

Tipi Generici Covarianza & Controvarianza

Sorgente:

Prof. Enrico Denti

Fondamenti di Informatica T-2 - Corso di Laurea in Ingegneria
Informatica

Universita' di Bologna

JCF "CLASSICA": PROBLEMI

- Abbiamo già visto che **usare il tipo Object per fare contenitori generici** causa **seri problemi**
 - di fatto, equivale ad abolire il controllo di tipo
 - ergo, *operazioni sintatticamente corrette* possono risultare *semanticamente errate* → errori a run-time
- Per questo, **la JCF classica non è "type safe"**
 - la correttezza delle operazioni è affidata a "commenti sul corretto uso" anziché ai controlli del compilatore
- **Soluzione: il nuovo concetto di TIPO GENERICO**
 - Notazione **<TIPO>**

ESEMPIO

- Anziché collezioni di Object
 - *in cui di fatto si può mettere qualunque cosa...*
- definiamo **collezioni di T**
 - *essendo T un TIPO GENERICO*

PRIMA

```
List myIntList = new ArrayList(); // list of integers
myIntList.add(113);
Integer i = (Integer) myIntList.get(0);
```

DOPO

```
List<Integer> myList = new ArrayList<Integer>();
myList.add(113);
Integer i = myList.get(0); // non serve più il cast
myList.add("ahahahah"); // TYPE ERROR!
```

ARRAY: PECULIARITÀ

A differenza delle collection, **gli array possono essere definiti di uno specifico tipo** (non solo di Object) e ciò apre nuovi scenari... e nuovi problemi.

Si consideri il seguente frammento di codice:

```
Integer[] myArray = new Integer[10];  
myArray[0] = 113;  
Integer i = myArray[0];
```

- Stavolta, il fatto che si ammettano solo interi è *formalmente espresso* perché il costrutto array lo permette.
- Ergo, **se il vincolo viene violato, il compilatore se ne accorge:**

```
Integer[] myArray = new Integer[10];  
myArray[0] = "ahahahah"; // TYPE ERROR DETECTED  
Integer i = myArray[0];
```

UN NUOVO PROBLEMA

Tutto a posto, dunque?

NO, perché poter specificare *array di tipi diversi* fa nascere nuove domande.. e nuovi dubbi.

- in particolare: *tipi di array diversi sono compatibili?*

Si consideri il seguente frammento di codice:

```
Integer[] arrayOfInt = new Integer[4];  
arrayOfInt[0] = new Integer(12);  
  
Object[] arrayOfObjects = arrayOfInt; // ??
```

- *Poiché Integer deriva da Object, "non sembra insensato"* che gli array di Integer possano essere compatibili con gli array di Object.

MA.. avrà davvero senso? Vediamo.

UN NUOVO PROBLEMA

Se quella frase è lecita, ora si può accedere *allo stesso array* in due modi:

- tramite il riferimento `arrayOfInt`
- *tramite il nuovo riferimento `arrayOfObjects`*

Peccato che il primo accetti solo interi, il secondo no!

Si consideri il seguente frammento di codice:

```
Integer[] arrayOfInt = new Integer[4];  
arrayOfInt[0] = new Integer(12);  
  
Object[] arrayOfObjects = arrayOfInt;  
arrayOfObjects[1] = "ciao"; // ATTENZIONE!
```

.. e ora, che succede??

IL PROBLEMA

Qual è il punto?

```
Integer[] arrayOfIntegers = new Integer[4];  
arrayOfIntegers[0] = new Integer(12);  
Object[] arrayOfObjects = arrayOfIntegers;  
arrayOfObjects[1] = "ciao"; //ASSURDO SEMANTICO
```

- Poiché una stringa è un Object, ***l'assegnamento è formalmente corretto ma SEMANTICAMENTE ASSURDO***, perché l'array "vero" sottostante è un array di Integer e come tale non può ospitare stringhe!
- CONSEQUENZA: **come prima, la compilazione passa**, ma poi..

```
Exception in thread "main"  
java.lang.ArrayStoreException: String
```

CONCLUSIONE

Di nuovo, la compilazione corretta non ha impedito di ritrovarsi con un programma sbagliato, perché *l'errore di progetto non è stato "smascherato" dal type system.*

MORALE: *neppure gli array sono "type safe" (sicuri sotto il profilo dei tipi), perché la gestione delle compatibilità di tipo non garantisce che un programma che passi la compilazione sia corretto.*

OSSERVAZIONE: *ciò è accaduto nonostante stavolta il tipo fosse specificato nell'array. Dunque, siamo di fronte a un problema diverso dal precedente... anche più subdolo !*

NUOVO PROBLEMA...

- Evidentemente, **c'è un problema di fondo nel considerare compatibili array di tipi diversi, anche quando tali tipi siano fra loro compatibili**
 - array di Object e array di Integer NON dovrebbero essere considerati "parenti" in modo superficiale...
 - ... *nonostante* Integer SIA un sottotipo di Object !
- Questo con gli array è in realtà la spia di un **problema molto più vasto e generale, che si manifesta con qualsunque collection**
 - negli array i problemi appaiono nel *memorizzare oggetti* (non nell'estrarli), ma in altri casi succede l'opposto
 - sono problemi di COVARIANZA & CONTROVARIANZA

... NUOVA SOLUZIONE

- Per venirne a capo occorre una scelta drastica:
*stabilire assoluta incompatibilità
fra collezioni di tipi diversi*
 - riservandoci però di predisporre *idonei meccanismi
per casi particolari "sicuri"*
- Perciò, *meglio evitare gli array nei casi critici*
 - per retrocompatibilità, non sono stati modificati
 - sono *unsafe* in determinate situazioni
 - *ci sono alternative migliori: List e altre collection!*

NUOVO PROBLEMA : SOLUZIONE

PRIMA (array)

```
Integer[] arrayOfInt = new Integer[4];  
arrayOfInt[0] = 12;  
Object[] arrayOfObjects = arrayOfInt;    // NESSUNO PROTESTA..  
arrayOfObjects[1] = "ciao"; // scaviamoci la fossa...
```

PRIMA (collections)

```
List listOfInt = new LinkedList();  
listOfInt.add(12);  
List listOfObjects = listOfInt;           // NESSUNO PROTESTA..  
listOfObjects.add("ciao"); // scaviamoci la fossa...
```

