



Alma Mater Studiorum-Università di Bologna  
Scuola di Ingegneria

---

# Enumerativi & ereditarietà Enumerativi & interfacce

*Corso di Laurea in Ingegneria Informatica*  
Anno accademico 2021/2022

**Prof. ENRICO DENTI**

*Dipartimento di Informatica – Scienza e Ingegneria (DISI)*

# ENUM COME CLASSE JAVA (1)

- Già sappiamo che gli enumerativi in Java, definiti con la keyword **enum**, sono sostanzialmente normali classi con:
  - costruttore privato → istanze tutte già prestabilite e statiche
  - vari metodi di utilità automatici (**ordinal**, **values**, **valueOf**)
- Come tali, possono comunque includere altri metodi, ad es:

```
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 + "°";    }  
}
```

# ENUM COME CLASSE JAVA (2)

- Sappiamo anche che tale classe è compilata internamente «circa» così:

```
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 COME CLASSE JAVA (3)

- In realtà, ora possiamo essere più precisi: **tutti gli enumerativi derivano dalla classe base *Enum***

```
Package java.lang  
Class Enum<E extends Enum<E>>  
  
java.lang.Object  
    java.lang.Enum<E>  
  
Type Parameters:  
E - The enum type subclass  
  
All Implemented Interfaces:  
Serializable, Comparable<E>, Constable  
  
This is the common base class of all Java language enumeration types.
```

- ogni volta che si definisce un nuovo enumerativo (es. **Direction**), il compilatore deriva una nuova sottoclasse da **Enum**
- **tale nuova classe è *final***, quindi **non può essere a sua volta estesa** derivandone un'altra (motivi: coerenza interna, integrità, efficienza)

# ENUM COME CLASSE JAVA (4)

- Quindi, in realtà, le cose stanno «effettivamente» così:

```
public final class Direction extends Enum<Direction> {  
    public static final Direction NORTH = Direction.NORTH;  
    public static final Direction SOUTH = Direction.SOUTH;  
    public static final Direction EAST = Direction.EAST;  
    public static final Direction WEST = Direction.WEST;  
  
    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)  
    }  
}
```

La classe-enumerativo **Direction** deriva da **Enum**, contestualmente specializzata allo specifico caso del tipo **Direction**

# CONTROPROVA

- E infatti, se tentassimo di estendere `Direction` → errore!

```
public enum Direction2 extends Direction { // NO!  
    @Override  
    public String toString() {  
        return super.toString().toUpperCase();  
    }  
}
```

```
Direction2.java:1: error: '{' expected  
public enum Direction2 extends Direction {  
        ^  
...  
}
```

- Tuttavia, questo non significa che gli enumerativi siano «intoccabili» e «inestensibili» in assoluto, anzi!
- Estenderli si può, *ma non tramite ereditarietà classica.*

# ESTENDERE UN ENUMERATIVO

- Un enumerativo può essere esteso *implementando interfacce*
  - ciò consente di inserirlo in tassonomie «trasversali» senza interferire con la gerarchia di ereditarietà
  - massima flessibilità, ma senza «accrocchi»
- **Esempio 1: Color**
  - iniziamo con una definizione standard
  - poi, introduciamo un'interfaccia
  - indi, adeguiamo (refactoring) l'enumerativo in modo che la usi
  - tocchiamo con mano i gradi di libertà che questo può dare
- **Esempio 2: operandi e operatori**
  - un esempio più articolato e omnicomprensivo



# ESTENDERE UN ENUMERATIVO

## ESEMPIO 1: Color

- Iniziamo con una definizione classica:

```
public enum Color {  
    WHITE, CYAN, PEACH, ORANGE, APPLE;  
    public Color sum(Color other){  
        return Color.values()[this.ordinal() + other.ordinal()];  
    }  
}
```

- Un semplice possibile main:

```
public static void main(String[] args){  
    System.out.println(WHITE.sum(CYAN));  
    System.out.println(WHITE.sum(PEACH));  
    System.out.println(CYAN.sum(PEACH));  
}
```

*Output*  
CYAN  
PEACH  
ORANGE

0+1 = 1  
0+2 = 2  
1+2 = 3





# ESTENDERE UN ENUMERATIVO

## ESEMPIO 1: Colore

- L'enumerativo espone il metodo pubblico **sum**,  
MA tale proprietà non è reificata a livello di tassonomia
  - il fatto che l'enumerativo abbia **sum** «passa sotto silenzio»: c'è, e chi guarda l'enumerativo lo vede, MA..
  - .. se ci si limita a questo, non si «accomuna» **Color** ad altre classi o enumerativi che abbiano anch'essi la medesima proprietà
- **REFACTORING: rendere esplicita la presenza di **sum****
  1. introdurre un'interfaccia che catturi l'idea di **entità «sommabili»**  
→ **Summable**
  2. far sì che **Color** la implementi, reificando il suo essere «sommabile»  
→ occorre *generalizzare la signature* del metodo **sum** in modo che *non dipenda più da **Color*** (ma soltanto da **Summable**)



# ESTENDERE UN ENUMERATIVO

## ESEMPIO 1: Color

- Refactoring:

```
public enum Color implements Summable {  
    WHITE, CYAN, PEACH, ORANGE, APPLE;  
    @Override  
    public Summable sum(Summable other) {  
        return Color.values()[this.getValue() + other.getValue()];  
    }  
    @Override  
    public int getValue() {  
        return this.ordinal();  
    }  
}
```

Ora sum dipende solo da Summable

```
interface Summable {  
    public Summable sum(Summable other);  
    public int getValue();  
}
```

NB: il main rimane immutato

# ESTENDERE UN ENUMERATIVO

## ESEMPIO 1: Colore

- Se abbiamo fatto un buon lavoro, dovrebbe essere facile catturare altre entità «sommabili»
  - non necessariamente enumerativi, anzi: qualunque cosa!
- Per verificarlo, inventiamoci **StrangeNumber**
  - un «curioso» numero che incapsula un reale, ma il cui `getValue` restituisce sempre un valore intero (arrotondato al più vicino)
  - anch'esso però sarà «sommabile» → **implements Summable**
- Un semplice possibile main:

```
public static void main(String[] args)
{
    System.out.println(
        StrangeNumber.of(12.4F) .sum(StrangeNumber.of(-1.4)) );
    System.out.println(
        StrangeNumber.of(12.5F) .sum(StrangeNumber.of(-1.5)) );
}
```

☺ Sommiamo cose totalmente diverse da prima, ma non importa!  
sum dipende solo da **Summable** !



# ESTENDERE UN ENUMERATIVO

## ESEMPIO 1: Color

```
public class StrangeNumber implements Summable {  
    private float value;  
    private StrangeNumber(float value) { this.value=value;}  
    public static StrangeNumber of(float value) {  
        return new StrangeNumber(value); }  
    public static StrangeNumber of(double value) {  
        return new StrangeNumber((float)value); }  
  
    @Override  
    public Summable sum(Summable other){  
        return StrangeNumber.of(this.getValue() + other.getValue());  
    }  
  
    @Override  
    public int getValue(){  
        return Math.round(value);  
    }  
  
    @Override  
    public String toString(){  
        return String.valueOf(value);  
    }  
}
```

*Con Color*

CYAN

PEACH

ORANGE

*Con StrangeNumber*

11.0

12.0



# ESTENDERE UN ENUMERATIVO

- Un enumerativo può essere esteso *implementando interfacce*
  - ciò consente di inserirlo in tassonomie «trasversali» senza interferire con la gerarchia di ereditarietà
  - massima flessibilità, ma senza «accrocchi»
- Esempio 1: **Color**
  - iniziamo con una definizione standard
  - poi, introduciamo un'interfaccia
  - indi, adeguiamo (refactoring) l'enumerativo in modo che la usi
  - tocchiamo con mano i gradi di libertà che questo può dare
- Esempio 2: operandi e operatori
  - un esempio più articolato (e un po'... «strano»)



# ESEMPIO 2: Operandi & Operatori

- Si vuole definire il concetto di Operando e Operatore

```
public interface Operand {  
    int getValueAsInteger();  
    double getValueAsReal();  
    Operand sum(Operand arg);  
    Operand mul(Operand arg);  
    Operand sub(Operand arg);  
    Operand div(Operand arg);  
}
```

Un **operando** è un'entità caratterizzata da un valore (recuperabile sia come intero, sia come reale) e da operazioni.

