Programmazione Parametrica

(a.k.a. Generics)

- **Programmazione parametrica: introduzione**
- **Generics e relationi di sottotipo**
  - wildcards
  - generics e vincoli
- **Implementazione di classi e metodi parametrici**
- **Supporto per i generics nella JVM**
- **Collections**

Programmazione *polimorfa*

- ***Polimorfo ~ multiforme, di molti tipi***
- ***Programmazione polimorfa:*** **creazione di costrutti (classi e metodi) che possono essere utilizzati in modo uniforme su dati di tipo diverso**
  - In Java, tradizionalmente ottenuta mediante i meccanismi di sottotipo ed ereditarietà
  - Da Java 1.5. anche mediante i meccanismi di parametrizzazione di tipo (a.k.a. generics)

Variabili di Tipo

- **Le variabili (o parametri) di tipo pemettono di creare astrazioni di tipo**
- **Classico caso di utilzzo nelle classi "*Container*"**

```
public class Set<E>
 {
   public ArrayList() { . . . }
   public void add(E element) { . . . }
   . . .
}
```

- **E = variabile di tipo**
  - astrae (e rappresenta) il tipo delle componenti

Continua

Variabili di Tipo

- Possono essere istanziate con tipi classe o interfaccia

```
ArrayList<BankAccount>
ArrayList<Measurable>
```

- Vincolo: tipi che istanziano variabili di tipo non possono essere primitivi (devono essere tipi riferimento)

```
ArrayList<double> // No!
```

- Classi wrapper utili allo scopo

```
ArrayList<Double>
```

Variabili di tipo e controlli di tipo

- **Utilizzare variabili di tipo nella programmazione permette maggiori controlli sulla correttezza dei tipi in fase di compilazione**
- **Aumenta quindi la solidità e robustezza del codice**

Continua

Variabili di tipo e controlli di tipo

- **Un classico caso di utilizzo di containers**

```
List intList = new LinkedList();
intList.add(new Integer(57));
Integer x = (Integer) intList.get(0);
```

- **Il cast è problematico, per vari motivi**
  - verboso, fonte di errori a run time
- **Ma necessario per la compilazione e per localizzare l'eventuale errore a run time**

Continua

Variabili di tipo e controlli di tipo

- **Container generici: più sintetici ed eleganti**

```
List<Integer> intList = new LinkedList<Integer>();
intList.add(new Integer(0));
Integer x = intList.get(0);
```

- **Compilatore può**
  - stabilire un invariante sugli elementi della lista
  - garantire l'assenza di errori a run-time in forza di quell'invariante.

Continua

Variabili di tipo e controlli di tipo

```
List<Integer> intList = new LinkedList<Integer>();
intList.add(new Integer(0));
Integer x = intList.get(0);
```

- Ora non è possibile aggiungere una stringa ad `intlist:List<Integer>`
- Le variabili di tipo rendono il codice parametrico più robusto e semplice da leggere e manutenere

Classi parametriche: uso

- **Usare un tipo parametrico = istanziarlo per creare riferimenti e oggetti**

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

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

Esempio: `Pair<T,S>`

- **Una semplice classe parametrica per rappresentare coppie di oggetti**

```
public class Pair<T, S>
{
   public Pair(T firstElement, S secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public T getFirst() { return first; }
   public S getSecond() { return second; }
   private T first;
   private S second;
}
```

Continua

Esempio: `Pair<T,S>`

- **Una semplice classe parametrica per rappresentare coppie di oggetti:**

```
Pair<String, BankAccount> result
      = new Pair<String, BankAccount>
            ("Harry Hacker", harrysChecking);
```

- **I metodi `getFirst` e `getSecond` restituiscono il primo e secondo elemento, con i tipi corrispondenti**

```
String name = result.getFirst();
BankAccount account = result.getSecond();
```

Variabili di tipo: convenzioni

Variabile	Significato Inteso
E	Tipo degli elementi in una collezione
K	Tipo delle chiavi in una mappa
V	Tipo dei valori in una mappa
T,S,U	Tipi generici

Esempio: LinkedList<E>

```
public class LinkedList<E>
{
   . . .
   public E removeFirst()
   {
      if (first == null) throw new NoSuchElementException();
      E element = first.data;
      first = first.next;
      return element;
   }
   . . .
   private Node first;

   private class Node
   {
       E data;
       Node next;
   }
}
```

Continua

Esempio: `LinkedList<E>`

- **Notiamo la struttura della classe `Node`**
  - se la classe è interna, come nell'esempio, non serve alcun accorgimento
  - all'interno di **`Node`** possiamo utilizzare il tipo **`E`**, il cui scope è tutta la classe
  - se invece la classe è esterna, dobbiamo renderla generica

Esempio: LinkedList<E>

```
class Node<F>
{
 F data;
 Node next;
}
public class LinkedList<E>
{
   . . .
   public E removeFirst()
   {
      if (first == null) throw new NoSuchElementException();
      E element = first.data;
      first = first.next;
      return element;
   }
   . . .
   private Node<E> first;
}
```

Continua

Generics e sottotipi

- I meccanismi di subtyping si estendono alle classi generiche

```
class C<T> implements / extends D<T> { . . . }
```

- `C<T>` <: `D<T>` per qualunque `T`
- Analogamente:

```
class C<T> implements / estends D { . . . }
```

- `C<T>` <: `D` per qualunque `T`
- Sembra tutto facile, MA . . .