DOPO (collections)

```
List<Integer> listOfInt = new ArrayList<Integer>();  
listOfInt.add(12);  
List<Object> listOfObjects = listOfInt; // STRONCATO
```

NUOVO PROBLEMA : SOLUZIONE

- *Stavolta, è esplicitamente detto che le due liste sono di TIPI DIVERSI e perciò ASSOLUTAMENTE INCOMPATIBILI*
- *Il compilatore lo sa e agisce di conseguenza, stroncando l'assegnamento che minerebbe la type safety*

```
TestEs2.java:7: incompatible types
found    : java.util.List<java.lang.Integer>
required: java.util.List<java.lang.Object>
    List<Object> listOfObjects = listOfInt;
                                   ^ 1 error
```

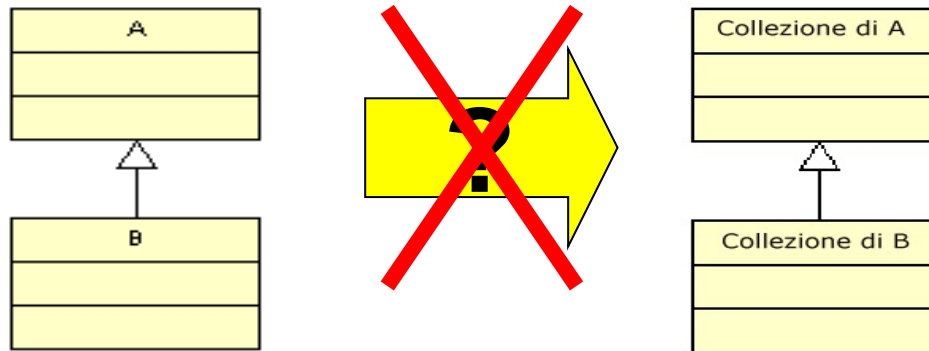
DOPO (collections)

```
List<Integer> listOfInt = new ArrayList<Integer>();
listOfInt.add(12);
List<Object> listOfObjects = listOfInt; // STRONCATO
```

CONCLUSIONE

Nel nuovo approccio, per scelta di progetto:

- se B è un sottotipo da A
- **"Collezione di B" NON è un sottotipo di "Collezione di A"**, così da prevenire a priori le conseguenze assurde viste.



Naturalmente, occorrerà evitare eccessive rigidità:

- **ALCUNE operazioni ben precise e selezionate potrebbero anche essere sensate e sicure dal punto di vista dei tipi..**
- **..ergo, cercheremo un modo per autorizzare solo quelle.**¹³

CLASSI GENERICHE

Il tipo generico si può usare anche per definire **classi, metodi, riferimenti, interfacce**.

Si consideri ad esempio la classe seguente:

```
public class MyStack<T> {  
    private List<T> storage;  
    public MyStack() { storage = new ArrayList<T>(); }  
    public void push(T elem) { storage.add(elem); }  
    public T pop() { return storage.remove(storage.size()-1); }  
    public boolean isEmpty() { return storage.isEmpty(); }  
}
```

OSSERVA: nel costruttore
NON si mette <T>

- **Questa classe definisce uno STACK GENERICO** (struttura dati ad accesso *LIFO – Last In, First Out*) appoggiandosi a una lista.

CLASSI GENERICHE

Un possibile cliente potrebbe quindi essere questo:

```
MyStack<Integer> stack = new MyStack<Integer>();  
stack.push(18); stack.push(22); stack.push(34);  
while (!stack.isEmpty()) {  
    System.out.println(stack.pop());  
}
```

Qui però il tipo effettivo
<Integer> ci vuole..

34
22
18

Politica LIFO:

- inserimento nell'ordine di 18, 22, 34
- estrazione in ordine inverso (34, 22, 18)

OSSERVA:

- **il tipo effettivo** usato per definire e creare lo stack a run-time **non può essere un tipo primitivo**
- tuttavia, grazie a boxing/unboxing automatici, è comunque *possibile inserire ed estrarre valori primitivi* senza complicazioni.

UN PICCOLO MIGLIORAMENTO

Se si devono fare più push consecutive, il nostro stack obbliga il cliente a scrivere:

```
stack.push(18) ; stack.push(22) ; stack.push(34) ;
```

Sarebbe più pratico poter scrivere semplicemente:

```
stack.push(18) .push(22) .push(34) ;
```

Si può? ***Certo che sì, ed è anche molto semplice!***

Basta cambiare leggermente la push:

- non più tipo di ritorno `void`, ma tipo di ritorno.. *stack* !
- restituendo poi semplicemente lo stack stesso: `this` 😊

Questo schema si chiama ***cascading*** ed è molto comodo: il classico "uovo di Colombo"... 😊

LO STACK REVISED

Ecco lo stack con la push migliorata:

```
public class MyStack<T> {  
    private List<T> storage;  
    public MyStack(){ storage = new ArrayList<T>(); }  
    public MyStack<T> push(T elem) {  
        storage.add(elem); return this; // CASCADING  
    }  
    public T pop(){ return storage.remove(storage.size()-1); }  
    public boolean isEmpty() { return storage.isEmpty(); }  
}
```

- Ora la push, dopo aver svolto la sua operazione, *restituisce un riferimento allo stack stesso*, che può così essere usato dal cliente per.. invocare una nuova push in cascata ☺

```
stack.push(18) .push(22) .push(34); // CASCADING
```

CONTROLLI DI TIPO

Verifichiamo l'efficacia del controllo di tipo:

```
MyStack<Integer> stack1 = new MyStack<Integer>() ;  
stack1.push(18).push(22).push(34) ;  
MyStack<Double> stack2 = new MyStack<Double>() ;  
stack2.push(1.8).push(22.0).push(0.34) ;
```

Come prevedibile, **il controllo di tipo è *STRINGENTE***:

- `stack2` accetta solo `double` e null'altro (né `float`, né `int`)
- i due stack sono totalmente incompatibili fra loro

```
MyStack<Integer> stack1 = new MyStack<Integer>() ;  
MyStack<Double> stack2 = new MyStack<Double>() ;  
stack2.push(18).push(22.0F).push(0.34) ;
```

