Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Università degli Studi dell'Aquila

Corso di Algoritmi e Strutture Dati con Laboratorio

La libreria standard: Generics

Generics

- La versione 1.5 ha introdotto una nuova, importante funzionalità nel linguaggio: la programmazione parametrica, in Inglese anche detta ***generics***.
- Si tratta della possibilità di **specificare il tipo di un elemento dotando classi, interfacce e metodi di parametri di tipo**. Questi parametri hanno come possibili valori i tipi del linguaggio.
  - In particolare, possono assumere come valore qualsiasi tipo, esclusi i tipi primitivi (tipi base).

Verso i tipi generici: un esempio

```
interface ListOfNumbers {
  boolean add(Number element);
  Number get(int index);
}
interface ListOfIntegers {
  boolean add(Integer element);
  Integer get(int index);
}
      … e ListOfStrings …

// Tipo generico con parametro di tipo:
  interface List<E> {
  boolean add(E n);
  E get(int index);
}
```

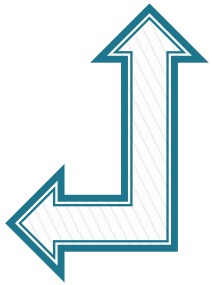
Possiamo istanziare il tipo generico usando qualunque tipo come parametro attuale

List<Integer>

List<Number>

List<String>

List<List<String>> …



Verso i tipi generici: un esempio

```
interface ListOfIntegers {
     boolean add(Integer element);
     Integer get(int index);
}
```

- Dichiarazione di un parametro di tipo
  - Istanziabile con un qualunque tipo
  - `List<Integer>`, `List<String>`, …

```
interface List<E> {
     boolean add(E elt);
     E get(int index);
}
```

Dichiarazione di generics

```
class Name<TypeVar1, …, TypeVarN> {…}

interface Name<TypeVar1, …, TypeVarN> {…}
```

- Convenzioni:

T per **Type, E** per **Element** (generalmente per elementi di una collezione), **K** per **Key, V** per **Value**

- Istanziare una classe generica significa fornire un valore di tipo

```
Name<Type1, …, TypeN>
```

Tipi parametrici

- All'interno di una classe, un parametro di tipo si comporta (tranne poche eccezioni) come un tipo di dati vero e proprio
  - In particolare, un parametro di tipo si può usare come **tipo di un campo**, **tipo di un parametro formale di un metodo** e **tipo di ritorno di un metodo**

```
class ClasseTipo <T> {
      T attributo;
      public ClasseTipo (T x) {attributo = x;}
      public T getValue() {return attributo;}
}
```

Tipi parametrici

- Uso della classe parametrica `ClasseTipo`

```
public class Prova {
      public static void main(String[] s){
      ClasseTipo<String> p1 = new ClasseTipo<String>(s[0]);
      ClasseTipo<Integer> p2 = new ClasseTipo<Integer>(10);
      String a = p1.getValue();
      System.out.println(a);
      Integer b = p2.getValue();
      System.out.println(b);
      }
}
```

Classi parametriche: uso

- Usare un tipo parametrico significa istanziare la classe per creare riferimenti ad oggetti. Es:

```
LinkedList<Integer> intList =
                    new LinkedList<Integer>();
```

- Tutte le occorrenze dei parametri formali sono rimpiazzate dall'argomento (parametro attuale)
- Diversi usi generano tipi diversi
- Le classi parametriche sono compilate una sola volta e danno luogo ad un unico file .class

Esempio

- La programmazione parametrica dimostra tutta la sua utilità nella realizzazione di **collezioni**, ovvero classi deputate a contenere altri oggetti

```
public class ArrayList<E>
```

- Per creare un'istanza, **`numberList`**, della classe **`ArrayList`**, i cui elementi siano di tipo (referenza a) **`Double`**:

```
ArrayList<Double> numberList = new
        ArrayList<Double>();
```

