECI, DUE, DUE, UNO,
Valore portafoglio: 85
```

CASO DI STUDIO (5/14)
Calcolo del valore di un portafoglio

- È proprio sbagliato il modello di base
  - non è una terza parte esterna (il "cliente" nello **`switch`**) a dover sapere quanto vale un taglio: *è il taglio che deve saperlo!*
  - *il valore è una proprietà intrinseca della banconota o della moneta*
- Refactoring

```
public enum Taglio  {
   CINQUECENTO(500), DUECENTO
   CINQUANTA(50), VENTI(20),
   DUE(2), UNO(1);
   private Taglio(int valore){ this.valore = valore; }
   private int valore;
   public int getValore() { return valore; }
}
```

Codice robusto

Se l'enumerativo cambia (si aggiunge o si toglie un valore), *tutto continua a funzionare senza alcun intervento* ☺

Java

Finalmente, ora è il **`Taglio`** a sapere "i fatti del **`Taglio`**"!

CASO DI STUDIO (6/14)
Calcolo del valore di un portafoglio

- Il calcolo del valore del portafoglio riflette il cambiamento di prospettiva: *chiediamo al taglio quanto vale*

```
static int valore(Taglio[] portafoglio){
   int sum = 0;
   for (Taglio t : portafoglio ) sum += t.getValore();
   return sum;
}
```

Java

Metodo di **Taglio** ☺

- Però, rimangono necessarie *nel cliente* le due funzioni statiche che operano sull'array

```
static int valore(Taglio[] portafoglio)
static String contenuto(Taglio[] portafoglio)
```

Java

  - motivo: il portafoglio è "solo" un array, *non esiste in quanto tale*
  - insoddisfacente, inelegante, inopportuno

CASO DI STUDIO (7/14)
Calcolo del valore di un portafoglio

- Un passo in più potrebbe essere almeno *togliere quelle funzioni statiche dal cliente,* mettendole *altrove*
  - già, ma dove?
  - non può essere "un posto a caso!"
  - in realtà, il loro posto sarebbe "il portafoglio", ma (ancora) non c'è…
  - .. magari una libreria `TaglioLib` (come `FrazLib`) ?
- Una tipica soluzione di compromesso (insoddisfacente) si riassume nella frase *"se manca il plurale, usiamo il singolare"*
  - ovvero: le due funzioni che lavorano su un insieme (array) di **`Taglio`** si collocano, *per vicinanza concettuale*, nella classe **`Taglio`**
  - meglio di niente, ma… chiaramente, ancora insoddisfacente!
  - Infatti non è splendido: anzi…

CASO DI STUDIO (8/14)
Calcolo del valore di un portafoglio

Java

```
public enum Taglio  {
    CINQUECENTO(500), DUECENTO(200), CENTO(100),
    CINQUANTA(50), VENTI(20), DIECI(10), CINQUE(5),
    DUE(2), UNO(1);
    private Taglio(int valore){ this.valore = val
    private int valore;
    public int getValore() { return valore; }
    public static String contenuto(Taglio[] portafoglio){
         StringBuilder sb = new StringBuilder();
         for (Taglio t : portafoglio) sb.append(t + ", ");
         return sb.toString();
    }
    public static int valore(Taglio[] portafoglio){
         int sum = 0;
         for (Taglio t : portafoglio ) sum += t.getValore();
         return sum;
    }
}
```

Messe qui per *vicinanza concettuale* all'idea di **`Taglio`**

MA non è casa loro!
Inquinano la pulizia della classe **`Taglio`**

CASO DI STUDIO (9/14)
Calcolo del valore di un portafoglio

- Per fare le cose bene bisogna riconoscere che *c'è un concetto che sta lottando per venir fuori:* il *Portafoglio*
  - analogia: per esprimere insiemi di frazioni non bastava un array da manipolare "a mano", serviva una vera `FractionCollection`
- **`Portafoglio`** come *classe autonoma* dotata di metodi
  - costruttore a partire da un insieme (array) di **`Taglio`** già fissato
  - costruttore vuoto (+ metodo **`add`** per aggiungere banconote via via)
  - metodi **`valore`** e **`contenuto`** (magari rinominato **`toString`** !)
  - finalmente tutto va a posto: *segno inequivocabile* che la soluzione è robusta, coerente e risolve *veramente* il problema *come va fatto*
- **`Taglio`** torna a essere quella di prima
  - senza funzioni statiche accessorie "messe lì perché non si sapeva dove altro metterle"

CASO DI STUDIO (10/14)
Calcolo del valore di un portafoglio

Java