NO! 18 è int

NO! 22.0F è float

CONTROLLI DI TIPO

Un possibile cliente potrebbe quindi essere questo:

```
MyStack<Integer> stack1 = new MyStack<Integer>();  
stack1.push(18).push(22).push(34);  
MyStack<Double> stack2 = new MyStack<Double>();  
stack2.push(1.8).push(22.0).push(0.34);
```

Come prevedibile, **il controllo di tipo è *STRINGENTE***:

- stack2 accetta solo double e null'altro (né float, né int)
- i due stack sono totalmente incompatibili fra loro

```
MyStack<Integer> stack1 = new MyStack<Integer>();  
MyStack<Double> stack2 = new MyStack<Double>();  
stack2.push(18).push(22.0F).push(0.34);
```

```
push(Double) in MyStack<Double> cannot be applied to (int)... to (float)  
stack2.push(18).push(22.0F).push(3.4);  
                ^      ^
```

2 errors

CONTROLLI DI TIPO

..e ovviamente, ogni tentativo di mischiarli è stroncato:

```
MyStack<Integer> stack1 = new MyStack<Integer>() ;  
MyStack<Double> stack2 = new MyStack<Double>() ;  
stack1 = stack2; // STRONCATO!  
stack2 = stack1; // STRONCATO!
```

```
TestEs3.java:10: incompatible types  
found    : MyStack<java.lang.Double>  
required: MyStack<java.lang.Integer>  
    stack1 = stack2; // STRONCATO!  
           ^
```

```
TestEs3.java:11: incompatible types  
found    : MyStack<java.lang.Integer>  
required: MyStack<java.lang.Double>  
    stack2 = stack1; // STRONCATO!  
           ^
```

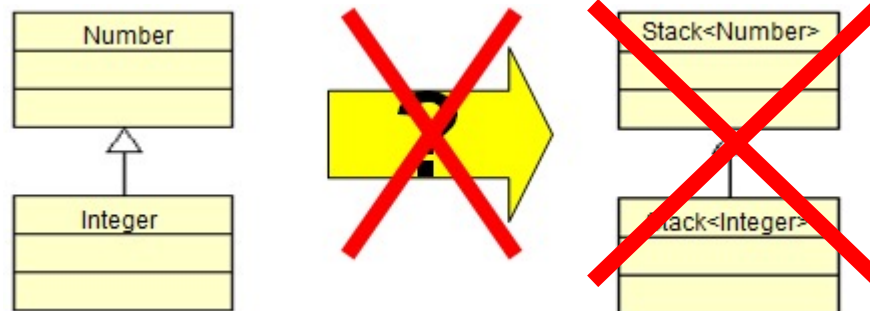
2 errors

CONTROLLI DI TIPO

Il mix è stroncato anche se un elemento deriva dall'altro
(nello specifico, Integer deriva da Number):

```
MyStack<Integer> stack1 = new MyStack<Integer>();  
MyStack<Number> stack0 = new MyStack<Number>();  
stack0 = stack1; // STRONCATO!
```

```
TestEs3.java:10: incompatible types  
found   : MyStack<java.lang.Integer>  
required: MyStack<java.lang.Number>  
    stack0 = stack1; // STRONCATO!  
                ^
```



UNA PICCOLA SEGNALAZIONE

E se ci venisse voglia di definire *array generici* ?

**Purtroppo, in Java vi sono *alcune limitazioni* (dovute alla compatibilità con la JVM precedente..):
*gli array generici si possono dichiarare, ma non allocare direttamente.***

Ad esempio, questo codice in Java NON è ammesso:

```
public class MyStackBis<T> {  
    private <T>[] storage;    // la dichiarazione è OK  
    public MyStackBis(int size) {  
        storage = new T[size]; // ALLOC. GENERICA NON AMMESSA!  
    }  
    ...  
}
```

**L'allocazione diretta di array generici
non è possibile in Java (ma lo è in C#)**

Esistono trucchi per "girarci intorno"..
...ma la cosa migliore è evitare gli array!

LO STESSO ESEMPIO IN C#

Il supporto ai tipi generici di C# è migliore e più completo rispetto a quello di Java

- nessuna necessità di classi wrapper → tipi primitivi ammessi
- sintassi spesso più chiara e lineare

```
class Stack<T> {  
    private System.Collections.Generic.List<T> storage;  
    public Stack(){  
        storage = new System.Collections.Generic.List<T>();  
    }  
    public Stack<T> push(T elem){ storage.Add(elem); return this; }  
    public T pop(){  
        T topElement = storage[storage.Count-1];  
        storage.RemoveAt(storage.Count-1); return topElement; }  
    public bool isEmpty() { return storage.Count==0; }  
}
```

Sintassi array-like:

- parentesi quadre per accedere a elementi
- proprietà Count anziché metodi size()

LO STESSO ESEMPIO IN C#

Analogamente, il cliente diventa:

```
Stack<int> stack1 = new Stack<int>();  
stack1.push(18).push(22).push(34);  
Stack<double> stack2 = new Stack<double>();  
stack2.push(1.8).push(22.0).push(3.4);
```

- Tipi primitivi usati direttamente, senza wrapper classes

..e volendo potremmo fare lo stack anche con gli array: 😊

```
class MyStack<T> {  
    private T[] storage;  
    private int elements;  
    public MyStack(int size) {  
        storage = new T[size]; elements = 0;  
    }  
    public void push(T elem) { storage[elements++] = elem; }  
    public T pop() { return storage[--elements]; }  
}
```

***.. ma non è poi una grande idea:
uno stack non dovrebbe avere una
dimensione massima prefissata!***

TIPI GENERICI: LIMITI

Come l'esempio dello stack ha mostrato, **coi tipi generici:**

- **si risolve il Problema 1**, potendo esprimere vincoli chiari sul tipo degli oggetti ammessi → **SODDISFACENTE**
- **si risolve il Problema 2**, rendendo incompatibili collezioni di tipi diversi → **CORRETTO MA RIGIDO**

In effetti, ripensando al nostro stack... *non sarebbe così assurdo pensare di poter essere un po' più flessibili.*

- è vero che uno stack di Number è cosa diversa da uno stack di Integer (e dunque è giusto che siano incompatibili)..
- ...ma è anche vero che operazioni come, ad esempio, *spostare o copiare uno stack di Integer in uno stack di Number sarebbero senza rischi: peccato dovervi rinunciare ... !*

UNO STACK "AMPLIATO"

Si consideri il seguente stack ampliato:

```
public class MyStack<T> {  
    private List<T> storage;  
    public MyStack(){ storage = new ArrayList<T>(); }  
    public MyStack<T> push(T elem){  
        storage.add(elem); return this; }  
    public T pop(){ return storage.remove(storage.size()-1);}  
    public boolean isEmpty() { return storage.isEmpty(); }  
    public void moveFrom(MyStack<T> source){  
        while(!source.isEmpty()) push(source.pop());  
    }  
}
```

In effetti, come **source** andrebbe bene **un qualunque stack di cose "almeno pari" a T**, non necessariamente proprio (e solo) T.

