



JAVA GENERICS

1



Java Interface e astrazione

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

... e ListOfStrings

```
// Necessario astrarre sui tipi  
interface List<E> {  
    boolean add(E n);  
    E get(int index);  
}
```

Usiamo i Tipi!!!

```
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 una espressione che ha il tipo richiesto (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 e’ 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:
T per **T**ype, E per **E**lement,
K per **K**ey, V per **V**alue, ...

Istanziare una classe generica significa fornire un valore di tipo:

```
Name<Type1, ..., TypeN>
```

5

Istanziano tipi



```
boolean add1(Object elt);
boolean add2(Number elt);
add1(new Date()); // OK
add2(new Date()); // compile-time error
```

Limite superiore
gerarchia

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

List1<Date> // OK, Date e' un sottotipo di Object
List2<Date> // compile-time error, Date non è
            // sottotipo di Number
```

6

Visione effettiva dei generici



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

- (analogo per le interface)
- (intuizione **Object** è il limite superiore di default nella gerarchia dei tipi)

Istanziatura identica:

```
Name<Type1, ..., TypeN>
```

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

7

Usiamo le variabili di tipo



Si possono effettuare tutte le operazioni compatibili con il limite superiore della gerarchia

- Concettualmente questo corrisponde a forzare una precondizione sulla istanziazione del tipo

```
interface List1<E extends Object> {  
    void m(E arg) {  
        arg.asInt(); // compiler error, E potrebbe  
                    // non avere l'operazione asInt  
    }  
}  
  
interface List2<E extends Number> {  
    void m(E arg) {  
        arg.asInt(); // OK, Number e tutti i suoi  
                    // sottotipo supportano asInt  
    }  
}
```

8

Vincoli di tipo



<TypeVar extends SuperType>

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

<TypeVar extends ClassA & InterfaceB & InterfaceC & ...>

- *Multiple upper bounds*

<TypeVar super SubType>

- *lower bound*; va bene il subtype o uno qualunque dei suoi supertipi

Esempio:

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

9

Esempio



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

10

Soluzione efficace



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

Dichiarare i
Vincoli sui tipi

Dichiarare i vincoli sui
tipi

11

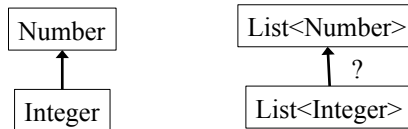
Metodi generici



- ☞ Metodi possono usare i tipi generici delle classi
- ☞ Possono dichiarare anche i loro tipi generici
- ☞ Le invocazioni di metodo generici devono obbligatoriamente istanziare i parametri di tipo
 - Staticamente: Una forma di inferenza di tipo

12

Generici e la nozione di sottotipo



- **Integer** e' un sottotipo di **Number**
- **List<Integer>** e' un sottotipo di **List<Number>**?

13

Quali sono le regole di Java?



Se **Type2** e **Type3** sono differenti, e **Type2** e' un sottotipo di **Type3** allora **Type1<Type2>** *non* è un sottotipo di **Type1<Type3>**

Formalmente: la nozione di sottotipo usata in Java è *invariante* per le classi generiche

14

Esempi (da Java)



Tipi e sottotipi

- Integer e' un sottotipo di Number
- ArrayList<E> e' un sottotipo di List<E>
- List<E> e' un sottotipo di Collection<E>

Ma

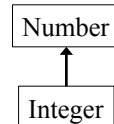
- List<Integer> non e' un sottotipo di List<Number>

15

List<Number> e List<Integer>



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



```
type List<Number>:  
    boolean add(Number elt);  
    Number get(int index);
```

```
type List<Integer> has:  
    boolean add(Integer elt);  
    Integer get(int index);
```

List<Number> non è un supertipo di List<Integer>

16

Una discussione (per capire anche la ricerca nel settore)



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

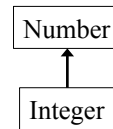
```
type List<Number>:  
    Number get(int index);
```

```
type List<Integer>:  
    Integer get(int index);
```

La nozione di sottotipo *covariante* sarebbe corretta:

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

Sfortunatamente Java non adotta questa soluzione



17

Allora parliamo anche di contravarianza



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

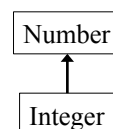
```
type List<Number>:  
    boolean add(Number elt);
```

```
type List<Integer>:  
    boolean add(Integer elt);
```

La nozione di sottotipo *contravariante* sarebbe altrettanto corretta:

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

Ma Java



18

Altri aspetti



- `List<Integer>` e `List<Number>` non sono correlati dalla nozione di sottotipo
- Tuttavia in diversi casi la nozione di sottotipo sui generici funziona come uno se lo aspetta 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>`
 - ...