Un esempio di uso

- Inserimento

```
numberList.add(new Double(2.7));
numberList.add(2.7);
```

- Accesso:

```
Double wrapValue = numberList.get(0);
```

- Uso in un'espressione:

```
Sum = sum + wrapValue.doubleValue();
Sum = sum + wrapValue;
```

Tipi parametrici

- **Vincoli di tipo**

```
<TypeVar extends SuperType>
```

  - upper bound: va bene il supertype o uno dei suoi sottotipi

```
<TypeVar super SubType>
```

  - lower bound: va bene il sottotipo o uno qualunque dei suoi supertipi

- Esempio: strutture di ordine su alberi

```
public class TreeSet<T extends Comparable<T>>
```

Tipi parametrici

```
class Name<TypeVar1 extends Type1,
           …, TypeVarN extends TypeN> {…}
```

- Analogo per le interfacce
- Notare che Object è il limite superiore di default nella gerarchia dei tipi
- Istanziazione identica

```
Name<Type1, …, TypeN>
```

- Compile-time error se il tipo non è un sottotipo del limite superiore della gerarchia

Tipi parametrici

```
interface List1<E extends Object> {…}
interface List2<E extends Number> {…}

List1<Date>
// OK perché Date è un sottotipo di Object

List2<Date>
// compile-time error
// Date non è sottotipo di Number
```

Tipi parametrici

- Si possono effettuare tutte le operazioni compatibili con il limite superiore della gerarchia
- concettualmente questo corrisponde a forzare una sorta di precondizione sulla istanziazione del tipo

```
class List1<E extends Object> {
      void m(E arg) {
      arg.asInt( );
      // compiler error, E potrebbe non avere asInt}
}

class List2<E extends Number> {
      void m(E arg) {
      arg.asInt( );
// OK, Number e i suoi sottotipi supportano asInt }
}
```

Tipi parametrici

Vantaggio:

- Questo meccanismo consente di scrivere **codice più robusto** dal punto di vista dei tipi di dato (fornisce una migliore gestione del **type checking** durante la compilazione), evitando in molti casi il ricorso al casting da `Object`

Tipi parametrici: un esempio

- Realizzare una classe **`Pair`**, che rappresenta una coppia di oggetti dello stesso tipo.
- In mancanza della programmazione parametrica (ad esempio, in Java 1.4) la classe `Pair` si sarebbe dovuta realizzare secondo il seguente schema:

```
class Pair {
  private Object first, second;
  public Pair(Object a, Object b) { ... }
  public Object getFirst() { ... }
  public void setFirst(Object a) { ... }
  ...
}
```

Tipi parametrici: un esempio

- Gli utenti della classe devono ricorrere al cast perché gli elementi estratti dalla coppia riacquistino il loro tipo originario, come nel seguente esempio:

```
Pair p = new Pair("uno", "due");
String a = (String) p.getFirst();
```

Tipi parametrici: un esempio

Rendiamo la classe `Pair` parametrica:

```
class Pair<T> {
  private T first, second;
  public Pair(T a, T b) {
     first = a;
     second = b;
  }
  public T getFirst() { return first; }
  public void setFirst(T a) { first = a; }
  ...
}
```

- La classe `Pair` ha un parametro di tipo, `T`

Tipi parametrici: un esempio

- La versione parametrica di **`Pair`** permette agli utenti della classe di specificare di che tipo di coppia si tratta e dunque di evitare i cast:

```
Pair<String> p = new Pair<String>("uno","due");
String a = p.getFirst();
```

- Sia nella dichiarazione della variabile `p`, sia nell'instanziazione dell'oggetto `Pair` va indicato il parametro di tipo desiderato
- Come per i normali parametri dei metodi, `String` è il parametro attuale, che prende il posto del parametro formale `T` di `Pair`

Tipi parametrici

- Per compatibilità con le versioni precedenti di Java, è possibile usare una classe (o interfaccia) parametrica come se non lo fosse
- Quando utilizziamo una classe parametrica senza specificare i parametri di tipo, si dice che stiamo usando la **versione grezza** di quella classe
  - Es: *l'interfaccia grezza Comparable vista in precedenza*!
- La versione grezza di queste classi permette alla nuova versione della libreria standard di essere compatibile con i programmi scritti con le versioni precedenti del linguaggio
  - Le classi grezze esistono solo per retro-compatibilità