- Il metodo **moveFrom**, che sposta gli elementi dallo stack **source** allo stack corrente, è costretto a operare solo su **MyStack<T>**
- **ma in realtà tale vincolo è una limitazione inutile, perché questa operazione avrebbe perfettamente senso in molti altri casi !**

UNO STACK "AMPLIATO"

Verifichiamo:

```
MyStack<Integer> stack1 = new MyStack<Integer>();  
MyStack<Integer> stack2 = new MyStack<Integer>();  
MyStack<Number> stack0 = new MyStack<Number>();  
// stack0 = stack1; // continua a essere stroncato (giustamente)  
stack1.moveFrom(stack2); // ok, tutto normale  
stack0.moveFrom(stack1); // STRONCATO INUTILMENTE
```

Come già osservato, *considerato quel che moveFrom deve fare, è un peccato che questa operazione sia stata stroncata* perché in realtà avrebbe avuto perfettamente senso e avrebbe quindi potuto tranquillamente funzionare.

Dobbiamo proprio subire questi vincoli anche quando inutili..?

TIPI GENERICI: CONCLUSIONI

Occorre un **modo per coniugare due esigenze opposte:**

- da un lato, **l'incompatibilità rigida** fra collezioni di tipi diversi, **necessaria** per far fronte al Problema 2
- dall'altro, **la opportunità di permettere comunque alcune operazioni** fra collezioni di tipi diversi quando sensate.

Serve un modo per *ESPRIMERE "DEROGHE MIRATE"*
alla inflessibilità dell'approccio fin qui visto.

Da qui nasce il concetto di
TIPO PARAMETRICO VARIANTE.

Tipi parametrici varianti

Covarianza, controvarianza, bivarianza

Dai TIPI GENERICI a TIPI PARAMETRICI VARIANTI ("WILDCARD")

- Senza rinunciare a quanto conquistato, ha senso pensare di *esprimere DEROGHE MIRATE* all'attuale rigido controllo di tipo, in specifiche situazioni prive di rischi.
- Più precisamente, come suggerito dall'esempio, ha senso pensare di *essere più flessibili nei tipi di collezioni accettati come argomenti (o restituiti) da singoli metodi.*
- Per questo fine nasce una **NUOVA NOTAZIONE**: quella dei *tipi parametrici varianti*, comunemente detti *"wildcard"*
- Tale notazione è usabile *solo ed esclusivamente per specificare collezioni che siano argomenti (o tipi di ritorno) di metodi*, NON per definire tipi, classi, etc.

TIPI PARAMETRICI VARIANTI (wildcard)

NUOVE NOTAZIONI:

<E extends T> (se non occorre il nome: **<? extends T>**)

- specifica che potrà essere passato in quel punto un argomento di qualunque tipo E che estenda T

<E super T> (se non occorre il nome: **<? super T>**)

- specifica che potrà essere passato in quel punto un argomento di qualunque tipo E che "stia sopra" T (ovvero, sia esteso da T)

<?>

- specifica che potrà essere passato in quel punto un argomento di qualunque tipo

Naturalmente, la scelta dovrà essere coerente con l'uso che il metodo o funzione farà di tale argomento al suo interno.

LO STACK.. CON WILDCARD (1)

Si consideri il seguente stack ampliato:

```
public class MyStack<T> {  
    private List<T> storage;  
    public MyStack(){ storage = new ArrayList<T>(); }  
    public MyStack<T> push(T elem){  
        storage.add(elem); return this; }  
    public T pop(){ return storage.remove(storage.size()-1);}  
    public boolean isEmpty() { return storage.isEmpty(); }  
    public <E extends T> void moveFrom(MyStack<E> source){  
        while(!source.isEmpty()) push(source.pop());  
    }  
}
```

Si specifica che **come source va bene un qualunque stack di cose che estendano T**, non necessariamente proprio (e solo) T.

- Così facendo, **il metodo moveFrom può ora operare su tipi di stack anche diversi da MyStack<T>**, purché si tratti di stack di "cose che estendono T" – guarda caso, **proprio quelle che garantiscono di rendere sicura e sensata l'operazione moveFrom.**

LO STACK.. CON WILDCARD (2)

La `moveFrom` precedente è corretta, ma espressa in modo *inutilmente prolisso*:

- *perché introdurre il tipo `E`, se dentro la funzione non serve?*

Si può semplificare la scrittura rendendola più chiara e comprensibile tramite la **notazione `<? extends T>`**

SINTASSI PROLISSA

```
public <E extends T> void moveFrom(MyStack<E> source) {  
    while(!source.isEmpty()) push(source.pop());  
}
```

SINTASSI WILDCARD

wildcard

```
public void moveFrom(MyStack<? extends T> source) {  
    while(!source.isEmpty()) push(source.pop());  
}
```

Dice quello che intendevamo in modo più diretto e compatto: che come `source` va bene *un qualunque stack di "cose che estendano `T`"*

Esempio

```
public class EsStack
{ public static void main(String args[])
  { MyStack<Integer> src= new MyStack<Integer>();
    MyStack<Number> dest= MyStack<Number>();
    src.push(10); src.push(29); src.push(34);
    dest.moveFrom(src);
    System.out.println(dest);
  }
}
```

UN CASO "CURIOSO" (1)

Si vuole scrivere *una funzione che copi tutti gli elementi di una collezione in un'altra*.

