UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Laboratorio di Programmazione ad Oggetti

Ph.D. Juri Di Rocco
juri.dirocco@univaq.it
http://jdirocco.github.io

Sommario

- Tipi Enumerativi
- Generic
  - Introduzione
  - Generic e collections
  - Type inference
  - Ereditarietà e tipi generici
  - Type erasure
  - Wildcard
  - Parametri bounded e wildcard bounded
  - Metodi e costruttori generici
  - Parametri covarianti

Tipi Enumerativi (1)

- E' un tipo speciale del tipo **classe**
- Compilatore converte il tipo enumerativo in una classe che estende da `java.lang.Enum`
- Permette di definire una variabile il cui valore viene preso da un insieme predefinito di costanti
  - Convenzione definiti in lettera maiuscola
- In Java sono molto più potenti che in altri linguaggi
- Il body di enum può contenere metodi e altri campi

```
public enum MiaEnumerazione {

      UNO,DUE,TRE;

}
```

Tipi Enumerativi (2)

› Poiché enum viene convertito in classe che eredita da `java.lang.Enum` possono essere utilizzati e/o ridefiniti alcuni metodi della superclasse

```
System.out.println(MiaEnumerazione.UNO.name());
System.out.println(MiaEnumerazione.UNO); //Stesso risultato:
                                          invocato toString();
System.out.println(MiaEnumerazione.valueOf("UNO"));
System.out.println(MiaEnumerazione.valueOf("UNOA")); //Eccezione:
                                                  IllegalArgumentException
System.out.println(MiaEnumerazione.UNO.ordinal()); //OUTPUT: 0
```

`valueOf` è un metodo statico di `java.lang.Enum`

Tipi Enumerativi (3)

```
for (MiaEnumerazione element: MiaEnumerazione.values()) {

      System.out.println("MiaEnumerazione(" + element.ordinal() + ")="
                         + element.name());

}
```

Tipi Enumerativi (4)

› Non è possibile creare una classe che estende da `Enum`

› La classe di tipo `Enum` non può estendere da altre classi

› E' possibile implementare una interfaccia

Tipi Enumerativi (5)

› Esempio

```
public interface Numeratore {
      void stampaIndice();
}
public enum MiaEnumerazione2 implements Numeratore {
    UNO, DUE, TRE;
    @Override
    public void stampaIndice() {
            System.out.println("Indice: " + this.ordinal());
    }
}

Numeratore n = MiaEnumerazione2.DUE;
n.stampaIndice();
```

Tipi Enumerativi (6)

› E' possibile dichiarare variabili metodi e costruttori

› Costruttore deve avere un accesso `private` o di `package`

Tipi Enumerativi (7)

› Esempio

```
public enum MiaEnumerazione3 {
    ZERO(), UNO(1), DUE(2), TRE(3);
    private int valore;
    private MiaEnumerazione3() {
    }
    MiaEnumerazione3(int valore) {
      setValore(valore);
    }
    public void setValore(int valore){this.valore = valore;}
    public int getValore(){return this.valore;}

    @Override
    public String toString() {
      return "" + valore;
    }
}
```

Tipi Enumerativi (8)

› From slides 03

Flusso di controllo: switch (1)

› Costrutto if/else può risultare scomodo quando si ha a che fare con selezioni multiple che prevedono diverse alternative

› Espressione

Tipi primitivi: `byte, short, char, int`

Tipi Enumerativi: `String`

Tipi classi speciali: `Character, Byte, Short, Integer`

Tipi Enumerativi (9)

› Esempio: uso switch

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
public class EnumTest {
    private Day day;
    public EnumTest(Day day) {
        this.day = day;
    }
    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;
……
```

Tipi Enumerativi (10)

```
.....

                case FRIDAY:
                  System.out.println("Fridays are better.");
                  break;
                case SATURDAY:
                case SUNDAY:
                  System.out.println("Weekends are best.");
                  break;
                default:
                  System.out.println("Midweek days are so-so.");
                  break;
        }
    }
.....
```

Tipi Enumerativi (11)

