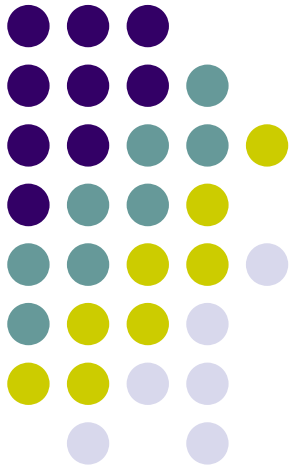
Università di Napoli Federico II - Dip. di Ingegneria Elettrica e delle Tecnologie dell'Informazione
Corso di Laurea in Ingegneria Informatica

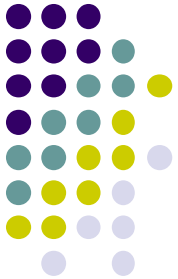
DIETI

Corso di Programmazione

Java aspetti avanzati: tipi generici

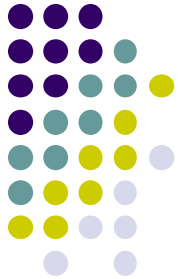


Programmazione Generica



- La programmazione generica consente di sviluppare programmi basati su modelli parametrici di algoritmi e di strutture dati
- Tali modelli sono validi e applicabili al variare dei parametri, e quindi definiscono potenzialmente delle famiglie di algoritmi o di strutture dati specificandone la struttura e il comportamento una sola volta
- In un certo senso è un approccio al riuso del software ***diverso*** da quello praticato mediante l'ereditarietà nel paradigma di programmazione orientato agli oggetti.

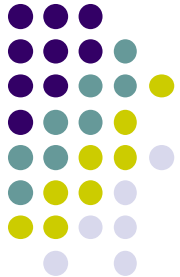
Funzioni e classi generiche



Motivazione: modelli per la costruzione di funzioni o di classi

- procedure che possono essere applicate a strutture dati differenti (ad es. algoritmi di ordinamento)
- classi la cui struttura ed i cui metodi sono gli stessi al variare del tipo o della dimensione delle variabili componenti la struttura dati (ad es. una classe pila)
  - Si deve prediligere (altamente consigliato) l'overloading quando la stessa funzionalità va implementata in modo diverso su tipi diversi.

Che cosa accomuna queste due classi?



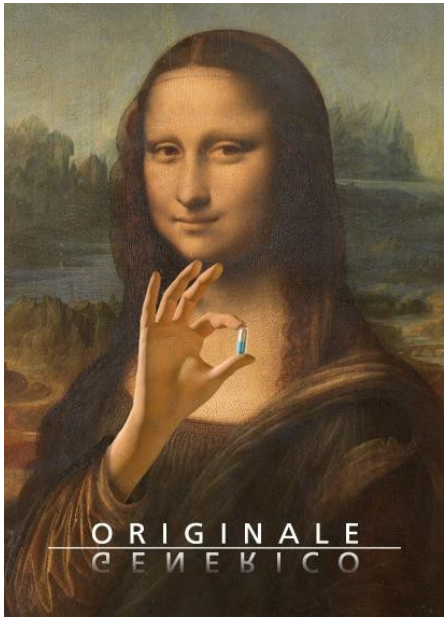
```java
public class CoppiaDiInteri
{
    private int a, b;

    public CoppiaDiInteri(int a, int b)
    {
        this.a = a;
        this.b = b;
    }

    public int getPrimo() { return a; }
    public int getSecondo() { return b; }
}
```

```java
public class CoppiaDiDouble
{
    private double a, b;

    public CoppiaDiDouble(double a, double b)
    {
        this.a = a;
        this.b = b;
    }

    public double getPrimo() { return a; }
    public double getSecondo() { return b; }
}
```

Stesso codice, ma tipi dei campi diversi!

Ci vorrebbe qualcosa di **generico**...

Che cos'è un generico? (1)

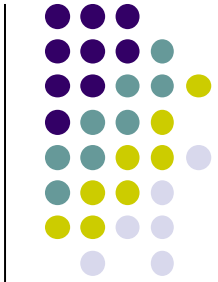


Che cos'è un generico? (1)

- I tipi generici, in Java, sono un modello di programmazione che permette di definire, con una sola dichiarazione, un intero insieme di metodi o di classi
- Un meccanismo MOLTO potente
- Da usare con consapevolezza


Cup<T>

Utilizzerete ampiamente i generici!