- NON può essere un metodo di `Collection`, perché il codice di `Collection` non lo controlliamo noi: sarà perciò una funzione statica di una classe di utilità.

In prima battuta, verrebbe spontaneo definirla così:

```
public class MyUtils{  
    public static <T> void copy(Collection<T> src, Collection<T>  
dest){  
        for (T elem : src) dest.add(elem);  
    }  
}
```

.. ma sarebbe molto limitativo, perché in questo modo potrebbe operare solo su collezioni di `T` (e null'altro!)

UN CASO "CURIOSO" (2)

Una definizione migliore può sfruttare i tipi parametrici per allentare i vincoli sulla collezione di destinazione:

```
public class MyUtils{  
    public static <T> void copy(Collection<? extends T> src,  
                                Collection<T> dest){  
        for (T elem : src) dest.add(elem);  
    }  
}
```

che è già migliore perché permette alla funzione di operare in lettura anche su collezioni diverse da quelle di T.

Tuttavia, è ancora limitata a scrivere gli elementi in collezioni di T (e null'altro): ma perché?

Si può fare di meglio!

UN CASO "CURIOSO" (3)

La definizione migliore in assoluto è quella che permette di rimuovere TUTTI i vincoli inutili:

```
public class MyUtils{  
    public static <T> void copy(Collection<? extends T> src,  
                               Collection<? super T> dest){  
        for (T elem : src) dest.add(elem);  
    }  
}
```

che permette alla funzione non solo di operare in lettura su collezioni diverse da quelle di T

- nello specifico: collezioni di cose "almeno pari" a T

ma anche di operare in scrittura su collezioni diverse da quelle di T

- nello specifico: collezioni di cose "al più pari" a T

UN CASO "CURIOSO" (4)

La definizione migliore in assoluto è quella che permette di rimuovere TUTTI i vincoli inutili:

```
public class MyUtils{  
    public static <T> void copy(Collection<? extends T> src,  
                                Collection<? super T> dest){  
        for (T elem : src) dest.add(elem);  
    }  
}
```

È il *vincolo minimo* che garantisce la correttezza dell'operazione di assegnamento interno:

- sorgente "almeno" di elementi di tipo T
- destinazione "al più" di elementi di tipo T

USO

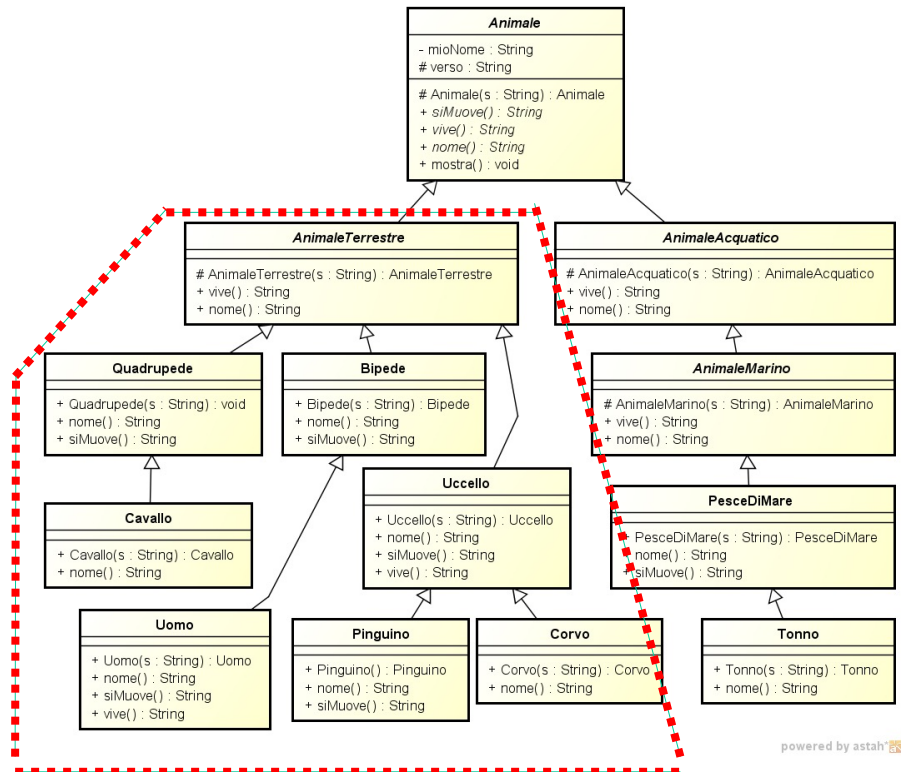
```
import java.util.*;
public class Main
{
    public static void main(String[] args)
    {
        List<Integer> l1=new ArrayList<Integer>();
        List<Number> l2=new ArrayList<Number>();
        l1.add(10);
        l1.add(20);
        l1.add(30);
        MyUtils.copy(l1,l2);
        System.out.println(l2);
    }
}
```

INTERPRETAZIONE (1)

Le notazioni **<? extends T>** e **<? super T>** si possono interpretare come **upper bound** e **lower bound** dei "tipi accettabili" in una collezione (rispetto a una tassonomia)

ESEMPIO:

- un argomento di tipo `List<AnimaleTerrestre>` è compatibile solo con altre `List<AnimaleTerrestre>`
- un argomento di tipo `List<? extends AnimaleTerrestre>` è compatibile con tutte le liste di "cose che estendono *AnimaleTerrestre*" (in rosso nel disegno)
OVVERO
liste di "almeno" AnimaleTerrestre