Tipi parametrici

- Ad esempio, se **Pair** è la classe parametrica descritta in precedenza, è possibile utilizzarla così:

```
Pair p = new Pair("uno", "due");
String a = (String) p.getFirst();
```

- La prima riga provoca un warning in compilazione
- Il cast nella seconda riga è indispensabile

NB: Il codice nuovo dovrebbe sempre specificare i parametri di tipo delle classi parametriche

```
new Pair<String>(…)
```

Tipi parametrici: un esempio

- Esaminiamo un'ulteriore versione di `Pair`, in grado di contenere due oggetti di tipo diverso

```
class Pair<T, U> {  // due parametri tipo
  private T first;
  private U second;
  public Pair(T a, U b) {
      first = a; second = b;
  }
  public T getFirst() { return first; }
  public void setFirst(T a) { first = a; }
  public U getSecond() { return second; }
  public void setSecond(U a) { second = a; }
}
```

Interfacce parametriche

- Anche **le interfacce** possono essere dichiarate come parametriche o generiche, come:

```
public interface Comparable<T> {
  int compareTo (T o);
}
```

- Ad esempio la classe **`String`** implementa **`Comparable<String>`**

Metodi parametrici

- Anche i singoli metodi e costruttori possono avere parametri di tipo, indipendentemente dal fatto che la classe cui appartengono sia parametrica o meno
  - I metodi statici non possono utilizzare i parametri di tipo della classe in cui sono contenuti
  - Il parametro di tipo va dichiarato prima del tipo restituito, racchiuso tra parentesi angolari
  - Questo parametro è visibile solo all'interno del metodo

Metodi parametrici

- Il seguente **metodo parametrico** restituisce l'elemento mediano (di posto intermedio) di un dato array

```
public static <T> T getMedian(T[] a) {
int l = a.length;
return a[l/2];
}
```

- In questo caso, il parametro di tipo permette di restituire un oggetto dello stesso tipo dell'array ricevuto come argomento

Metodi parametrici

- Quando si invoca un metodo parametrico, è opportuno, ma non obbligatorio, specificare il parametro di tipo attuale per quella chiamata
- Ad esempio, supponendo che il metodo **`getMedian`** appartenga ad una classe **`Test`**, lo si può invocare così:

```
String[] x = {"uno", "due", "tre"};
String s = Test.<String>getMedian(x);
```

- Il parametro attuale di tipo va quindi indicato prima del nome del metodo

Metodi parametrici

- È possibile omettere il parametro attuale di tipo. In questo caso, il compilatore cercherà di dedurre il tipo più appropriato, mediante un meccanismo chiamato ***type inference*** (inferenza di tipo)
- La type inference cerca di individuare il tipo più specifico che rende la chiamata corretta
- L'algoritmo di type inference non è né corretto né completo
  - *Le regole precise che il compilatore adotta nella type inference esulano dagli scopi di questo corso*

Metodi parametrici

- Anche i **costruttori** possono essere parametrici, indipendentemente dal fatto che la loro classe sia parametrica o meno

```
Public class NomeClasse<T> {
   Public <U> NomeClasse(T x, U y) { ... }
     … }
```

  - Il costruttore della classe parametrica **`NomeClasse`** ha a sua volta un **parametro di tipo chiamato `U`**
  - Mentre il parametro **`T`** è visibile in tutta la classe **`A`**, il parametro **`U`** è visibile solo all'interno di quel costruttore

Metodi parametrici

- Il costruttore in questione può essere invocato con la seguente sintassi

```
NomeClasse<String> a =
new <Integer>NomeClasse<String>("ciao", new
    Integer(100));
```

- Il parametro di tipo del costruttore (**`Integer`**) va specificato prima del nome della classe
- Il parametro di tipo della classe, come abbiamo già visto per la classe **`Pair`**, va specificato dopo il nome della classe.