Verso i tipi generici in Java: un esempio

```
interface ListOfNumbers {
  boolean add(Number elt);
  Number get(int index);
}
interface ListOfIntegers {
  boolean add(Integer elt);
  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>>  …
```

2

Parametri e parametri di tipo


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

- Dichiarazione del parametro formale
- Istanziato con espressione che ha il tipo richiesto (es: lst.add(7))
- Tipo di add: Integer → boolean

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

- Dichiarazione di un parametro di tipo
- Istanziare con un qualunque tipo
  - List<String>
- "Il tipo" di List: Type → Type

3

Variabili di tipo

Dichiarazione

```
class NewSet<T> implements Set<T> {
  // rep invariant:
  //   non-null, contains no duplicates
  // …
  List<T> theRep;
  T lastItemInserted;
  …
}
```

Utilizzo

4

Dichiarare e istanziare classi generiche


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

- Convenzioni standard per nomi i variabili di tipo:
  `T` per `Type`, `E` per `Element`,
  `K` per `Key`, `V` per `Value`, …

Istanziare una classe generica significa fornire un valore di tipo:

```
Name<Type1, …, TypeN>
```

5

Un esempio: l'interfaccia generica `Comparable<T>`


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

- Definisce un ordinamento totale (l' *"ordinamento naturale"*) sugli oggetti di ogni classe che implementa l'interfaccia
  - `x.compareTo(y) < 0` se `x` precede `y` nell'ordinamento
  - `x.compareTo(y) > 0` se `x` segue `y` nell'ordinamento
  - `x.compareTo(y) = 0` se `x` e `y` sono equivalenti nell'ordinamento
- Spesso si vuole che `compareTo()` sia consistente con `equals()`:
  `x.compareTo(y) == 0` se e solo se `x.equals(y) == true`
- Viene usato da algoritmi di ordinamento (es: `Arrays.sort()`, `Collections.sort()`)
- È implementata da numerose classi Java come `String`, `Number`, …

6

Vincoli di tipo

- Possiamo limitare i tipi sostituibili ad un parametro di tipo usando un **vincolo di tipo**. invece di un semplice identificatore
- Il caso più semplice è **extends**:

```
interface NumBag<E extends Number> {…}

NumBag<Number> // OK
NumBag<Integer> // OK, Integer e' sottotipo di Number
NumBag<String> // compile-time error, String non è
               // sottotipo di Number
```

- I metodi dell'interfaccia `NumBag` possono assumere che gli oggetti di tipo `E` rispondano a tutti i metodi di `Number`
- Possiamo leggere p.es. `List<E>` come `List<E extends Object>`

7

Vincoli di tipo "extends"

`<TypeVar extends SuperType>`

- *upper bound*; parametro di tipo attuale deve essere sottotipo di **`SuperType`**

`<TypeVar extends ClassA & InterfaceB & InterfaceC & …>`

- *Multiple* upper bounds: max una classe + arbitrarie interfacce

Esempio:

```
// strutture di ordine su alberi
public class TreeSet<T extends Comparable<T>> {…}
```

- I metodi di `TreeSet` possono assumere che gli oggetti di `T` sono confrontabili con il metodo **`compareTo()`** dell'interfaccia **`Comparable`**

8

Metodi generici

- Anche i metodi possono avere parametri di tipo
- Nel corpo possono usare come normali tipi sia gli eventuali parametri di tipo formali della classe (se è generica), sia i propri
- Le chiamate di metodo generici devono in generale istanziare i parametri di tipo
  - questo si può evitare se il compilatore è in grado di inferire i parametri di tipo staticamente

9

Esempio: verso metodi generici


```
class Utils {
  static double sumList(List<Number> lst) {
      double result = 0.0;
      for (Number n : lst) {  //nota il "for esteso"
        result += n.doubleValue();
      }
      return result;
  }
  static Number choose(List<Number> lst) {
      int i = … // numero random < lst.size
      return lst.get(i);
  }
}
```

- **NB:** `sumList` e `choose` non possono essere invocati passando un parametro di tipo **`List<Integer>`** o **`List<Double>`** o … (vedremo dopo perché!) dovremmo ridefinire i metodi…

10

Soluzione con metodi generici

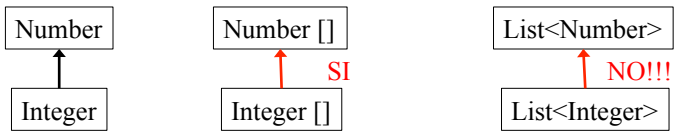
```
class Utils {
  static <T extends Number>
  double sumList(List<T> lst) {
      double result = 0.0;
      for (T n : lst) { // T also works
        result += n.doubleValue();
      }
      return result;
  }

  static <T> T choose(List<T> lst) {
      int i = … // random number < lst.size
      return lst.get(i);
  }
}
```