```
.....
        public static void main(String[] args) {
            EnumTest firstDay = new EnumTest(Day.MONDAY);
            firstDay.tellItLikeItIs();
            EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
            thirdDay.tellItLikeItIs();
            EnumTest fifthDay = new EnumTest(Day.FRIDAY);
            fifthDay.tellItLikeItIs();
            EnumTest sixthDay = new EnumTest(Day.SATURDAY);
            sixthDay.tellItLikeItIs();
            EnumTest seventhDay = new EnumTest(Day.SUNDAY);
            seventhDay.tellItLikeItIs();
    }
}
```

Generic (1)

- Caso particolare di polimorfismo dove vengono *parametrizzati* dei tipi di una classe od interfaccia
- In tal caso il codice diventa più robusto poiché alcuni controlli vengono **spostati** da tempo di esecuzione a tempo di compilazione
- Lo scopo è di garantire al programmatore la massima espressività possibile nella scrittura di classi e/o metodi allentando i vincoli sui tipi con cui funzionano questi oggetti
- Sono stati introdotti in Java 5 e si ispirano ai *template* del C++ ma sono molto diversi

Generic (2)

› Esempio

```
class Automobile {}

public class Contenitore {
  private Automobile a;
  public Contenitore(Automobile a) {
      this.a = a;
  }
  public Automobile get() {
      return a;
  }
}
```

Generic (3)

› Esempio

```
public class Contenitore {
  private Object a;
  public Contenitore (Object a) {this.a = a;}
  public Object get() {return a;}
  public void set(Object a) {this.a = a;}
}
public class Test{
       public static void main(String[] args) {
              Contenitore c = new Contenitore(new Automobile());
              Automobile a = (Automobile) c.get();
              c.set("Not an Automobile");
              String s = (String) c.get();
       }
}
```

Generic (4)

› Esempio

```
public class ContenitoreGenerics<T> {
  private T a;
  public Contenitore(T a) {this.a = a;}
  public T get() {return a;}
  public void set(T a) {this.a = a;}
}
public class Test {
  public static void main(String[] args) {
    ContenitoreGenerics<Automobile> c =
                        new ContenitoreGenerics<Automobile>(new Automobile());
    Automobile a = c.get();
    c.set("Not an Automobile"); //Errore compilazione
  }
}
```

Generic (5)

› Esempio

```
public class TwoTuple<A, B> {
  public final A first;
  public final B second;
  public TwoTuple(A a, B b) {
     first = a;
     second = b;
  }
  public String toString() {
    return "(" + first + ", " + second + ")";
  }
}
```

Generic (6)

```
public class ThreeTuple<A, B, C> extends TwoTuple<A, B> {
  public final C third;
  public ThreeTuple(A a, B b, C c) {
    super(a, b);
    third = c;
  }
  public String toString() {
    return "(" + first + ", " + second + ", " + third +")";
  }
}
```

Generic (7)

› Esempio: interfacce

```
public interface Generator<T> {
     T next();
}
```

Generic: Convenzione (8)

› Convenzione per i nomi dei tipi parametrizzati
  – Nome singolo in maiuscolo
  – Nomi più utilizzati
  `E` Element (utilizzato all'interno delle Java Collection Framework)
  `K` key
  `N` Number
  `T` Type
  `V` Value
  `S, U, V` ecc. per il secondo, terzo quarto tipo, ecc.

Generic e collection (1)

- Collection libreria all'interno di Java per memorizzare e gestire dati che sono correlati tra loro
- Liste (array con dimensione variabile), insiemi, mappe, ecc.
- Esempio

```
package java.util;
public interface List<E> extends Collection<E> {
…
}
List<Auto> lista = new ArrayList<Auto>();
```

Metodo dentro interfaccia

`public boolean add(E o)` → `public boolean add(Auto o)`

Generic e collection (2)

```
List<String> lista = new ArrayList<String>();
lista.add("E' possibile aggiungere String");
lista.add(new Date()); //errore in compilazione
```

Tipi primitivi non ammessi

```
List<int> ints = new ArrayList<int>(); //Non ammesso

List<Integer> lista = new ArrayList<Integer>(); //Ammesso
Lista.add(10); //Autoboxing
```

Type Inference

› A partire da Java 7

```
List<String> lista = new ArrayList<String>();
lista.add("E' possibile aggiungere String");
```

E' consigliabile scrivere

```
List<String> lista = new ArrayList<>();
lista.add("E' possibile aggiungere String");
```

Ereditarietà e tipi generici (1)

› Esempio

```
ArrayList<Integer> arrayList = new ArrayList<>();
List<Integer> list = arrayList; //OK


ArrayList<Number> list = arrayList; //NOT OK
```

› `Number` è super classe di `Integer`. Non vuol dire che `ArrayList<Number>` è super classe di `ArrayList<Integer>`

› Idem per

