Programmazione orientata agli oggetti

Generics: concetti base

Introduzione

- A partire dalla versione 1.5 di Java sono stati introdotti i Generics
- La motivazione principale è quella di introdurre costrutti per rafforzare il controllo sui tipi a tempo di compilazione

Controllo sui tipi

- Considereremo di seguito del codice che fa riferimento alla seguente classe **`Persona`**

```
import java.util.*;

class Persona {
  private String nome;

  public Persona(String nome) {
    this.nome = nome;
  }

  public String toString() {
    return this.nome;
  }
}
```

Controllo sui tipi

- Consideriamo il seguente codice: compila e gira correttamente

```
public class CollezioniTest{
  @Test
  public void testCheCompilaEgira() {
     List elencoPersone = new ArrayList();
     Persona p1 = new Persona("Pippo");
     elencoPersone.add(p1);
     Persona p2 = new Persona ("Pluto");
     elencoPersone.add(p2);
     Iterator it = elencoPersone.iterator();
     while (it.hasNext()) {
        Persona persona = (Persona)it.next();
        System.out.println(persona);
    }
  }
}
```

Controllo sui tipi

- Consideriamo il seguente codice: compila correttamente ma l'esecuzione fallisce

```
public class ProblemiConCollezioniTest {
  @Test
  public void testCheCompilaMaNonGira() {
     List elencoPersone = new ArrayList();
     Persona p1 = new Persona("Pippo");
     elencoPersone.add(p1);
     String p2 = new String("Pluto");
     elencoPersone.add(p2);
     Iterator it = elencoPersone.iterator();
     while (it.hasNext()) {
        Persona persona = (Persona)it.next();
        System.out.println(persona);
    }
  }
}
```

Solleva un errore a tempo di esecuzione!

Introduzione

- Abbiamo visto qual è il principale limite delle collezioni pre-Generics
  - un controllo lasco dei tipi a tempo di compilazione ha queste conseguenze
    - ci costringe a fare un cast ogni volta che accediamo ad un elemento della collezione (ma questo non è l'aspetto peggiore)
    - rimanda a tempo di esecuzione alcuni errori che tutto sommato erano rilevabili anche a tempo di compilazione
- Vediamo come questi problemi sono stati affrontati e risolti con i Generics

Obiettivi della lezione

- I generics sono uno strumento per scrivere classi e metodi parametrici rispetto ad un tipo
- Ci concentriamo soprattutto su come usare classi generiche
  - Al temine del corso lo studente dovrà essere in grado (cioè verrà chiesto all'esame) di usare classi generiche (in particolare quelle del package **`java.util`**)
- La progettazione di classi generiche va oltre gli obiettivi del corso
  - Però, da un punto di vista didattico è utile introdurre i Generics progettando una semplice classe

Introduzione

- Supponiamo di dover scrivere una classe **`Coppia`**, che consente di gestire coppie di oggetti dello stesso tipo
- Vogliamo una classe generica, che possa essere usata in contesti diversi. Ad esempio per gestire:
  - Coppie di stringhe (istanze della classe **`String`**)
  - Coppie di attrezzi (istanze della classe **`Attrezzo`**)
  - Coppie di URL (istanze della classe **`URL`**)
  - ...

La classe generica `Coppia`

- La classe **`Coppia`** deve offrire:
  - Un metodo per ottenere il primo elemento della coppia
  - Un metodo per ottenere il secondo elemento della coppia
  - Un costruttore che prende come parametri due riferimenti ad oggetti dello stesso tipo

La classe generica `Coppia`

- La definizione di una classe generica prevede la dichiarazione del parametro di tipo racchiuso tra parentesi acute

```
public class Coppia<T> {
    …
}
```

- In questo modo abbiamo detto che, all'interno della classe **`Coppia,`** ogni volta che usiamo il simbolo T stiamo indicando il tipo rispetto a cui la classe è parametrica

La classe generica `Coppia`

- All'interno della classe, T viene usato come una dichiarazione di tipo per la definizione di campi e metodi

```
public class Coppia<T> {
	private T primo;
	private T secondo;

	public Coppia(T primo, T secondo) {
		this.primo = primo;
		this.secondo = secondo;
	}
	public T getPrimo() {
		return this.primo;
	}
	public T getSecondo() {
		return this.secondo;
	}
	…
}
```

