
PROGRAMMAZIONE 2

13. Java Generics

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 e ...

```
// 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>>
...
```

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 elt);  
    E get(int index);  
}
```

- Dichiarazione di un parametro di tipo
- Istanziabile con un qualunque tipo
 - List<String>, etc.

Variabili di tipo

Dichiarazione

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

Utilizzo

Dichiarare classi generiche

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

- convenzioni standard
 - 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>
```

Istanziare 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 è un sottotipo di Object
```

```
List2<Date> // compile-time error, Date non è  
// sottotipo di Number
```

Visione effettiva dei generici

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

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

Istanziamento identico

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

- Ma *compile-time error* se il tipo non è un sotto-tipo del limite superiore della gerarchia

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

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


Vincoli di tipo

<TypeVar extends SuperType>

- *upper bound*: va bene il super-tipo o uno dei suoi sotto-tipi

<TypeVar extends ClA & InterB & InterB & ... >

- *multiple* upper bounds

<TypeVar super SubType>

- *lower bound*: va bene il sotto-tipo o uno qualunque dei suoi super-tipi

Esempio

```
// strutture di ordine su alberi
```

```
public class TreeSet<T extends Comparable<T>> { ... }
```

Esempio di metodi ausiliari

```
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);  
    }  
}
```

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.size
        return lst.get(i);
    }
}
```

Dichiarare nella firma
i vincoli sui tipi

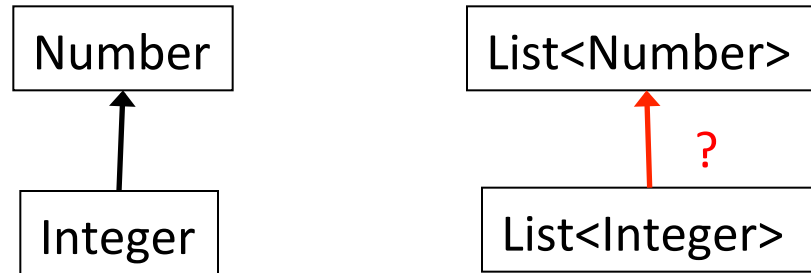
Metodi generici

- Metodi che possono usare i tipi generici delle classi
- Possono dichiarare anche i loro tipi generici
- Le invocazioni di metodi generici devono *obbligatoriamente* istanziare i parametri di tipo
 - staticamente: una forma di inferenza di tipo

Esempio di tipo istanziato dalla classe

```
public class InsertionSort<T extends Comparable<T>> {  
    public void sort(T[] x) {  
        T tmp;  
        for (int i = 1; i < x.length; i++) {  
            // invariant: x[0],...,x[i-1] sorted  
            tmp = x[i]; int j;  
            for (j = i;  
                j > 0 && x[j-1].compareTo(tmp) > 0; j--)  
                x[j] = x[j-1];  
            x[j] = tmp;  
        }  
    }  
}
```

I generici e la nozione di sotto-tipo



- **Integer** è un sotto-tipo di **Number**
- **List<Integer>** è un sotto-tipo di **List<Number>**?

Quali sono le regole di Java?

Anche se **Type2** è un sotto-tipo di **Type3**, se i due tipi sono diversi allora **Type1<Type2>** *non* è un sotto-tipo di **Type1<Type3>**

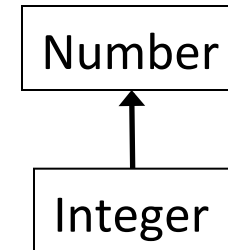
Formalmente: la nozione di sotto-tipo usata in Java è *invariante* per le classi generiche

Esempi (da Java)

- Tipi e sotto-tipi
 - Integer è un sottotipo di Number
 - ArrayList<E> è un sotto-tipo di List<E>
 - List<E> è un sotto-tipo di Collection<E>
- Ma
 - List<Integer> non è un sotto-tipo di List<Number>

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);
```

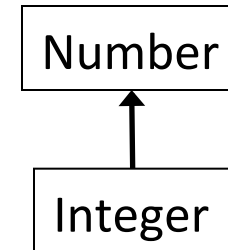
List<Number> non è un super-tipo di List<Integer>

Una discussione (per capire la ricerca nel settore)

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

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

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



La nozione di sotto-tipo *covariante* sarebbe corretta

- **List<Integer>** sotto-tipo di **List<Number>**

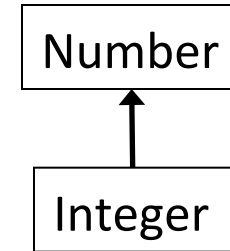
Sfortunatamente Java non adotta questa soluzione

Allora parliamo anche di contravarianza

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

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

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



La nozione di sotto-tipo *contravariante* sarebbe altrettanto corretta

- `List<Number>` è sotto-tipo di `List<Integer>`

Ma Java ...

Altri aspetti

- **List<Integer>** e **List<Number>** non sono correlati dalla nozione di sotto-tipo
- Tuttavia, sul primo argomento la nozione di sotto-tipo sui generici funziona come uno se lo aspetta anche in Java
- Esempio: assumiamo che **LargeBag** extends **Bag**, allora
 - **LargeBag<Integer>** è un sotto-tipo di **Bag<Integer>**
 - **LargeBag<Number>** è un sotto-tipo di **Bag<Number>**
 - **LargeBag<String>** è un sotto-tipo di **Bag<String>**
 - ...

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 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 è `void addAll(Set<E> c);`

Troppo restrittivo

- un parametro attuale di tipo `List<E>` non sarebbe permesso, e ciò è 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);`

Troppo restrittivo

- il parametro attuale di tipo `List<Integer>` per `Set<Number>` non va bene anche se `addAll` ha solo bisogno di leggere da `c` e 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 può vedere nell'implementazione il tipo `T`, sa solo che è un sotto-tipo di `E`, 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);  
}
```