```
ArrayList<Number> list = new ArrayList<Integer>(); //NOT OK
```

Ereditarietà e tipi generici (2)

› Se fosse possibile allora

```
List<Integer> list = new ArrayList<>();
List<Object> objList = list;
//Sto inserendo una stringa dentro una lista di Integer
objList.add("Stringa in un generic di Integer?");
```

Type erasure

- I generic vengono risolti a tempo di compilazione ovvero è il compilatore che rimuove il tipo generico
- E' stato implementato in questo modo per ragioni di compatibilità all'indietro

```
ArrayList<Integer> arrayList = new ArrayList<Integer>();
ArrayList<Number> list = arrayList;
```

Compilatore

```
ArrayList arrayList = new ArrayList();
ArrayList list = arrayList;
```

Wildcard (1)

› Esempio

```
public void printList(List al) {
        Iterator i = al.iterator();
        while (i.hasNext()) {
            Object o = i.next();
            System.out.println(o);
        }
}
```

`Iterator` interfaccia che permette di *navigare* sulla collezione

Compilatore lancia warning perché **non** si stanno usando i generic

Wildcard (2)

› Esempio

```
public void printList(List<Object> al) {
        Iterator<Object> i = al.iterator();
        while (i.hasNext()) {
            Object o = i.next();
            System.out.println(o);
        }
}

List<String> list = new ArrayList<>();
printList(list); //errore in compilazione poiché String!=Object
```

Wildcard (3)

› Esempio

```
public void printList(List<?> al) {
        Iterator<?> i = al.iterator();
        while (i.hasNext()) {
            Object o = i.next();
            System.out.println(o);
        }
        al.add("Stringa"); //Errore in compilazione
}
List<String> list = new ArrayList<>();
printList(list); //OK
```

Nota: Poiché il compilatore non può controllare la correttezza del tipo del parametro in input quando si usa la wildcard, errore in compilazione se si prova ad aggiungere o impostare elementi nella lista (*sola lettura*)

Parametri bounded (1)

```
public class OwnGeneric<E> {
    private List<E> list;
    public OwnGeneric() {
        list = new ArrayList<E>();
    }
    public void add(E e) {
        list.add(e);
    }
    public void remove(int i) {
        list.remove(i);
    }
    public E get(int i) {
        return list.get(i);
    }
……
```

Parametri bounded (2)

```
    public int size() {
        return list.size();
    }
    public boolean isEmpty() {
        return list.size() == 0;
    }
    public String toString() {
        StringBuilder sb = new StringBuilder();
        int size = size();
        for (int i = 0; i < size; i++) {
            sb.append(get(i) + (i != size - 1 ? "-" : ""));
        }
        return sb.toString();
    }
}
```

Parametri bounded (3)

```
class TestOwnGenericWithMinus {
    public static void main(String[] args) {
        OwnGeneric<String> own = new OwnGeneric<>();
        for(int i = 0; i < 10; i++) {
           own.add("" + i);
         }
         System.out.println(own);
    }
}
```

```
$ java TestOwnGenericWithMinus
0-1-2-3-4-5-6-7-8-9
```

Parametri bounded (4)

Altro uso della classe

```
class TestOwnGenericWithMinus {
    public static void main(String[] args) {
        OwnGeneric<String> own = new OwnGeneric<>();
        own.add("-");
        own.add("--");
        own.add("---");
        System.out.println(own);
    }
}
```

```
$ java TestOwnGenericWithMinus
--------
```

Parametri bounded (5)

```
public class OwnGeneric<E extends Number> {
    …
}

OwnGeneric<String> own = new OwnGeneric<>();
//Errore in compilazione

OwnGeneric<Integer> own = new OwnGeneric<>();
//OK Integer è figlio di Number
```

Wildcard bounded

› Esempio

```
public void print(List<? extends Number> list) {
    ……
}
List<Integer> l = new ArrayList<>();
print(l); //OK
List<String> l1 = new ArrayList<>();
print(l1); //NOT OK

public void print(List<? super Integer> list) {
    ……
}
List<Integer> l = new ArrayList<>();
print(l); //OK
List<Number> l1 = new ArrayList<>();
print(l1); //OK
```

Metodi e costruttori generici (1)

- E' possibile parametrizzare, oltre alle classi e/o interfacce, anche solo i metodi all'interno di una classe
- La classe **non** deve essere per forza generica
- Per definire un metodo generico è sufficiente indicare un elenco di parametri generici **prima** del valore di ritorno

Metodi e costruttori generici (2)

› Esempio