19

addAll



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

Quale è il miglior tipo per il parametro formale?

- Il piu' ampio possibile...
- ... che permette di avere implementazioni corrette

20

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.
Spiacevole

21

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);`

Troppo restrittivo:

- Parametro attuale di tipo `List<Integer>` per `Set<Number>` 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

22

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 puo' modificare la collection `c`

23

Altro esempio



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

Va bene ma questo è meglio:

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

24

Wildcards



Sintassi delle wildcards:

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

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 e' l'unicità del tipo

25

Esempi



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

- Maggiormente flessibile rispetto a
`void addAll(Collection<E> c);`
- Espressiva come
`<T extends E> void addAll(Collection<T> c);`

26



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);
```

27



? vs Object

? Tipo particolare anonimo

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

Quale e' la differenza tra **List<?>** e **List<Object>**:

- Possiamo istanziare ? Con un tipo qualunque: **Object, String, ...**
- **List<Object>** è piu' restrittivo: **List<String>** non va bene.

Quale e' 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**

28

Java arrays



Sappiamo bene come operare con gli array in Java ... Vero?

Analizziamo questa classe:

```
class Array<T> {  
    public T get(int i) { ... "op" ... }  
    public T set(T newVal, int i) { ... "op" ... }  
}
```

Domanda: Se **Type1** è un sottotipo di **Type2**, quale è la relazione tra **Type1 []** e **Type2 []**??

29

Sorpresa!



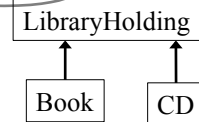
- ✎ Sappiamo che per i generici la nozione di sottotipo è invariante, pertanto se **Type1** è un sottotipo di **Type2**, allora **Type1 []** e **Type2 []** non sono correlati
- ✎ Ma Java è strano, se **Type1** è un sottotipo di **Type2**, allora **Type1 []** è un sottotipo di **Type2 []**
 - Java (ma 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)

30

Ma ci sono anche cose “buone”



Gli inglesi dicono: “Programmers do okay stuff”



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

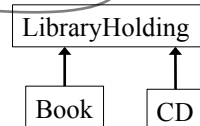
// cliente
Book[] books = ...;
maybeSwap(books); // usa la covarianza degli array
```

31

Ma puo' 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!!
```

32

Le scelte di Java



- ✎ Tipo dinamico è un sottotipo di quello statico
 - Violto nel caso di `Book b`
- ✎ La scelta di Java:
 - Ogni array “conosce” il suo tipo dinamico (`Book []`)
 - Modificare a (run-time) coun 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

33

Troppo bello per essere vero: 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
```

34

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
}
```

35

equals



```
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());
    }
    ...
}
```

Erasure: tipo dell'argomento non esiste a runtime

36

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

37

Tips (da stackoverflow)



- ✎ 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

38

Java Generics (JG)




- Il compilatore verifica l'uso 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 e' 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	✗	✓	✗	✗	✗



ANTICIPAZIONE DI PROGRAMMAZIONE AVANZATA



Generic C#

- Kennedy and Syme have extended CLR to support parametric types (the same proposal has been made for PolyJ by Cartwright and Steele)
- The verifier, JIT and loader have been changed
- When the program needs an instantiation of a generic type the loader generates the appropriate type
- The JIT can share implementation of reference instantiations (Stack<String> has essentially the same code of Stack<Object>)

Generic C# compiler



- GC# compiler implements a JG like notation for parametric types
- Bounds are the same as in JG
- NO type-inference on generic methods: the type must be specified in the call
- Exact runtime types are granted by CLR so virtual generic methods are allowed
- All type constructors can be parameterized: struct, classes, interfaces and delegates.

Esempio



```
using System;
namespace n {
    public class Foo<T> {
        T[] v;
        Foo() { v = new T[15]; }
        public static
        void Main(string[] args) {
            Foo<string> f =
                new Foo<string>();
            f.v[0] = "Hello";
            string h = f.v[0];
            Console.Write(h);
        }
    }
}

.field private !0[] v
.method private hidebysig
    specialname
    rtspecialname
instance void .ctor() cil
managed {
    .maxstack 2
    ldarg.0
    call         instance void
[mscorlib]System.Object::.c
    tor()
    ldarg.0
    ldc.i4.s    15
    newarr     !0
    stfld     !0[] class n.Foo<!
0>::v
    ret
} // end of method Foo::.ctor
```