La classe generica Coppia

```
public class Coppia<T> {
    private T primo;
    private T secondo;

    public Coppia(T primo, T secondo) {
      this.primo = primo;
      this.secondo = secondo;
    }
    public T getPrimo() {
      return this.primo;
    }
    public T getSecondo() {
      return this.secondo;
    }
    public void setPrimo(T primo) {
      this.primo = primo;
    }
    public void setSecondo(T secondo) {
      this.secondo = secondo;
    }
}
```

Usare una classe generica

- Quando usiamo una classe generica, dobbiamo *istanziare il tipo*
- Ad esempio, usiamo la nostra classe generica **`Coppia`**, per gestire coppie di oggetti **`Persona`**

```
import omessi
public class CoppiaTest {
   @Test
   public void testDiCoppiaDiPersone() {
      Coppia<Persona> coppia;
      Persona p1 = new Persona("Stanlio");
      Persona p2 = new Persona("Olio");
      coppia = new Coppia<Persona>(p1, p2);
      assertSame(p1,coppia.getPrimo());
      assertSame(p2,coppia.getSecondo());
   }
}
```

Usare una classe generica

- Vediamo la classe parametrica **`Coppia<T>`** istanziata su un altro tipo (**`java.awt.Color`**)

```
import java.awt.Color;
altri import omessi
public class CoppiaTest {
   @Test
    public void testDiCoppiaDiColori() {
      Coppia<Color> coppia;
      Color rosso = new Color(255,0,0);
      Color blue = new Color(0,0,255);
      coppia = new Coppia<Color>(rosso, blue);
      assertSame(rosso,coppia.getPrimo());
      assertSame(blue,coppia.getSecondo());
  }
}
```

Controllo sui tipi

- Se proviamo ad usare tipi diversi da quelli istanziati, si genera un errore a tempo di compilazione

```
import …
public TestCoppia {
    @Test
    public void testScambiaCoppia_nonCompila() {
      Coppia<Persona> coppia;
      Persona p1 = new Persona("Stanlio");
      String p2 = new String("Olio");
      coppia = new Coppia<Persona>(p1, p2);
      assertSame(p1,coppia.getPrimo());
      assertSame(p2,coppia.getSecondo());
      UtilitàCoppie.scambiaElementi(coppia);
      assertSame(p2,coppia.getPrimo());
      assertSame(p1,coppia.getSecondo());
    }
}
```

Tipo Formale – Tipo Attuale

- Non è difficile trovare una similitudine tra
  - il concetto di parametro formale/attuale inerente l'invocazione dei metodi
  - il concetto di tipo formale/attuale inerente la tipizzazione di classi generiche
- Attenzione a non dimenticare la prima delle differenze
  - il legame tra parametri formali/attuali è operato dalla JVM a tempo di esecuzione
  - il legame tra tipi formali/attuali è operato dal compilatore a tempo di compilazione

Metodi generici

- È possibile definire anche metodi generici (cioè parametrici rispetto ad un tipo)
- Un metodo generico definisce i parametri (formali) di tipo nella segnatura del metodo, prima del tipo di ritorno

```
static <T> int mioMetodo(Coppia<T> c, T obj)
```

Metodi generici: esempio

- Supponiamo di voler definire la classe **`UtilitàPerCoppie`**, che definisce un insieme di metodi di utilità per la classe **`Coppia`**
- Ad esempio vogliamo definire il metodo statico *scambiaElementi*, che prende come parametro una coppia *c* e ne scambia gli elementi
  - Il parametro c è una coppia di elementi, il cui tipo è parametrico!

Metodi generici: esempio

```
public class UtilitàCoppia {

   public static <T> void scambiaElementi(Coppia<T> c) {
       T tmp;

       tmp = c.getPrimo();
       c.setPrimo(c.getSecondo());
       c.setSecondo(tmp);
   }
}
```

Generics e collezioni

- I Generics sono stati introdotti prevalentemente per poter avere un controllo sui tipi nelle collezioni
- Tutte le collezioni sono interfacce e classi generiche
- Iniziamo a vedere le interface **`Collection<E>`**, **`List<E>`** e la classe **`ArrayList<E>`**
- In particolare, rivediamo alcuni metodi e introduciamo altri concetti relativi ai Generics

Generics e collezioni

```
public interface Collection<E> {
    public void add(E x);
    public Iterator<E> iterator();
    …
}

public interface Iterator<E> {
    public E next();
    public boolean hasNext();
    public void remove();
}
```

Controllo sui tipi

- Rivediamo il codice con cui abbiamo iniziato la lezione

```
import java.util.*;
class Persona {
  private String nome;

  public Persona(String nome) {
    this.nome = nome;
  }

  public String toString() {
    return this.nome;
  }
}
public class MenoProblemiConCollezioniTest {
 …@Test…
}
```