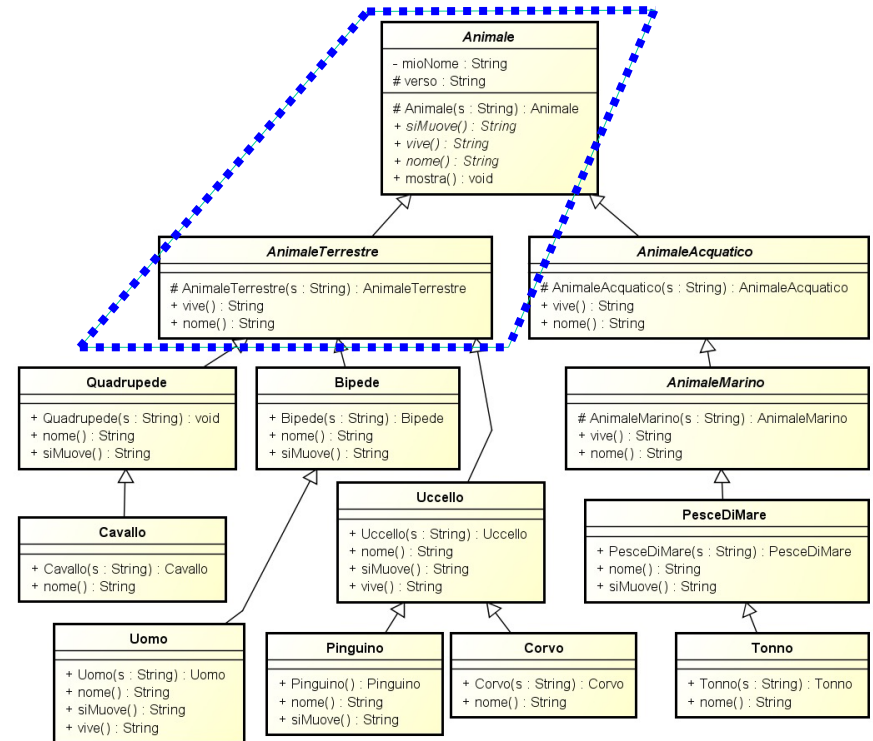


INTERPRETAZIONE (2)

Le notazioni **<? extends T>** e **<? super T>** si possono interpretare come **upper bound** e **lower bound** dei "tipi accettabili" in una collezione (rispetto a una tassonomia)

ESEMPIO:

- un argomento di tipo `List<AnimaleTerrestre>` è compatibile solo con altre `List<AnimaleTerrestre>`
- un argomento di tipo `List<? super AnimaleTerrestre>` è compatibile con tutte le liste di *"cose che stanno sopra AnimaleTerrestre"* (in blu nel disegno)
OVVERO
liste di "al più" AnimaleTerrestre



LAVORARE SU UNA COLLEZIONE "ASSOLUTAMENTE GENERICA"

*A volte, però, **upper bound e lower bound** ancora non bastano.*

Si consideri ad esempio il caso di **una funzione che stampi tutti gli elementi di una generica collezione.**

- una tale funzione dovrebbe operare idealmente ***su qualunque tipo di collezione, senza restrizioni: come esprimerla?***
- **si potrebbe pensare di usare come argomento il tipo `Collection<Object>`, ma sarebbe sbagliato**, perché esso sarebbe compatibile solo con altre collezioni di *Object* (e null'altro!) – mentre interessa la compatibilità con *collezioni di qualunque tipo*.

LAVORARE SU UNA COLLEZIONE "ASSOLUTAMENTE GENERICA"

La risposta è che serve un tipo che rappresenti una *collezione di qualunque cosa*:

`Collection<? extends Object>`

più brevemente abbreviato come `Collection<?>`

Il tipo `Collection<?>` rappresenta l'idea di *collezione di oggetti di tipo sconosciuto*

- come tale, è compatibile con qualunque collezione effettivamente passata come argomento..
- *..ma per lo stesso motivo non consente alcuna modifica al contenuto della collezione* in quanto *non si possono fare ipotesi sul tipo degli elementi contenuti*.
 - l'unica cosa che si potrà dire è che conterrà degli Object.

ESEMPIO

La prima idea (sbagliata):

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

NO, perché il tipo `Collection<Object>` esprime l'idea che la funzione accetta *collezioni di Object e null'altro.*

La versione corretta:

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

OK, perché il tipo `Collection<?>` esprime l'idea che la funzione accetta *collezioni di ogni tipo* (su cui non si possono fare ipotesi, poiché resta sconosciuto)

Nel caso del nostro stack:

```
public static void  
emptyAndPrintStack(MyStack<?> stack) {  
    while(!stack.isEmpty()) {  
        System.out.println(stack.pop());  
    }  
}
```

OK: non importa conoscere il tipo.. perché le pop si possono fare comunque, e println stampa già qualunque Object ☺

COVARIANZA, CONTROVARIANZA, BIVARIANZA

Questi nuovi tipi varianti si dicono, rispettivamente:

<? super T>

tipo **controvariante** rispetto a T

- si usa per i tipi-collezione in cui si devono *inserire, aggiungere, "scrivere"* elementi di tipo (almeno) T

<? extends T>

tipo **covariante** rispetto a T

- si usa per i tipi-collezione da cui si devono *estrarre, togliere, "leggere"* elementi di tipo (almeno) T

<?>

tipo **bivariante** rispetto a T

- si usa per i tipi-collezione sui cui elementi *non si devono fare ipotesi (tipo sconosciuto)*: in tali collezioni non si possono né scrivere, né leggere elementi di tipo T.
- è uno shortcut per **<? extends Object>**
- ergo, l'unica certezza è che conterrà degli **Object**.. 😊

ESEMPIO DALLA JCF: UN ESTRATTO DALLA CLASSE Collections

```
interface Comparable<T> {  
    boolean isGreaterThan(T element);  
}
```

```
interface Comparator<T>{  
    int compare(T element1, T element2);  
}
```

Possibile scrittura equivalente (ma più prolissa):