```
public class GenericMethod {
    public static <N extends Number> String getValue(N number) {
        String value = number.toString();
        return value;
    }
    public static void main(String[] args) {
        String value = getValue(new Integer(25));
        System.out.println(value);
    }
}
```

Metodi e costruttori generici (3)

› Esempio (costruttore)

```
public class AdvancedInference {
    public <E> AdvancedInference(E e) {
        System.out.println(e);
    }
}

AdvancedInference ai = new AdvancedInference("");
```

Oppure

```
AdvancedInference ai = new <String>AdvancedInference("");
```

Poco usata

Parametri covarianti (1)

› Esempio

```
public class Punto {
      public Punto elaboraPunto() {
            …
      }
}
public class PuntoTridimensionale extends Punto {
      @Override
      public PuntoTridimensionale elaboraPunto() {
            …
      }
}
```

Covariante

Parametri covarianti (2)

› Esempio (parametri di input)

```
public class Punto {
      public double distanza(Punto punto) {
            …
      }
}
public class PuntoTridimensionale extends Punto {
      @Override
      public double distanza(PuntoTridimensionale punto) {
            …
      }
}
```

Parametri covarianti (3)

› Esempio 2

```
interface Cibo {
    String getColore();
}
interface Animale  {
    void mangia(Cibo cibo);
}
public class Erba implements Cibo {
    public String getColore() {
        return "verde";
    }
}
```

Parametri covarianti (4)

```
public class Carnivoro implements Animale {
    public void mangia(Cibo cibo) {
        //un carnivoro potrebbe mangiare erbivori
    }
}
public class Erbivoro implements Cibo, Animale {
    public void mangia(Cibo cibo) {
        //un erbivoro mangia erba
    }
    public String getColore() {
        //. . .
    }
}
```

Erbivoro potrebbero essere `Cibo`

Carnivoro ed Erbivoro potrebbero mangiare **qualsiasi** cosa

Parametri covarianti (5)

```
public class Carnivoro implements Animale {

    public void mangia(Cibo cibo) throws CiboException {

        if (!(cibo instanceof Erbivoro)) {

           throw new CiboException("Un carnivoro deve " +
                                   "mangiare erbivori");

        }

    }

}
```

Parametri covarianti (6)

```
public class Erbivoro implements Cibo, Animale {
    public void mangia(Cibo cibo) throws CiboException {
        if (!(cibo instanceof Erba)) {
           throw new CiboException("Un erbivoro deve " +
                                   "mangiare erba!");
         }
    }
    public String getColore() {
        //. . .
    }
}
```

Parametri covarianti (7)

```
public class CiboException extends Exception {
    public CiboException(String msg){
      super(msg);
    }
    //. . .
}

interface Animale  {
    void mangia(Cibo cibo) throws CiboException;
}
```

Parametri covarianti (8)

```
public class TestAnimali {
    public static void main(String[] args) {
        try {
            Carnivoro tigre = new Carnivoro();
            Cibo erba = new Erba();
            tigre.mangia(erba);
        } catch (CiboException ex) {
            ex.printStackTrace();
        }
    }
}
```

```
$ java TestAnimali
CiboException: Un carnivoro deve mangiare erbivori
        at Carnivoro.mangia(Carnivoro.java:5)
        at TestAnimali.main(TestAnimali.java:8)
```

Parametri covarianti (9): Soluzione

› Ideale

```
public class Carnivoro implements Animale {
    public void mangia(Erbivoro erbivoro) {
        //. . .
    }
}

public class Erbivoro implements Cibo, Animale {
    public void mangia(Erba erba) {
        //. . .
    }
    public String getColore() {
        //. . .
    }
}
```

Parametri covarianti (10): Soluzione

```
interface Animale <C extends Cibo> {
    void mangia(C cibo);
}
public class Carnivoro<E extends Erbivoro> implements Animale<E> {
    public void mangia(E erbivoro) {
        //un carnivoro potrebbe mangiare erbivori
    }
}
public class Erbivoro<E extends Erba> implements Cibo, Animale<E> {
    public void mangia(E erba) {
        //un erbivoro mangia erba
    }
    public String getColore() {
        //. . .
    }
}
```