Generics

Marco Patrignani

mailto: marco.patrignani@unitn.it

(basato sulle slides di Picco, Ronchetti, Marchese)

Cast a volontà…

```
public class Pila {
   public Object estrai() { … }
   public void inserisci(Object o) { … }
}

…
Pila p = new Pila(10);
p.inserisci("5");
…
Integer x = (Integer) p.estrai();
```

Strutture dati polimorfe richiedono conversioni di tipo ***al momento dell'estrazione***

Ciò può generare ***errori di tipo a runtime!***

Generics

Consentono di specificare un ***tipo come parametro*** …

```
public class Pila<T> {
   public T estrai() { … }
   public void inserisci(T o) { … }
}
…
Pila<Integer> p = new Pila<Integer>(10);
p.inserisci("5");
Integer x = (Integer) p.estrai();
```

… evitando la necessità di cast espliciti …

… e consentendo di identificare gli errori ***a compilation time***

```
error: incompatible types: String cannot be converted to Integer
    p.inserisci("5");
```

Generics in Java

- Presenti a partire da Java 5
- Analoghi concetti in altri linguaggi
  - Es., template in C++ + ML, Rust, OCaml
- Consentono la definizione di:
  - ***tipo generico***: una classe o interfaccia la cui definizione include uno o più tipi come parametro
  - ***metodo generico***: include la dichiarazione di uno o più tipi usati come parametro

Tipi generici

- **`class name<T1, T2, ..., Tn>`** dove **`T1`**…**`Tn`** sono i tipi con cui la classe è parametrizzata
  - Analogo per le interfacce
- Il tipo «attuale» deve essere specificato all'atto della dichiarazione

```
class Group<T> { …        //definizione
Group<Student> gs = …  //uso
Group<Tourist> gt = …  //uso
```

Esempio

tipo generico

I «parametri tipo» sono visibili nell'intera definizione della classe

... dove sono usati come un qualsiasi altro tipo (a parte alcuni casi particolari)

```
class Pair<X,Y> {
  private X first;
  private Y second;
  public Pair(X a1, Y a2) {
    first  = a1;
    second = a2;
  }
  public X getFirst()  { return first; }
  public Y getSecond() { return second; }
  public void setFirst(X arg)  {first = arg;}
  public void setSecond(Y arg) {second = arg;}
}
```

«argomenti tipo»: rimpiazzano i parametri all'atto della dichiarazione

«*diamond operator*» (Java 7): gli argomenti possono essere omessi e inferiti dal compilatore

```
Pair<String,Double> p = new Pair<String,Double>("PI", 3.14);
Pair<String,Double> p = new Pair<>("PI", 3.14);
```

Esempio

`Insieme` può contenere un qualsiasi **`Object`**: nessuna garanzia sul fatto che i due insiemi contengano oggetti dello stesso tipo

```
public static Insieme unione(Insieme s1,
                             Insieme s2) {
  Insieme u = new Insieme(s1);
  u.aggiungiTutti(s2);
  return u;
}
```

dichiarazione dell'argomento tipo, locale al metodo

```
public static <E> Insieme<E> unione(Insieme<E> s1,
                                    Insieme<E> s2) {
  Insieme<E> u = new Insieme<>(s1);
  u.aggiungiTutti(s2);
  return u;
}
```

metodo generico

Esempio

```
class Pair<X,Y> {
  private X first;
  private Y second;
  public Pair(X a1, Y a2) {
    first  = a1;
    second = a2;
  }
  public X getFirst()  { return first; }
  public Y getSecond() { return second; }
  public void setFirst(X arg)  {first = arg;}
  public void setSecond(Y arg) {second = arg;}
}
class Test {
  public static void main(String[] args) {
    Pair<String, Double> pair =
      new Pair<>("PI", 3.14);
    String s = pair.getFirst();
    Double d = pair.getSecond();
    Object o = pair.getSecond();
  }
}
```

```
public class Pair {
  private Object first;
  private Object second;
  public Pair(Object a1,
              Object a2) {
    first = a1;
    second = a2;
  }
  public Object getFirst() { return first; }
  public Object getSecond() { return second; }
  public void setFirst(Object arg) { first = arg; }
  public void setSecond(Object arg) { second = arg; }
}
final class Test {
  public static void main(String[] args) {
    Pair pair = new Pair("PI", 3.14L);
    String s = (String) pair.getFirst();
    Double d = (Double) pair.getSecond();
    Object o =          pair.getSecond();
  }
}
```