- In tutte le collezioni Java!
- Creare istanze di classi con generici:

```
new ArrayList<String>();
```

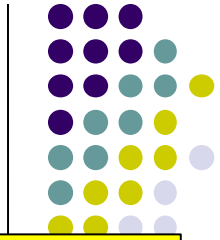
- Dichiarare e assegnare variabili di tipi generici

```
List<String> listaDiStringhe = new ArrayList<String>();
```

- Dichiarare metodi che prendono in input tipi generici

```
public void metodo(List<String> lista)
{
    // ...
}
```

Esempio di classe generica



```
public class Valore <T> {
    private final T val;
    public Valore(T val) {this.val=val;}
    public T get() {return val;}
    @Override
    public String toString() { return ""+val;}
    public String getType() {return val.getClass().getName();
    }
}
```

Definisce un tipo generico della classe

- Per definire un tipo generico della classe, si utilizza la sintassi a parentesi angolari dopo il nome della classe con il tipo generico da utilizzare
- Da quel punto, si utilizza il tipo generico come un qualsiasi altro tipo di classe

Esempio di classe generica

```
public class Valore <T> {
    private final T val;
    public Valore(T val) {this.val=val;}
    public T get() {return val;}
    @Override
    public String toString() { return ""+val;}
    public String getType() {return val.getClass().getName();
    }
}
```

Usa il tipo generico della classe

- Per definire un tipo generico della classe, si utilizza la sintassi a parentesi angolari dopo il nome della classe con il tipo generico da utilizzare
- Da quel punto, si utilizza il tipo generico come un qualsiasi altro tipo di classe

Istanziare la classe generica

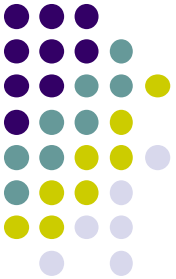
```
public static void main(String[] args) {
    Valore<Integer> i = new Valore<>(42);
    Valore<String> s = new Valore<>("Napoli");
    Valore<Double> d = new Valore<>(77.11);

    System.out.println(i.get()+":"+i.getType());
    System.out.println(s.get()+":"+s.getType());
    System.out.println(d.get()+":"+d.getType());
}
```

Inferenza automatica del tipo generico

- L'output sarà:

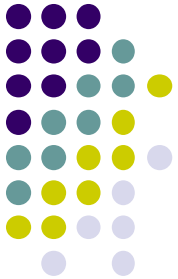
```
run:
42:java.lang.Integer
Napoli:java.lang.String
77.11:java.lang.Double
BUILD SUCCESSFUL (total time: 0 seconds)
```

Altro esempio: Coppia di elementi di tipo generico



```
public class Coppia <T> {
    private T a, b;
    public Coppia (T a, T b) {
        this.a = a;
        this.b = b;
    }

    public T getPrimo() {return a;}
    public T getSecondo() {return b;}

}
```

Esempio: Utilizzare la classe coppia



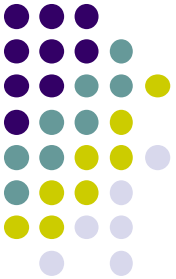
- Semplicemente si istanzia la classe specificando il tipo desiderato:

```
public static void main(String[] args) {
    Coppia<Integer> ci = new Coppia<>(10,20);
    Coppia<Double> cd = new Coppia<>(10.5,20.5);
    Coppia<String> cs = new Coppia<>("abc", "def");
    Coppia<Object> co = new Coppia<>("abc", 20);

    System.out.println(ci.getPrimo());
    System.out.println(cd.getSecondo());
    System.out.println(cs.getSecondo());
    System.out.println(co.getPrimo()+" : "+ co.getSecondo());

}
```

```
run:
10
20.5
def
abc : 20
BUILD SUCCESSFUL (total time: 0 seconds)
```

- ...e chi più ne ha più ne metta!

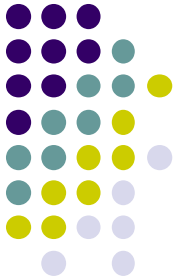
Specificare più tipi generici di classe



- Specifichiamo i tipi generici separati da virgola
- Per convenzione i tipi generici sono chiamati con le lettere **T**, **S**, ecc. (**E** nel caso in cui siano elementi di una collection)

```
public class CoppiaGenerici <T, S> {

    private final T a;
    private final S b;

    public CoppiaGenerici (T a, S b) {
        this.a = a;
        this.b = b;
    }

    public T getPrimo() {return a;}
    public S getSecondo() {return b;}

}
```

Esempio d'uso

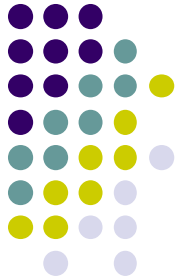


```
public static void main(String[] args) {
    CoppiaGenerici<Integer,Double> cg = new CoppiaGenerici<> (4, 5.5);
    System.out.println(cg.getPrimo()+" - " +cg.getSecondo());
}
```