Dichiaro un parametro di tipo con vincolo

Si notino i diversi significati di **T**:
Parametro formale
Tipo del risultato
Parametro attuale

Tipi generici, gerarchia dei tipi in Java e nozione di sottotipo



In Java, diciamo che il tipo **A** è "compatibile (per assegnamento)" con il tipo **B** se posso assegnare un oggetto di tipo **A** a una variabile di tipo **B**

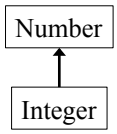
- Una forma di "principio di sostituzione", ristretta all'assegnamento
- Comprende ereditarietà tra classi o interfacce, e "implements"

- **`Integer`** è compatibile per assegnamento con **`Number`**
- Quindi **`Integer[]`** è compatibile per assegnamento con **`Number[]`**
- Ma **`List<Integer>`** **non è** compatibile per assegnamento con **`List<Number>`**!!!
- La nozione di compatibilità di Java è *invariante* per le classi generiche

12

Principio di sostituzione:
`List<Number>` e `List<Integer>`


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



type `List<Number>` has:

```
boolean add(Number elt);
Number get(int index);
```

type `List<Integer>` has:

```
boolean add(Integer elt);
Integer get(int index);
```

Vediamo che `List<Number>` non è un supertipo (né sottotipo) di `List<Integer>`, perché il principio di sostituzione non vale:

- `List<Number> x = ...; List<Integer> y = ...;`
- `x = y; x.add(new Number(4)); // errore!`
- `y = x; Integer z = y.get(0);  // errore!`

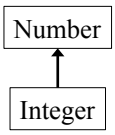
13

Principio di sostituzione:
`Number[]` e `Integer[]`

Esattamente lo stesso problema si ha con gli array: non vale il principio di sostituzione, ma **Integer[]** è compatibile per assegnamento con **Number[]**

- `Number[] x = ...; Integer[] y = ...;`
  `// Compila, ma lancia una java.lang.ArrayStoreException`
- `x = y; x[0] = new Number(4));`
  `// Non compila`
- `y = x; Integer z = y[0];  // errore!`

Quindi la gerarchia dei tipi Java (compatibilità per assegnamento) non rispetta sempre il principio di sostituzione!



- Java (ma anche C#) hanno fatto questa scelta prima dell'introduzione dei generici.
- Cambiarla ora è un po troppo invasivo per i pigri programmatori Java (commento obbligato per chi fa ricerca sui principin dei linguaggi di programmazione)

14

Un caso particolare: sottotipi **covarianti**


```
interface List<T> {
  T get(int index);
}
```

type `List<Number>`:

```
Number get(int index);
```

type `List<Integer>`:

```
Integer get(int index);
```

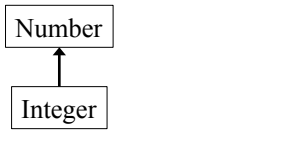
La nozione di sottotipo ***covariante*** sarebbe corretta, perché si accede solo in lettura (ma non è realizzata in Java):

- `List<Integer>` sottotipo di `List<Number>`

"Covariante": la relazione di sottotipo ha lo stesso verso di quella tra i parametri di tipo

15

Un caso particolare: sottotipi **contravarianti**




```
interface List<T> {
  boolean add(T elt);
}
```

type `List<Number>`:

```
boolean add(Number elt);
```

type `List<Integer>`:

```
boolean add(Integer elt);
```

La nozione di sottotipo *contravariante* sarebbe corretta, perché si accede solo in scrittura(ma non è realizzata in Java):

- `List<Number>` è sottotipo di `List<Integer>`

"Contravarianza": la relazione di sottotipo ha verso contrario a quella tra i parametri di tipo

16

Compatibilità tra classi generiche con stesso parametro di tipo

- La nozione di sottotipo sui generici, tra istanze con gli stessi parametri, funziona come uno se lo aspetterebbe anche in Java
- Esempio: Assumiamo che **`LargeBag`** extends **`Bag`**, allora
  - **`LargeBag<Integer>`** è un sottotipo di **`Bag<Integer>`**
  - **`LargeBag<Number>`** è un sottotipo di **`Bag<Number>`**
  - **`LargeBag<String>`** è un sottotipo di **`Bag<String>`**
  - …

Un esempio: `addAll`


```
interface Set<E> {
  // Aggiunge a this tutti gli elementi di c
  // (che non appartengono a this)
  void addAll(??? c);
}
```

Qual è il miglior tipo per il parametro formale?