Generics e sottotipi

- Dati due tipi generici **G<A>** e **G<B>** dove **B** è una sottoclasse di **A**, ***non*** è vero che **G<B>** è una sottoclasse di **G<A>**
- In altre parole: fra tali tipi generici non vale il principio di sostituzione
- Esempio:

```
Person s = new Student();                    //ok
Person[] s = new Student[100];               //ok
Group<Person> g = new Group<Student>();      //no!
```

- Motivo: **G<A>** e **G<B>** sono sottoclassi di **Object**
  - In realtà sono la ***stessa*** sottoclasse **G**

… perché?

- I generics in Java sono implementati mediante ***type erasure***
  - L'informazione sui parametri tipo esiste solo a compilation time
  - Viene eliminata dopo i controlli statici e dopo aver inserito gli opportuni cast per mantenere i vincoli sul tipo
- Questa scelta riduce l'overhead a runtime e mantiene compatibilità con API che non usano generics …
  - Es. Java Collection framework pre-Java 5
- … ma genera una serie di limitazioni

Generics e sottotipi: *wildcard*

- … ma allora come definire strutture dati generiche e allo stesso tempo polimorfe?
  - Ad esempio:

```
Group<Object> g = new Group<Student>();
```

darebbe errore in compilazione…

- È possibile specificare una o più *wildcard*
  - Rappresentano un tipo non noto

```
Group<?> g = new Group<Student>();
```

  - Non possono essere usate per creare oggetti

```
Group<?> g = new Group<?>(); // no!
```

- Utile ad esempio per realizzare metodi generici…

Esempio

L'intento è quello di stampare un insieme contenente qualsiasi tipo di oggetto

```
static void stampaInsieme(Insieme<Object> set) {
  //ciclo con variabile elem su set
    System.out.println(elem + " ");
  …
}
```

In realtà consente di stampare solo liste di `Object`

```
Insieme<Person> sp = new Insieme<>();
… // popolo l'insieme
stampaInsieme(sp);  // errore in compilazione!
```

La wildcard risolve il problema consentendo subtyping

```
static void stampaInsieme(Insieme<?> set) {
  …
```

Bounded wildcards e bounded types

- È possibile «limitare» le wildcard specificando che sono accettati solo sottotipi di un tipo dato

```
Group<? extends Persona> g
            = new Group<Studente>();
```

- Si usa **`extends`** anche con le interfacce

```
Group<? extends Volatile> g
            = new Group<Rondine>();
```

- È possibile specificare più di un tipo:
se c'è una classe deve essere la prima

```
Group<? extends Animale & Volatile
                        & Acquatico> g
            = new Group<Anatra>();
```

- La stessa notazione si può usare per i parametri (*bounded types*) nella definizione di tipi e metodi generici

```
class Squadra<T extends Atleta> { …
static <T extends Number> double average(T[] p) { …
```

Esempio

Svuota una pila generica e ne ritorna la somma degli elementi, i quali sono vincolati a essere numeri

```
static double svuotaAggrega(Pila<? extends Number> p){
  double s = 0.0;
  int dim = p.numElementi();
  for (int i=0; i<dim; i++)
    s += p.estrai().doubleValue();
  return s;
}
Pila<Integer> pi = new Pila<>(); // [0,1,2,3]
Pila<Double> pd = new Pila<>();  // [0.0,1.1,2.2,3.3]
System.out.println(svuotaAggrega(pi)); // 6.0
System.out.println(svuotaAggrega(pd)); // 6.6
Pila<String> ps = new Pila<>();  // ["0","1","2","3"];
System.out.println(svuotaAggrega(ps)); // compilation
                                       // error
```

