

Linguaggi

*Corso M-Z - Laurea in Ingegneria Informatica
A.A. 2009-2010*

Alessandro Longheu

<http://www.diit.unict.it/users/alongheu>

alessandro.longheu@diit.unict.it

Generics in Java



Cosa sono i Generics?

- Un generic è uno strumento che permette la definizione di un tipo parametrizzato, che viene successivamente esplicitato in fase di compilazione secondo le necessità (simile ai template C++).
- Svantaggi:
 - Invece di: `List words = new ArrayList();`
 - Si definisce: `List<String> words = new ArrayList<String>();`
- Vantaggi:
 - Fornisce una migliore gestione del type checking durante la compilazione
 - Evita il casting da Object. I.e., invece di:
`String title = ((String) words.get(i)).toUpperCase();`
 - utilizzeremo *`String title = words.get(i).toUpperCase();`*



Cosa sono i Generics?

- Per permettere una maggiore genericità si potrebbe rilassare il controllo sui tipi, ma questo presta il fianco ad errori...
 - È sbagliato abolire il controllo di tipo!
 - Occorre *un altro modo* per esprimere genericità, che consenta un controllo di tipo a compile time, in modo da garantire la type safety: *"se viene compilato, è certamente corretto"*
- Java 1.5 introduce i tipi parametrici ("generici")
 - Il tipo può essere un parametro:
 - in attributi e metodi di classi, purché non statici; la notazione è NomeClasse<TIPO> se serve indicare un oggetto di classe NomeClasse, o semplicemente TIPO nei punti dove c'era Object
 - Si possono definire relazioni fra "tipi generici", quindi si recupera il "lato buono" dell'ereditarietà, inquadrandolo in un contesto solido.



Una classe parametrica

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



Uso della classe parametrica

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



Generics dalla 1.5

- Tutta la JFC è stata riscritta dalla versione 1.5 per far uso dei generici
 - Anche classi preesistenti (come Vector) sono state reingegnerizzate e riscritte in accordo al nuovo idioma
 - Le operazioni sulla JFC "generica" si dicono *checked (controllate)* o *type-safe (sicure come tipo)*



Uso dei Generics

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator(); }
public interface Iterator<E> {
    E next();
    boolean hasNext(); }
```

```
...
List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); // 2'
Integer x = myIntList.iterator().next(); // 3'
```

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer)myIntList.iterator().next(); // 3
```

- Tipi parametrizzati, perchè tutte le occorrenze del parametro “formal type” (E in questo caso) sono sostituite dagli argomenti, “actual type” (in questo caso Integer).



Comportamento dei Generics

- Il bytecode dello stesso codice con e senza generics è identico:
List
List<String>
<List<List<String>>
- sono identici in bytecode e corrispondono alla plain old List.
- L'**Erase** è il processo che converte il programma scritto con i generics nella forma senza generics che rispecchia più da vicino il bytecode prodotto. Il termine *Erase* non è proprio corretto in quanto vengono rimossi i generics, ma aggiunti i cast.
- L'aggiunta dei cast è implicita, e java fornisce la **Cast-iron guarantee**: il cast implicito aggiunto dalla compilazione dei generics NON DEVE MAI FALLIRE. Questa regola si applica solo per il codice che non presenta *unchecked warnings*.
- I vantaggi dell'implementazione via Erasure sono:
 - mantiene le cose semplici (non c'è nulla di nuovo)
 - mantiene le cose piccole (una sola implementazione di List)
 - semplifica l'evoluzione (la stessa libreria può essere acceduta da codice *generico* e da codice legacy)



Ereditarietà dei tipi generici

- In generale, se C2 è una classe derivata da C1 e G è una dichiarazione generica, non è vero che G<C2> è un tipo derivato da G<C1>.

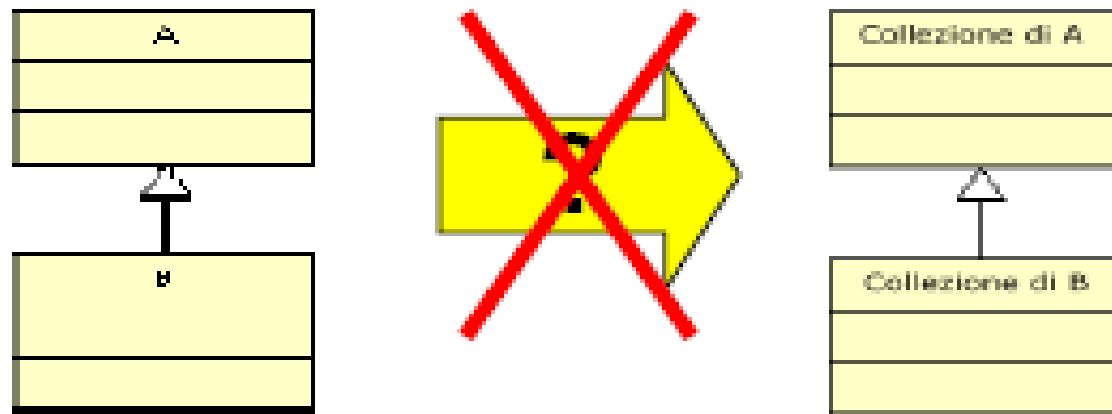
List<String> ls = new ArrayList<String>(); //1