- Il più ampio possibile…
- … che permette di avere implementazioni corrette

addAll


```
interface Set<E> {
  // Aggiunge a this tutti gli elementi di c
  // (che non appartengono a this)
  void addAll(??? c);
}
```

Una prima scelta: `void addAll(Set<E> c);`

Troppo restrittivo:

- Un parametro attuale di tipo **`List<E>`** non sarebbe permesso (perchè **List<E>** non estende **Set<E>**. Spiacevole///

addAll


```
interface Set<E> {
  // Aggiunge a this tutti gli elementi di c
  // (che non appartengono a this)
  void addAll(??? c);
}
```

Secondo Tentativo: `void addAll(Collection<E> c);`

Meglio, perché **Collection<E>** è il tipo di collezione più generlate
Ma ancora troppo restrittivo:

- Se istanzio l'interfaccia come **`Set<Number>`**, un parametro attuale di tipo **`List<Integer>`** non va bene anche se **`addAll`** ha solo bisogno di leggere da **c** non di modificarlo!!!|
- Questa è la principale limitazione della nozione di invarianza per i generici in Java

addAll

```
interface Set<E> {
  // Aggiunge a this tutti gli elementi di c
  // (che non appartengono a this)
  void addAll(??? c);
}
```

Proviamo ancora:

```
<T extends E> void addAll(Collection<T> c);
```

Idea buona: un parametro generico ma vincolato

- Posso avere un parametro attuale di tipo `List<Integer>` per `Set<Number>`
- `addAll` non puo' vedere nell'implementazione il tipo `T` sa solo che è un sottotipo di `E`. Non può modificare la collection `c`

Altro esempio

```
<T> void copyTo(List<T> dst, List<T> src) {
    for (T t : src)
      dst.add(t);
}
```

Va bene ma questo è meglio, perché consente di avere liste di tipo diverso (ma compatibile) come argomenti di **copyTo**:

```
<T1, T2 extends T1> void copyTo(List<T1> dst,
                                List<T2> src) {
    for (T2 t : src)
      dst.add(t);
}
```

Wildcards

wildcard = una variabile di tipo anonima

- **?** : Tipo non conosciuto
- Si usano le wildcard quando si usa un tipo esattamente una volta ma non si conosce il nome
- L'unica cosa che si conosce è l'unicità del tipo

Sintassi delle wildcards:

- **? extends Type**, sottotipo non specificato del tipo **Type**
- **?**, notazione semplificata per **? extends Object**
- **? super Type**, supertipo non specificato del tipo **Type**

23

Esempi


```
interface Set<E> {
  void addAll(Collection<? extends E> c);
}
```

- Più flessibile di
  `void addAll(Collection<E> c);`
- Espressiva come
  `<T extends E> void addAll(Collection<T> c);`

24

PECS:
Producer Extends, Consumer Super

Quando si usano le wildcards?

- Usare `? extends T` nei casi in cui si vogliono ottenere dei valori (da un produttore di valori)
- Usare `? super T` nei casi in cui si vogliono inserire valori (in un consumatore)
- Non usare (basta `T`) quando si ottengono e si producono valori

```
<T> void copy(List<? super T> dst,
              List<? extends T> src){
    for (T t : src) dst.add(t);
}
```

25

? vs Object

`?` è un tipo particolare anonimo

```
void printAll(List<?> lst) {…}
```

Qual è la differenza tra `List<?>` e `List<Object>`:

- Possiamo istanziare `?` Con un tipo qualunque: `Object`, `String`, …
- `List<Object>` è più restrittivo: `List<String>` non va bene.
- Ogni istanza di `List<E>` è compatibile per assegnamento con `List<?>`, non con `List<Object>`

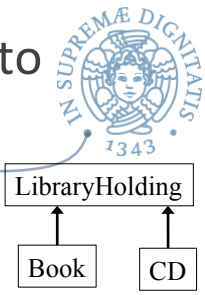
Qual è la differenza tra `List<Foo>` e `List<? extends Foo>`

- Nel secondo caso il tipo anonimo è **un** sottotipo sconosciuto del tipo `Foo`

  `List<? extends Animal>` puo' memorizzare `Giraffe` ma non `Zebre`

26

Compatibilità per assegnamento tra array. A volte funziona…

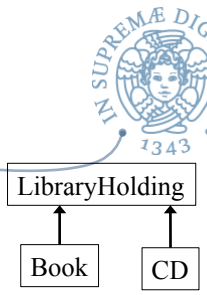


Gli inglesi dicono: “Programmers do okay stuff”

```
void maybeSwap(LibraryHolding[] arr) {
    if(arr[17].dueDate() < arr[34].dueDate())
        // … swap arr[17] and arr[34]
}

// cliente
Book[] books = …;
maybeSwap(books); // usa la covarianza degli array
```