Controllo sui tipi

```
public class MenoProblemiConCollezioniTest {
  @Test
  public void testCheCompila() {
     List<Persona> elencoPersone =
           new ArrayList<Persona>();
     Persona p1 = new Persona("Pippo");
     elencoPersone.add(p1);
     Persona p2 = new Persona("Pluto");
     elencoPersone.add(p2);
     Iterator<Persona> it = elencoPersone.iterator();
     while (it.hasNext()) {
        Persona persona = it.next();
        System.out.println(persona);
    }
  }
}
```

Non è più necessario il cast

Controllo sui tipi

- Proviamo a ripetere l'errore di inserire un oggetto di un tipo sbagliato:

```
import java.util.*;
class Persona { … }

public class MenoProblemiConCollezioniTest {
  @Test
  public void testCheNonCompila() {
     List<Persona> elencoPersone =
       new ArrayList<Persona>();
     Persona p1 = new Persona("Pippo");
     elencoPersone.add(p1);
     String p2 = new String("Pluto");
     elencoPersone.add(p2);
     Iterator<Persona> it = elencoPersone.iterator();
     while (it.hasNext()) {
        Persona persona = it.next();
        System.out.println(persona);
     }
  }
}
```

Questa volta abbiamo un errore a tempo di compilazione!

Iteratori e forma breve istruzione `for`

- Con l'introduzione dei Generics, grazie al maggior controllo sui tipi a tempo di compilazione, possiamo iterare su una lista con l'istruzione **`for`** con sintassi abbreviata
- Le istruzioni

```
Iterator it = elencoPersone.iterator();
while(it.hasNext())
  System.out.println(it.next())
```

- sono equivalenti a:

```
for (Persona p : elencoPersone)
  System.out.println(p);
```

Ricapitoliamo

- L'interface **`Collection<E>`** (e con essa tutte le sue estensioni e implementazioni) è generica
- Questo garantisce un controllo sui tipi degli oggetti che vengono aggiunti alla collezione a tempo di compilazione
- Non c'è più bisogno di fare il downcast quando prendiamo un oggetto dalla collezione
- Per scandire la lista possiamo usare l'istruzione **`for`** con sintassi abbreviata, senza ricorrere agli iteratori

Approfondimenti

- È possibile definire classi, interfacce e metodi generici con più parametri di tipo
  - Sintatticamente, si separano i vari parametri con una virgola

    `public class Esempio<T, S> {…}`

Approfondimenti: tipi primitivi

- Non è possibile istanziare i tipi di una classe, interface o metodo generico con un tipo primitivo (si devono quindi usare le classi wrapper)

```
public class TestCoppia {
   @Test
   public void testCheNonCompila() {
      Coppia<int> coppia;      // ERRORE: non compila
      int i1 = 100;
      int i2 = 200;
      coppia = new Coppia<int>(i1, i2); // ERRORE
   }
}
```

Approfondimenti: tipi primitivi

- Si devono quindi usare le classi wrapper

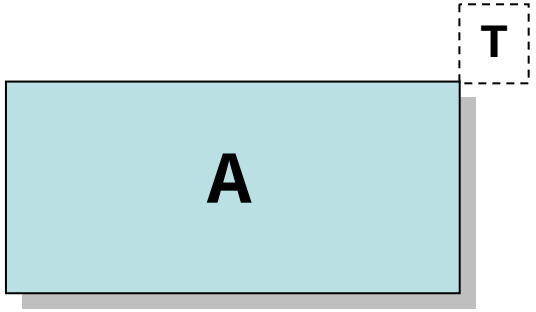
```
public class TestCoppia {
   @Test
   public void testCheCompila() {
      Coppia<Integer> coppia;       // OK
      Integer i1 = new Integer(100);
      Integer i2 = new Integer(200);
      coppia = new Coppia<Integer>(i1, i2); // OK
   }
}
```

Generics: convenzioni sui nomi

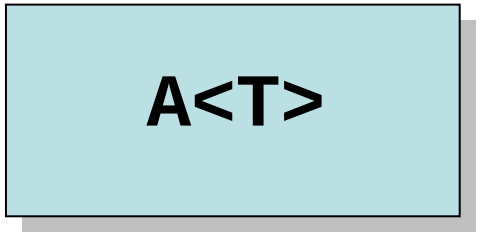
- <T>: Type (classe o interface)
- <S>: usato quando T è già in uso
- <E>: Elemento (molto usato nelle collezioni Java)
- <K>: chiave
- <V>: valore
- <N>: numero (una classe wrapper tra quelle usate per "oggettificare" i valori numerici)

Rappresentazione diagrammatica

- Rappresentazione diagrammatica



- oppure



Ancora sui Generics

- Approfondiamo altri concetti relativi ai generics analizzando metodi della interface **`List<E>`**
- Analizziamo il metodo **`addAll`**
  - Aggiunge alla collezione tutti gli elementi di un'altra collezione che viene passata come parametro
- Vediamone la segnatura:

```
boolean addAll(Collection<? extends E> c)
```

- Cosa significa? Di che tipo deve essere il parametro?