List<Object> lo = ls; //2

- L'istruzione 2 genera un errore in compilazione

Ereditarietà dei tipi generici

- Se B deriva da A, NON si può dire che una collezione di elementi di B derivi dalla collezione di elementi di A, perché in generale ciò non ha senso (operazioni impossibili)



- ALCUNE operazioni potrebbero anche essere sicure (negli array, la lettura), ma ciò non è vero in generale.



Ereditarietà dei tipi generici

- Consideriamo la classe generica `LinkedList <T>`:
- prendiamo due sue "istanziazioni"
 - `LinkedList <Number>`
 - `LinkedList<Integer>`
- Sono due tipi diversi, incompatibili fra loro, anche se `Integer` estende `Number`; situazione in contrasto a quella che si verifica negli array, dove `Integer[]` è un sottotipo di `Number[]`.
- Per verificarlo, creiamo due liste:
 - `LinkedList<Number> l1 = new LinkedList<Number>();`
 - `LinkedList<Integer> l2 = new LinkedList<Integer>();`
- ...e consideriamo i due possibili assegnamenti `l1 = l2` e `l2 = l1`; si ottiene in ogni caso errore perché...
- `LinkedList<Integer>` non estende `LinkedList<Number>`



Ereditarietà dei tipi generici

- Gli assegnamenti in precedenza sono impossibili perché viene violato il (noto) principio di sostituzione: Ad una variabile di un dato tipo può essere assegnato un valore di ogni sottotipo e un metodo con un argomento di un dato tipo può essere chiamato con un argomento di ogni sottotipo.

```
List<Number> numbers = new ArrayList<Number>();
numbers.add(2);
numbers.add(3.14d);
assert numbers.toString().equals("[2, 3.14]");
```

- Qui il principio vale fra List e ArrayList e fra Number e Integer e Double rispettivamente.
- List<Integer> invece NON E' UN SOTTOTIPO di List<Number> in quanto viene violato il principio di sostituzione, ad esempio:

```
List<Integer> integers = Arrays.asList(1, 2);
List<Number> numbers = integers; // non compila!
numbers.add(3.14d);
assert integers.toString().equals("[1, 2,3.14]");
```



Tipi parametrici varianti

- L'esperimento precedente ha mostrato che non ha senso cercare una compatibilità generale fra tipi parametrici, perché non può esistere.
- Ha senso invece cercare compatibilità fra casi specifici e precisamente fra tipi di parametri di singoli metodi.
- Perciò, alla normale notazione dei tipi generici `List<T>`, usata per creare oggetti, si affianca una nuova notazione, pensata esplicitamente per esprimere i tipi accettabili come parametri in singoli metodi
- Si parla quindi di tipi parametrici varianti, in Java più brevemente detti WILDCARD.



Tipi parametrici varianti

- la notazione `List<T>` denota il normale tipo generico
- il tipo **covariante** `List<? extends T>` fattorizza le proprietà dei `List<X>` in cui `X` estende `T`
- il tipo **controvariante** `List<? super T>` fattorizza le proprietà dei `List<X>` in cui `X` è esteso da `T`
- il tipo **bivariante** `List<?>` fattorizza tutti i `List<T>` senza distinzione (equivale a scrivere `List<? extends Object>`)



Tipi parametrici varianti

- Vale il cosiddetto Get and Put PRINCIPLE: si usa extends quando si devono solo estrarre valori da una struttura
- Relativamente al tipo controvariante, poiché la classe supertipo potrebbe essere qualsiasi, fino ad Object, i metodi a cui si potrebbe ricorrere entro il codice che contiene il tipo controvariante non possono essere i metodi di T, ma al massimo quelli di Object, sicuramente supportati, quindi questa wildcard non permette di operare efficacemente e si usa quando si devono solo inserire valori in una struttura (senza farci null'altro); diviene necessario nei casi in cui si debbano copiare elementi da una collezione ad un'altra di un suo supertipo, che si ipotizza accessibile come parametro del metodo.
- non si usano wildcard quando si devono sia estrarre che₁₅ inserire valori nella stessa struttura.