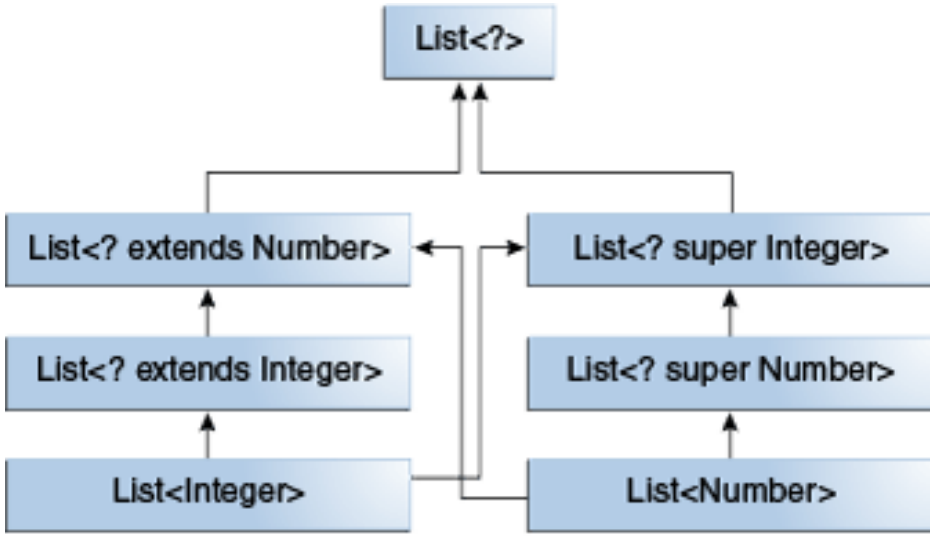
`estrai` ritorna un `Number`: posso invocarne i metodi!

Generics e sottotipi: *bounded wildcards*

- È possibile «limitare» le wildcard anche verso le superclassi

```
Group<? super Studente> g
                = new Group<Persona>();
```

- Le regole di tipo diventano complicate…
- Nota: non è possibile con i parametri bounded type



Una Pila generica?

```
public class Pila<T> {
   int size;
   int defaultGrowthSize;
   int marker;
   T[] contenuto;
   public PilaGenerica() {
       size = initialSize;
       defaultGrowthSize = initialSize;
       marker = 0;
       contenuto = new T[initialSize];
   }
   public T estrai() { … }
   public void inserisci(T o) { … }
   ...
}
```

errore in compilazione:
generic array creation

Generics e array

- Non si può creare un array generico:
  - Es.: tutte queste espressioni sono illegali:
    ```
    new Group<T>[…]
    new Group<Persona>[…]
    new T[…]
    ```
- Motivo: non sarebbe type safe e quindi minerebbe il vantaggio principale dei generics
  - i tipi generici alla fine diventano tutti **`Object`**…
- Se necessario, è possibile usare un cast
  - Ma non è desiderabile, per lo stesso motivo
- In generale, meglio usare **`List`**
  - Più flessibili e generiche
  - Parte del Java Collection Framework (**`java.util`**)

Altre limitazioni dei generics

- I parametri tipo non possono essere tipi primitivi
  - Es. `Pila<int>`; ma con wrapper e autoboxing solitamente non è un problema
- Generics non si possono usare con **`instanceof`**

```
Object obj = new Group<Studente>();
obj instanceof Group
obj instanceof Group<?>
obj instanceof Group<Studente>                  //error
obj instanceof Group<? extends Persona> //error
obj instanceof Group<? super Liceale>   //error
```

- Svariati altri casi particolari …
- Per chi vuole approfondire:
  http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html