Generics: wildcard (o caratteri jolly)

`Collection<? extends E>`

- Significa: un oggetto **`Collection`** istanziato su **`E`** o su un qualsiasi sottotipo di **`E`**
- Esempio:

```
List<Strumento> strumenti;
List<Chitarra> chitarre;
…
strumenti.addAll(chitarre); // OK
```

Generics: wildcard (o caratteri jolly)

- Consideriamo ora la classe **`Collections`** (con la 's' finale): contiene un insieme di operazioni molto utili per manipolare collezioni
  - Es. metodi che implementano ordinamento, ricerca, calcolo dell'elemento max/min, inversione, ecc. ecc.
- Consideriamo il metodo **`reverse()`**
  - Inverte una collezione (il primo elemento diventa l'ultimo, il secondo il penultimo, ecc)
- Vediamone la segnatura:

```
static void reverse(List<?> list)
```

- Cosa significa? Di che tipo deve essere il parametro?

Generics: wildcard (o caratteri jolly)

`List<?>`

- Significa: un oggetto **`List`** istanziato su qualsiasi tipo (a cui non è necessario/utile associare un tipo formale con nome esplicito)
- Esempio:

```
List<Strumento> strumenti;
…
Collections.reverse(chitarre); // OK
```

Generics: wildcard (o caratteri jolly)

- Ancora dalla classe **`Collections`**, consideriamo il metodo **`fill()`**
  - Riempe gli elementi della lista che viene passata come primo parametro, con un oggetto passato come secondo parametro
- Vediamone la segnatura:

```
static <T> void fill(List<? super T> list, T obj)
```

- Cosa significa? Di che tipo deve essere il primo parametro?

Generics: wildcard (o caratteri jolly)

`List<? super T>`

- Significa: un oggetto **`List`** istanziato su **`T`** o su un qualsiasi supertipo di **`T`**
- Esempio:

```
List<Strumento> strumenti;
Chitarra fender;
…
Collections.fill(strumenti, fender); // OK
```

Più difficile…

- Ancora dalla classe **`Collections`**, consideriamo il metodo **`max()`**
  - Trova il massimo all'interno di una collezione, in base ad un criterio di ordinamento definito da un "comparatore"
- Vediamone la segnatura

```
public static <T> T max(
                Collection<? extends T> coll,
                Comparator<? super T> comp
            )
```

Più difficile…

```
public static <T> T max(
                Collection<? extends T> coll,
                Comparator<? super T> comp
              )

public interface Comparator<T> {
   public int compare(T o1, T o2)
}
```

- Cosa significa? Di che tipo devono essere i due parametri e perché?

Più difficile…

`Collection<? extends T>`

- Significa: un oggetto **`Collection`** istanziato su **`T`** o su un qualsiasi sottotipo di **`T`**

`Comparator<? super T>`

- Significa: un oggetto **`Comparator`** istanziato su **`T`** o su un qualsiasi supertipo di **`T`**
- Perché?

Esempio più difficile…

- Esempio dove T è **`Tamburo`**

```
List<Tamburo> tamburi;

Comparator<Strumento> comp =
        new ComparatoreDiDb<Strumento>();

Tamburo rumoroso =
        Collections.max(tamburi, comp); // OK
```

- Perché? Perché per ottenere il max di una collezione di un certo tipo basta un comparatore che sappia dettare il criterio di ordinamento per un suo qualsiasi supertipo

Ancora più difficile…

```
public static
    <T extends Object & Comparable<? super T>> T
    max( Collection<? extends T> coll )
```

- Returns the maximum element of the given collection, according to the natural ordering of its elements. All elements in the collection must implement the **`Comparable`** interface.

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

Riferimenti

- Se non vi siete ancora arresi e siete curiosi esistono articoli che spiegano i dettagli:

  http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

- Per sapere (quasi) tutto sui generics:

  http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html