Tipi parametrici varianti

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    while (i.hasNext()){  
        //for (k = 0; k < c.size(); k++) {  
            System.out.println(i.next());  
        }  
    }  
}
```

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

Wildcards

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




Tipi parametrici varianti

```
public class MyList<T> {  
    private T head;  
    private MyList<T> tail;  
    public T getHead(){ return head; }  
    public <E extends T> void setHead(E element){  
        head=element; }  
}
```

```
MyList<Number> list1 = new MyList<Number>();  
MyList<Integer> list2 = new MyList<Integer>();  
list1.setHead( new Double(1.4) ); // OK!  
list1.setHead( list2.getHead() ); // OK!
```



Tipi parametrici varianti

```
public class MyList<T> {  
    private T head;  
    private MyList<T> tail;  
    public T getHead(){ return head; }  
    public <E extends T> void setHead(E element){ head=element; }  
    public void setTail(MyList<T> l){ tail=l; }  
    public MyList<? extends T> getTail(){ return tail; }  
}
```

- Restituisce una lista di elementi di tipo T o più specifico di T

MyList<? extends Number> list3 = list1.getTail();

MyList<? extends Number> list4 = list2.getTail();

MyList<? extends Integer> list5 = list2.getTail();

*I primi due restituiscono una lista di Number, compatibile col tipo
"lista di qualcosa che estenda Number"*

*Il terzo restituisce una lista di Integer, compatibile col tipo "lista di
qualcosa che estenda Integer"*



Tipi parametrici varianti

```
public class MyList<T> {  
    private T head;  
    private MyList<? extends T> tail;  
    public T getHead(){ return head; }  
    public <E extends T> void setHead(E element){...}  
    public MyList<? extends T> getTail(){ return tail; }  
    public void setTail(MyList<? extends T> l){ tail=l;}  
}
```

- Non c'è realmente bisogno che la coda sia una lista di T!
- Possiamo essere più generici!
- Conseguentemente, possiamo rilassare il vincolo su setTail, il cui argomento ora può essere una lista di qualunque cosa estenda T
- `list1.setTail(list2); // SI', ORA VA BENE!`
- Si rendono così SELETTIVAMENTE possibili TUTTE e SOLE le operazioni "sensate" e significative!



Tipi parametrici varianti

- Nell'esempio precedente abbiamo usato:
- il tipo generico `MyList<T>` *per creare oggetti*
 - `MyList<Number>`, `MyList<Integer>`, ...
- tipi covarianti come `MyList<? extends Number>`
 - fattorizza le proprietà dei tipi di liste che estendono `Number`, come `MyList<Integer>`, `MyList<Double>`, o `MyList<Number>` stessa
- NON abbiamo invece usato:
 - tipi controvarianti come `MyList<? super Number>`
 - fattorizzerebbe le proprietà di tutte le liste di tipi più generici di `Number`, come ad esempio `MyList<Object>`
 - il tipo bivariante `MyList<?>`



Tipi parametrici varianti

```
public abstract class Shape {  
    public abstract void draw(Canvas c);}
```

```
public class Circle extends Shape {  
    private int x, y, radius;  
    public void draw(Canvas c) { ... }}
```

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

```
public void drawAll(List<Shape> shapes) {  
    for (Shape s: shapes) { s.draw(this); }}
```

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



Tipi parametrici varianti

- $\text{List}<? \textbf{extends Shape}>$ è un esempio di *bounded wildcard*.
- *Il simbolo?* Sta per un tipo sconosciuto
- Sappiamo che in questo caso tale tipo sconosciuto è un subtype di Shape.
- Diremo che Shape è un *upper bound di una wildcard*.



Tipi parametrici varianti

- Supponiamo di voler scrivere un metodo che prende un array di objects e pone gli oggetti dell'array in una collection.
- Soluzione:

```
static void fromArrayToCollection  
(Object[] a, Collection<?> c)  
{ for (Object o : a) { c.add(o); // compile time  
error  
}}
```



Tipi parametrici varianti

- I metodi generici consentono di superare un tale problema.
- Così come in una dichiarazione di tipo, la dichiarazione di un metodo può essere generica cioè parametrizzata rispetto ad uno o più parametri

```
static <T> void fromArrayToCollection  
    (T[] a, Collection<T> c)  
    { for (T o : a) { c.add(o); // correct}  
    }
```




Argomenti avanzati su Generics

- Siti in cui è possibile trovare argomenti avanzati sui generics:
- http://home.dei.polimi.it/ghezzi/PRIVATE/Generics_Paper.pdf
- http://www.jugpadova.it/files/JUGPD41_Zoleo_Generics.pdf