```
public class Portafoglio {
   private Taglio[] contenuto;
   int logicalSize;
   public Portafoglio(int n){
     contenuto = new Taglio[n];
     int logicalSize = 0;
   }
   public void add(Taglio t){
     contenuto[logicalSize++] = t;
   }
   public Portafoglio(Taglio[] contenuto){
     this.contenuto = Arrays.copyOf(contenuto, contenuto.length);
     logicalSize = contenuto.length;
   }

   ... segue ...
```

`Portafoglio` incapsula l'array di **`Taglio`**

Così finalmente il portafoglio esiste in quanto tale, non è solo "una variabile con un nome evocativo" definita "da qualche parte".

Ha opportuni costruttori e metodi: *espone l'interfaccia esterna ritenuta più idonea,* non quella imposta dal fatto di essere un array!

CASO DI STUDIO (11/14)
Calcolo del valore di un portafoglio

Java

```
public class Portafoglio {

   ... continua ...

   public int getValore(){
     int sum = 0;
     for (int i=0; i<logicalSize; i++) sum += contenuto[i].getValore();
     return sum;
   }

   public String toString(){
     StringBuilder sb = new StringBuilder();
     for (int i=0; i<logicalSize; i++) sb.append(contenuto[i] + ", ");
     return sb.toString();
   }

}
```

Ex metodi statici

Ora finalmente hanno trovato casa!

Però la stringa così costruita non è splendida, mette una virgola anche in fondo... *bisognerebbe fare di meglio*

```
Contenuto portafoglio: CINQUANTA, VENTI, DIECI, DUE, DUE, UNO,
Valore portafoglio: 85
```

CASO DI STUDIO (12/14)
Calcolo del valore di un portafoglio

Java

```
public class Portafoglio {

   ... continua ...

   public int getValore(){
     int sum = 0;
     for (int i=0; i<logicalSize; i++) sum += contenuto[i].getValore();
     return sum;
   }

   public String toString(){
     StringJoiner sj = new StringJoiner(", ");
     for (int i=0; i<logicalSize; i++) sj.add(portafoglio[i].toString());
     return sj.toString();
   }

}
```

Invece di **StringBuilder**, **StringJoiner!**

Si specifica il separatore:
il resto lo fa il joiner.
Basta aggiungere le stringhe

```
Contenuto portafoglio: CINQUANTA, VENTI, DIECI, DUE, DUE, UNO
Valore portafoglio: 85
```

CASO DI STUDIO (13/14)
Calcolo del valore di un portafoglio

Java

```
public static void main(String[] args) {

  Portafoglio pf1 = new Portafoglio(10);
  for(Taglio t: new Taglio[] {
     Taglio.CINQUANTA, Taglio.VENTI, Taglio.DIECI, Taglio.DUE,
     Taglio.DUE, Taglio.UNO }) pf1.add(t);

  Portafoglio pf2 = new Portafoglio(
     new Taglio[] {
        Taglio.CINQUANTA, Taglio.CINQUANTA, Taglio.VENTI,
        Taglio.DIECI, Taglio.DUE, Taglio.DUE, Taglio.DUE, Taglio.UNO
     });
     System.out.println("Contenuto portafoglio: " + pf);
     System.out.println("Valore portafoglio: "    + pf1.getValore());
     System.out.println("Contenuto portafoglio: " + pf2);
     System.out.println("Valore portafoglio: "    + pf2.getValore());
}
```

Costruisce un portafoglio inizialmente vuoto (poi lo riempie)

Costruisce un portafoglio già inizializzato con quel contenuto

```
Contenuto portafoglio: CINQUANTA, VENTI, DIECI, DUE, DUE, UNO
Valore portafoglio: 85
Contenuto portafoglio: CINQUANTA, CINQUANTA, VENTI, DIECI, DUE, DUE, DUE, UNO
Valore portafoglio: 137
```

CASO DI STUDIO (14/14)
Riassumendo..

- L'enumerativo basico *non manteneva in sé tutta la conoscenza* che lo riguardava
  - costringeva altri (i clienti) e fare i salti mortali per scrivere algoritmica
  - non si sapeva *dove mettere* tali algoritmi, perché "non avevano casa"
- L'enumerativo evoluto *incapsula tale conoscenza*
  - evita alla radice gli **`switch`** che cercavano di ricostruire ex post ciò che mancava ex ante
  - codice robusto ed estendibile al posto di codice fragile ad hoc
  - ma per esprimersi al meglio va completato col concetto di *portafoglio*
- Il portafoglio incapsula la nozione "collezione di banconote"
  - l'idea di portafoglio *assurge a vera classe,* non è più solo un bel nome evocativo per un array piazzato da qualche parte
  - niente più funzioni statiche accessorie, tutto va al suo posto