Ma può andare male




```
void replace17(LibraryHolding[] arr,
               LibraryHolding h) {
  arr[17] = h;
}

// il solito cliente
Book[] books = …;
LibraryHolding theWall = new CD("Pink Floyd",
                                "The Wall", …);
replace17(books, theWall);
Book b = books[17]; // contiene un CD
b.getChapters(); // problema!!
```

Le scelte di Java

- Tipo dinamico è un sottotipo di quello statico
  - Violato nel caso di `Book b`
- La scelta di Java:
  - Ogni array "conosce" il suo tipo dinamico (`Book []`)
  - Modificare a (run-time) con un un supertipo determina `ArrayStoreException`
- Pertanto `replace17` solleva una eccezione
  - *Every Java array-update includes run-time check*
    - ✓ (dalla specifica della JVM)
  - Morale: fate attenzione agli array in Java

I generici a run-time… non esistono: type erasure

Tutti i tipi generici sono trasformati in `Object` nel processo di compilazione

- Motivo: backward compatibility con il codice vecchio
- Morale, a run-time, tutte le istanziazioni generiche hanno lo stesso tipo

```
List<String>  lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass()  // true
```

Generici e casting

```
List<?> lg = new ArrayList<String>();  // ok
List<String> ls = (List<String>) lg;   // warning
```

Dalla documentazione Java "Compiler gives an unchecked warning, since this is something the runtime system *will not check for you*"

Problema:

```
public static <T> T badCast(T t, Object o)
{
   return (T) o;    // unchecked warning
}
```

31

equals

Erasure: tipo dell'argomento non esiste a runtime

```
class Node<E> {
  …
  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof Node<E>))  {
      return false;
    }
    Node<E> n = (Node<E>) obj;
    return this.data().equals(n.data());
  }
  …
}
```

32

Equals

```
class Node<E> {
  …
  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof Node<?>))  {
      return false;
    }
    Node<E> n = (Node<E>) obj;
    return this.data().equals(n.data());
  }
  …
}
```

Erasure: A run time, non si sa cosa e' E

Tips (da stackoverflow.com)

- Start by writing a concrete instantiation
  - Get it correct (testing, reasoning, etc.)
  - Consider writing a second concrete version
- Generalize it by adding type parameters
  - Think about which types are the same or different
  - The compiler will help you find errors

Compilazione di Java Generics (JG)

- Il compilatore verifica l'utilizzo corretto dei generici
- I parametri di tipo sono eliminati nel processo di compilazione e il "class file" risultante dalla compilazione è un normale class file senza poliformismo parametrico.

Esempio


```
class Vector<T> {
  T[] v; int sz;
  Vector() {
    v = new T[15];
    sz = 0;
  }
  <U implements Comparer<T>>
  void sort(U c) {
    …
    c.compare(v[i], v[j]);
    …
  }
}
…
Vector<Button> v;
v.addElement(new Button());
Button b = v.elementAt(0);
```

```
class Vector {
  Object[] v; int sz;
  Vector() {
    v = new Object[15];
    sz = 0;
  }

  void sort(Comparer c) {
    …
    c.compare(v[i], v[j]);
    …
  }
}
…
Vector v;
v.addElement(new Button());
Button b =
    (Button)v.elementAt(0);
```

Considerazioni

- JG aiutano a migliorare il polimorfismo della soluzione
- Limitazione principale: il tipo effettivo è perso a runtime a causa del type erasure
- Tutte le istanziazioni sono identificate
- Esistono altre implementazioni dei generici per Java

Generics e Java

System / Feature	JG/Pizza Bracha, Odersky, Stoutamire, Wadler	NextGen Cartwright, Steele	PolyJ Bank, Liskov, Myers	Agesen, Freund, Mitchell	Generic CLR Kennedy, Syme
Parameterized types	✓ + bounds	✓ + bounds	✓ + constraints	✓ + bounds	✓ + bounds
Polymorphic methods	✓	✓	×	×	✓
Type checking at point of definition	✓	✓	✓	×	✓
Non-reference instantiations	×	×	✓	✓	✓
Exact run-time types	×	✓	?	✓	✓
Polymorphic virtual methods	×	✓	×	×	✓
Type parameter variance	×	✓	×	×	×