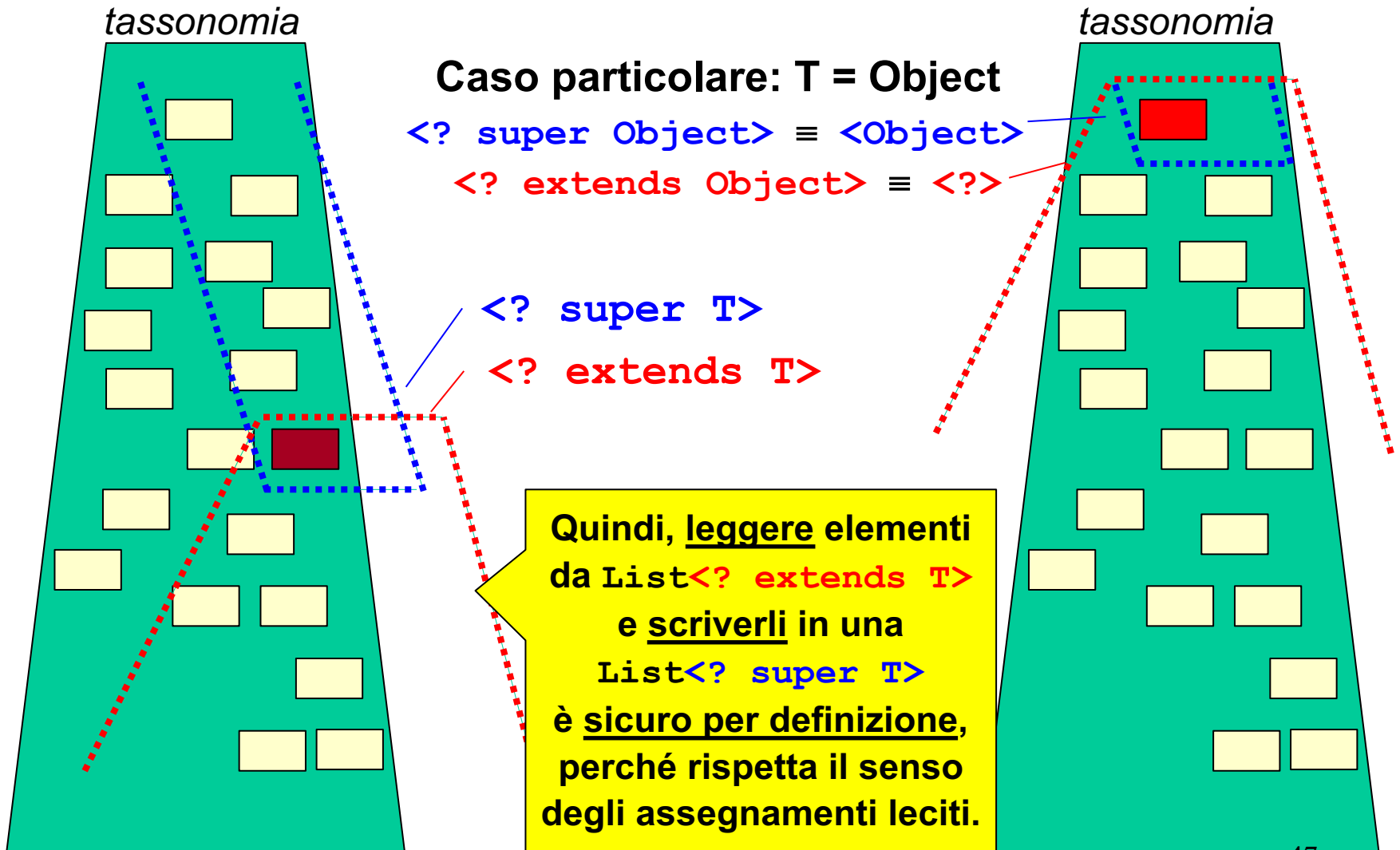
```
void <E, T extends E> fill(List<E> list, T elem)
```

```
<T> void fill(List<? super T> list, T elem);
```

```
<T> void copy(List<? super T> destination,  
             List<? extends T> source)
```

```
<T> void sort(List<T> list,  
             Comparator<? super T> comp)
```

SCHEMA RIASSUNTIVO (1)



COMPATIBILITÀ FRA TIPI VARIANTI

Per completare lo schema riassuntivo dobbiamo precisare *che compatibilità c'è* fra i vari tipi parametrici varianti.

- OVVERO: che relazione c'è, ad esempio, fra i due tipi `List<? extends Number>` e `List<? extends Double>` ?

La risposta è nel significato stesso di notazione variante:

- per definizione, a un argomento di tipo `List<? extends Number>` si può passare una lista di qualunque cosa estenda `Number`, mentre a un argomento di tipo `List<? extends Double>` si può passare una lista di qualunque cosa estenda `Double`
- poiché `Double` estende `Number`, *il primo tipo di argomento accetta più tipi del secondo, ossia è più generale: ogni argomento valido per `List<? extends Double>` lo è anche per `List<? extends Number>`, ma non viceversa. Quindi il primo tipo è un sottotipo del secondo.*
- Discorso duale per i tipi della forma `List<? super T>`

SCHEMA RIASSUNTIVO (2)

CHI È COMPATIBILE CON CHI ?

Dunque, riassumendo insiemisticamente:

`List<? extends Object>` = { `List<Object>`, `List<Number>`, `List<Integer>`, `List<Double>` } }