In esecuzione:

```
run:
4 - 5.5
BUILD SUCCESSFUL (total time: 0 seconds)
```

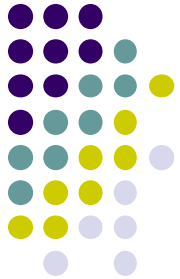
I generici funzionano solo con i wrapper



- Non è possibile utilizzare tipi primitivi, ad es. int, double, char, ecc.
- Es:

```
Valore<int> v;
Valore<double> v;
```

Estendere le classi generiche



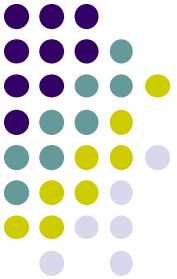
- Ovviamente è possibile estendere le classi generiche per creare classi più specifiche
- Ad esempio, una classe Orario può estendere la classe Coppia:

```
public class Orario extends Coppia<Integer, Integer>
{
    public Orario(Integer a, Integer b)
    {
        super(a, b);
    }
}
```

- O una classe Data:

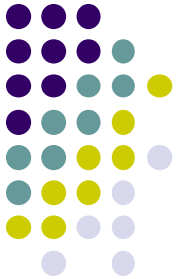
```
public class Data extends Coppia<Integer, Coppia<Integer, Integer>>
{
    public Data(Integer giorno, Integer mese, Integer anno)
    {
        super(giorno, new Coppia<Integer, Integer>(mese, anno));
    }
}
```

Esempio



- Vogliamo implementare una camera da montare sulle automobili a guida autonoma che può "riconoscere" ostacoli.
- Esistono diversi ostacoli:
  - Bici, pedoni, segnali, etc..
  - Di fatto la camera deve poter riconoscere degli ostacoli generici.
  - Per ora consideriamo pedoni e bici.

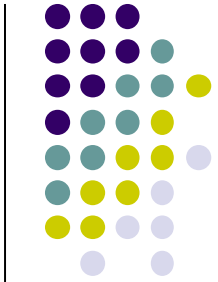
Le classi ostacolo concreto...



```
public class Bici {
    @Override
    public String toString(){
        return " un ostacolo bici";
    }
}
```

```
public class Pedone {
    @Override
    public String toString(){
        return " un ostacolo pedone";
    }
}
```

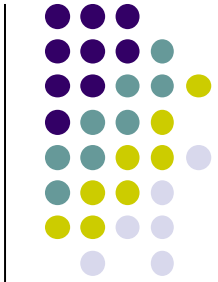
Definiamo il tipo generico ostacolo



- Lo possiamo implementare indipendentemente dalle classi base definite in precedenza

```
public class Ostacolo<T> {
    private final T tipologia;

    public Ostacolo(T t){
            tipologia=t;
    }
    public T getTipoOstacolo() {
            return tipologia;
    }
}
```

Ora definiamo la classe Camera...



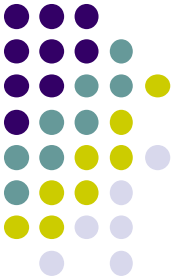
- La classe camera ha un metodo che permette di riconoscere il tipo di ostacolo:

```
public class Camera {
    public void riconosciOstacolo(Ostacolo<?> ostacolo){
        System.out.println("Ho riconosciuto"+ostacolo.getTipoOstacolo());
    }
}
```

Il carattere Jolly (?) ha permesso di realizzare un unico metodo `riconosciOstacolo()` in grado di ricevere in input ostacoli di diverse tipologie.

Senza il carattere Jolly saremmo stati costretti a definire un metodo per ciascuno tipo di Ostacolo.

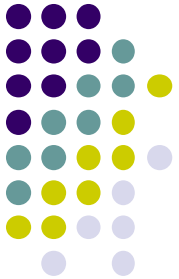
In esecuzione



```
public static void main(String[] args) {
    Ostacolo<Pedone> ostacolo1 = new Ostacolo<>(new Pedone());
    Ostacolo<Bici> ostacolo2 = new Ostacolo<>(new Bici());
    Camera videoCamera = new Camera();
    videoCamera.riconosciOstacolo(ostacolo1);
    videoCamera.riconosciOstacolo(ostacolo2);
}
```

```
run:
Ho riconosciuto un ostacolo pedone
Ho riconosciuto un ostacolo bici
BUILD SUCCESSFUL (total time: 0 seconds)
```

Riferimenti



- Programmare in Java
  - Cap. 20 per definizioni e approfondimenti