```
public enum Operator {  
    PLUS,  
    MINUS,  
    TIMES,  
    DIVBY;  
}
```

Un **operatore** è un **enumerativo** a cui ora dovremo aggiungere le **operazioni**

Come aggiungere le operazioni?



# ESEMPIO 2: Operandi & Operatori

- L'operatore deve definire le operazioni
  - **IDEA:** dichiarare nell'enumerativo un metodo astratto..
  - ..e implementarlo *per ogni costante enumerativa* tramite un *blocco* (tecnicamente, una classe anonima interna)

```
public enum Operator {  
    PLUS { @Override public Operand doOp(Operand arg1, Operand arg2) {  
        return arg1.sum(arg2); } },  
    MINUS { @Override public Operand doOp(Operand arg1, Operand arg2) {  
        return arg1.sub(arg2); } },  
    TIMES { @Override public Operand doOp(Operand arg1, Operand arg2) {  
        return arg1.mul(arg2); } },  
    DIVBY { @Override public Operand doOp(Operand arg1, Operand arg2) {  
        return arg1.div(arg2); } };  
    public abstract Operand doOp(Operand arg1, Operand arg2);  
}
```

A ogni costante è associata la **ridefinizione** del metodo astratto (definisce una classe anonima)

NB: il metodo astratto va dichiarato *dopo* le costanti enumerative



# ESEMPIO 2: Operandi & Operatori

- I due tipi di Operandi: Real...

```
class Real implements Operand {  
    private double v;  
    public Real(double v) { this.v=v; }  
    public String toString() { return String.valueOf(v); }  
    @Override public int getValueAsInteger() { return (int) Math.round(v); }  
    @Override public double getValueAsReal() { return v; }  
    @Override public Operand sum(Operand arg) {  
        return new Real(this.v + arg.getValueAsReal());}  
    @Override public Operand mul(Operand arg) {  
        return new Real(this.v + arg.getValueAsReal());}  
    @Override public Operand sub(Operand arg) {  
        return new Real(this.v + arg.getValueAsReal());}  
    @Override public Operand div(Operand arg) {  
        return new Real(this.v + arg.getValueAsReal());}  
}
```





# ESEMPIO 2: Operandi & Operatori

- I due tipi di Operandi: ...e Colore

```
enum Colore implements Operand {  
    BLU      { @Override double costo() {return 0.2; } },  
    GIALLO   { @Override double costo() {return 1.1; } },  
    VERDE    { @Override double costo() {return 0.8; } },  
    ROSSO    { @Override double costo() {return 1.4; } },  
    ARANCIO  { @Override double costo() {return 1.6; } };  
  
    abstract double costo();  
  
    @Override public int getValueAsInteger() { return ordinal(); }  
    @Override public double getValueAsReal() { return costo(); }  
    @Override public Operand sum(Operand arg) { return  
        Colore.values()[this.getValueAsInteger() + arg.getValueAsInteger()]; }  
    @Override public Operand mul(Operand arg) { return  
        Colore.values()[this.getValueAsInteger() * arg.getValueAsInteger()]; }  
    @Override public Operand sub(Operand arg) { return  
        Colore.values()[this.getValueAsInteger() - arg.getValueAsInteger()]; }  
    @Override public Operand div(Operand arg) { return  
        Colore.values()[this.getValueAsInteger() / arg.getValueAsInteger()]; }  
}
```



# ESEMPIO 2: Operandi & Operatori

- Un possibile main:

```
public static void main(String[] args){  
    Operand op1 = new Real(12.34);  
    Operand op2 = new Real(-2.34);  
    System.out.println(Operator.PLUS.doOp(op1, op2));  
    Operand op3 = Colore.VERDE;  
    Operand op4 = Colore.GIALLO;  
    System.out.println(Operator.PLUS.doOp(op3, op4));  
    Operand op5 = Colore.ROSSO;  
    System.out.println(Operator.PLUS.doOp(op5, op4));  
    System.out.println(Operator.TIMES.doOp(op3, op4));  
}
```

Un tipo particolare  
di Operando

Un altro tipo di  
Operando

L'Operatore lavora su *qualsunque tipo  
di Operando*, presente e futuro! ☺

*Output*  
10.0  
ROSSO  
ARANCIO  
VERDE