`List<? extends Number>` = { `List<Number>`, `List<Integer>`, `List<Double>` } }

`List<? extends Integer>` = { `List<Integer>` } }

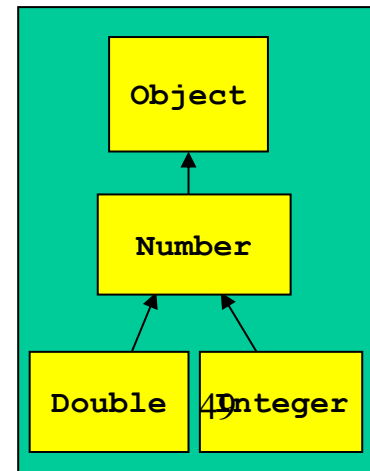
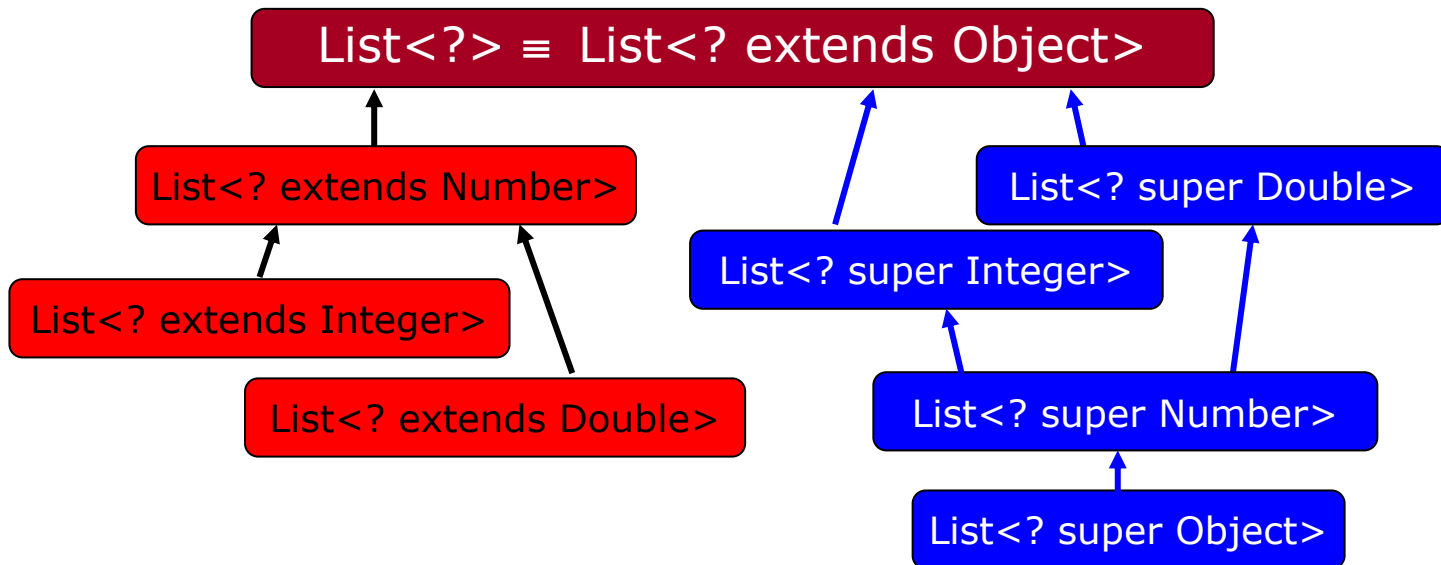
`List<? extends Double>` = { `List<Double>` } }

`List<? super Double>` = { `List<Object>`, `List<Number>`, `List<Double>` } }

`List<? super Integer>` = { `List<Object>`, `List<Number>`, `List<Integer>` } }

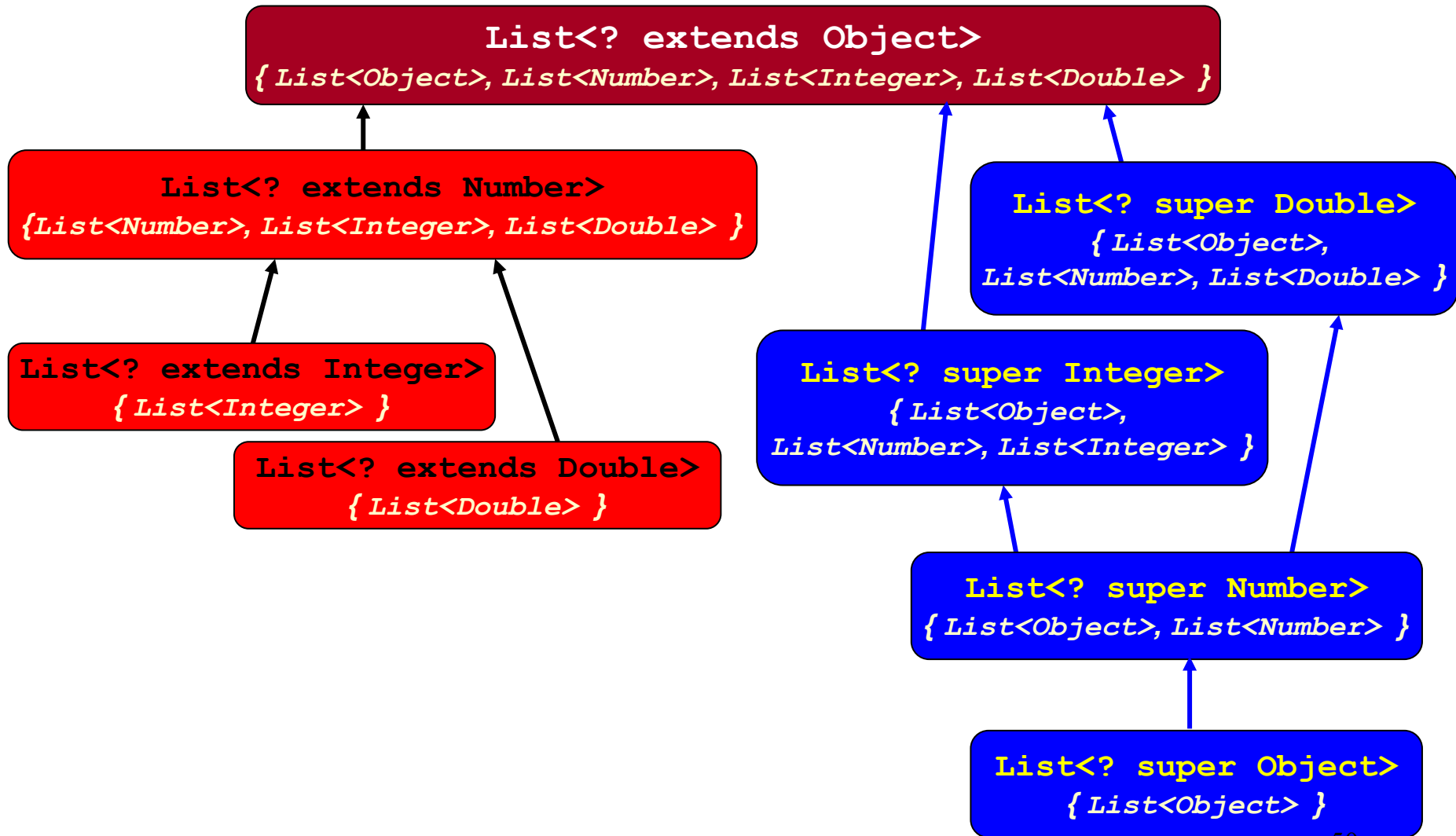
`List<? super Number>` = { `List<Object>`, `List<Number>` } }

`List<? super Object>` = { `List<Object>` } } *[non molto utile]*



SCHEMA RIASSUNTIVO (3)

CHI È COMPATIBILE CON CHI ?



SCHEMA RIASSUNTIVO (4)

CHI È COMPATIBILE CON CHI ?