La soluzione va bene, ma ancora meglio

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

Wildcard

Sintassi delle wildcard

- ? **extends Type**, sotto-tipo non specificato del tipo **Type**
- ? notazione semplificata per ? **extends Object**
- ? **super Type**, super-tipo 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 sa è l'unicità del tipo

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);`

Producer Extends, Consumer Super

Quando si usano le wildcard?

- si usa ? **extends T** nei casi in cui si vogliono ottenere dei valori (da un produttore di valori)
- si usa ? **super T** nei casi in cui si vogliono inserire valori (in un consumatore)
- non vanno usate (basta **T**) quando si ottengono e si producono valori

```
<T> void copy(List<? super T> dst,  
             List<? extends T> src) ;
```

? vs Object

? Tipo particolare anonimo

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

Quale differenza tra **List<?>** e **List<Object>**?

- possiamo istanziare ? con un tipo qualunque: **Object**, **String**, ...
- **List<Object>** è più restrittivo: **List<String>** non va bene

Quale differenza tra **List<Foo>** e **List<? extends Foo>**?

- nel secondo caso il tipo anonimo è un sotto-tipo sconosciuto di **Foo**

List<? extends Animal>

può memorizzare **Giraffe** ma non anche **Zebre**

Raw types

- Con *raw type* si indica una classe/interfaccia senza nessun argomento di tipo (*legacy code*)

```
Vector<Integer> intVec = new Vector<Integer>();  
Vector rawVec = new Vector(); // OK
```

```
Vector<String> stringVec = new Vector<String>();  
Vector rawVec = stringVec; // OK
```

```
Vector rawVec = new Vector();  
    // rawVector is a raw type of Vector<T>  
Vector<Integer> intVector = rawVector;  
    // warning: unchecked conversion
```

Inferenza di tipo

- Inferenza: capacità di controllare una invocazione di metodo e la dichiarazione di metodo associata al fine di determinare il tipo *più specifico*, in accordo ai vincoli di tipo presenti, che rende il metodo effettivamente invocabile

```
class MyClass<S> {  
    <T> MyClass(T t) { ... }  
}
```

```
new MyClass<Integer>("")
```

diamond notation

```
MyClass<Integer> myObject = new MyClass<>("")
```

Ancora l'inferenza di tipo

```
static <T> List<T> emptyList();
```

metodo in Collections

```
List<String> listOne = Collections.<String>emptyList();
```

```
List<String> listOne = Collections.emptyList();
```

```
void processStringList(List<String> stringList) { ... }
```

```
processStringList(Collections.<String>emptyList());
```

```
processStringList(Collections.emptyList());
```

Java 8 inferisce dal tipo del parametro

obbligatorio in Java 7

Java array

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 sotto-tipo di **Type2**, che relazione esiste tra **Type1 []** e **Type2 []**??

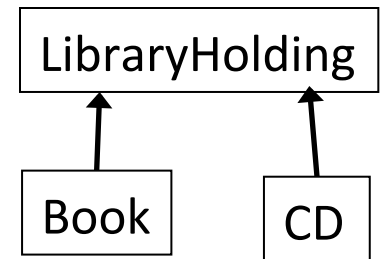
Sorpresa!

- Sappiamo che per i generici la nozione di sottotipo è invariante, pertanto anche se **Type1** è un sotto-tipo di **Type2**, i tipi **Type1[]** e **Type2[]** non dovrebbero essere correlati
- Ma Java ha delle peculiarità: se **Type1** è un sotto-tipo di **Type2**, allora **Type1[]** è un sotto-tipo di **Type2[]**
 - Java (e C#) ha 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 principi dei linguaggi di programmazione)

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
```

Ma può andare male

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

```
// il solito cliente
```

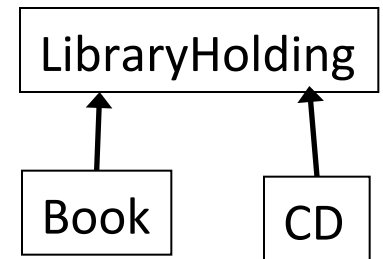
```
Book[] books = ...;
```

```
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

- Il tipo dinamico è un sotto-tipo 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 super-tipo 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**

Too good to be true: 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 doc. Java: “Compiler gives an unchecked warning, since this is something the run-time system *will not check for you*”

Problema

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

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

Erase: il tipo dell'argomento non esiste a run-time

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 sia E

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

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
- Limite principale: il tipo effettivo è perso a run-time a causa della type erasure
- Tutte le istanziazioni sono identificate
- Esistono altre implementazioni dei generici per Java

Altri generici per 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	✗	✓	✗	✗	✗