Generics e sottotipi

- **Consideriamo**

```
List<Integer> li = new ArrayList<Integer>();
List<Number> ln = li;
```

- **La prima istruzione è legale, la seconda è più delicata …**
  - **`Number`** è una classe che ha **`Integer`** , **`Double`** e altre classi wrapper come sottotipi
  - Per capire se la seconda istruzione sia da accettare continuiamo con l'esempio …

Continua

Generics e sottotipi

```
List<Integer> li = new ArrayList<Integer>();
List<Object> lo = li;  // type error
lo.add("uh oh"));
Integer i = li.get(0); // uh oh ...
```

- **Problema**
  - nella terza istruzione inseriamo un **`Double`**
  - nella quarta estraiamo un **`Integer`** !
- **Errore è nella seconda istruzione**
  - soluzione: errore di compilazione per l'assegnamento

Continua

Generics e sottotipi

- **In generale:**

```
A ≤ B NON implica C<A> ≤ C<B>
```

- **Quindi, ad esempio:**

```
Set<Integer> NON è sottotipo di Set<Object>
```

- **Come dimostrato, vincoli necessari per la correttezza del principio di sostituibilità**

Generics e sottotipi

- **Limitazione sul subtyping con generics contro-intuitive**
  - uno degli aspetti più complessi dei generics
- **Spesso anche troppo restrittive**
  - illustriamo con un esempio

Continua

Generics e sottotipi

- **Stampa gli elementi di una qualunque collection**
- **Primo tentativo**

```
static void printCollection(Collection<Object> els)
{
   for (Object e:els) System.out.println(e);
   els.add("pippo"); // ok
}
```

  - Inutile: **`Collection<Object>`** non è il supertipo di **`Collection<T>`** per alcun **`T != Object`**

Continua

Wildcards

- **Stampa degli elementi di una collezione**
- **Secondo tentativo**

```
static void printCollection(Collection<?> els)
{
   for (Object e:els) System.out.println(e);
   els.add("pippo"); // ko
}
```

  - **`Collection<?>`** è supertipo di qualunque **`Collection<T>`**
  - Wildcard **`?`** indica un qualche tipo, non specificato

Continua

Wildcards

```
void printCollection(Collection<?> els)
{
   for (Object e:els) System.out.println(e);
}
```

- Possiamo estrarre gli elementi di `els` al tipo `Object`
- Corretto perché, qualunque sia il loro vero tipo, sicuramente è sottotipo di `Object`

Continua

Wildcards

- **Però …**

```
Collection<?> c = new ArrayList<String>();
c.add(new String()); // errore di compilazione!
```

- **Poichè non sappiamo esattamente quale tipo indica ?, non possiamo inserire elementi nella collezione**
- **In generale, non possiamo modificare valori che hanno tipo ?**

Continua

Domanda

- **Date un esempio di codice che causerebbe errore in esecuzione se permettessimo di aggiungere elementi a `Collection<?>`**

Risposta

```
Collection<Integer> ci = new ArrayList<Integer>;
Colletion<?> c = ci;
c.add(“a string”); // non compila
ci.get(0).intValue();
```

- L’ultima istruzione invocherebbe `intValue()` sul primo elemento di `ci`
- ma quell’elemento ha tipo `String` …
- Il compilatore previene l’errore, rigettando la `add()`

Wilcards con vincoli (*bounded*)

- **Shapes: (again!)**

```
interface Shape
{
   public void draw(Graphics g);
}

class Circle extends Shape
{
   private int x, y, radius;
   public void draw(Graphics g) { ... }
}

class Rectangle extends Shape
{
   private int x, y, width, height;
   public void draw(Graphics g) { ... }
}
```

Wilcards con vincoli (*bounded*)

- **`Graphics` e il metodo `draw()`**

```
public class Graphics
{
   // disegna una shape
   public void draw(Shape s) { s.draw(this); }

   // disegna tutte le shapes di una lista
   public static void drawAll(List<Shape> shapes) {
       for (Shape s:shapes) s.draw(this)
   }
   . . .
}
```

- **Solito problema: `drawAll()` non può essere invocato su una `List<Circle>`**

Continua

Bounded Wilcards

- **Quello che ci serve è un metodo che accetti liste di qualunque (sotto) tipo di `Shape`**

```
void drawAll(List<? extends Shape> shapes) { ... }
```

- **`List<? extends Shape>`**
  - bounded wildcard
  - indica un tipo sconosciuto, sottotipo di **`Shape`**
  - il bound può essere qualunque tipo riferimento (classe o interfaccia)
- **Ora il metodo ha la flessibilità necessaria e desiderata**

Continua

Bounded Wilcards

- **`Graphics` e il metodo `draw()`**

```
public class Graphics
{
   // disegna una shape
   public void draw(Shape s) { s.draw(this); }

   // disegna tutte le shapes di una lista
   public void drawAll(List<? extends Shape> shapes)
   {
       for (Shape s:shapes) s.draw(this)
   }
   . . .
}
```

Continua

Bounded Wilcards

- Attenzione: c’è sempre un prezzo da pagare

```
void addRectangle(List<? extends Shape> shapes) {
    // errore di compilazione
    shapes.add(new Rectangle());
 }
```

- Non possiamo modificare strutture con questi tipi [perché?]