Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Varianza & sostituibilità dei componenti
Il principio di sostituzione di Liskov

Corso di Laurea in Ingegneria Informatica

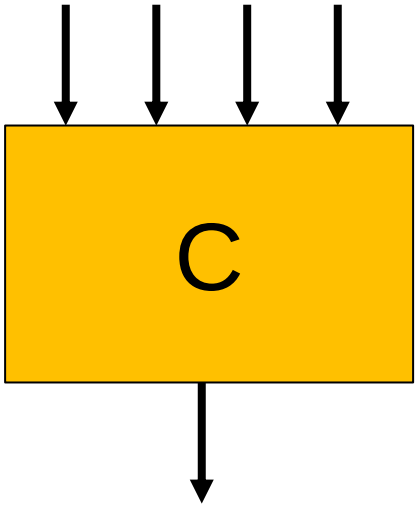
Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

SOSTITUIBILITÀ DEI COMPONENTI

- Supponiamo di avere un componente C, ben funzionante, che accetti certi input e produca un certo output



ESEMPIO HARDWARE

Alimentatore

- Ingresso 220-240V
- Uscita 5-9V

ESEMPIO SOFTWARE

Funzione che abbia:

- Ingresso: un Integer
- Uscita: un Counter

Sotto quali condizioni esso può essere sostituito da un altro componente C' ?

SOSTITUIBILITÀ DEI COMPONENTI

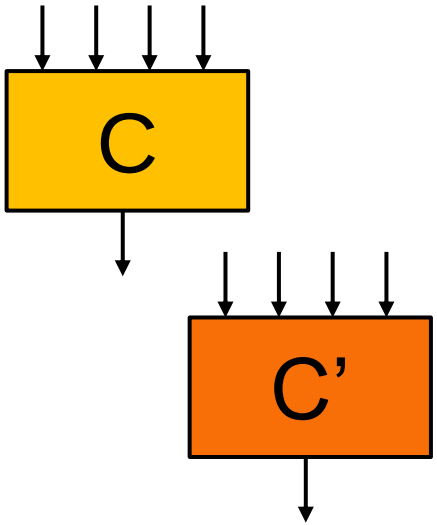
- Principio di Sostituzione di Liskov

  Sia P una proprietà vera su oggetti x di tipo T.

  Allora, P deve valere anche su ogni oggetto y di tipo S in cui S sia sottotipo di T.

Barbara J. Liskov

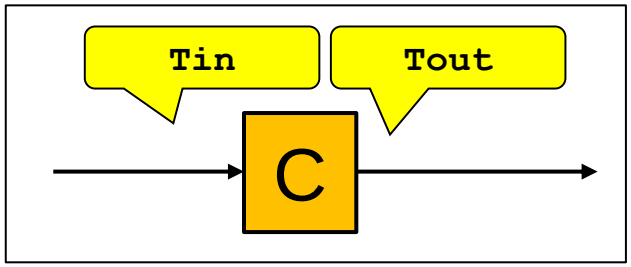
- Ciò implica che il nuovo componente C':
  - in input, abbia specifiche *pari o più ampie* dell'originale
  - in output, abbia specifiche *pari o più stringenti* dell'originale

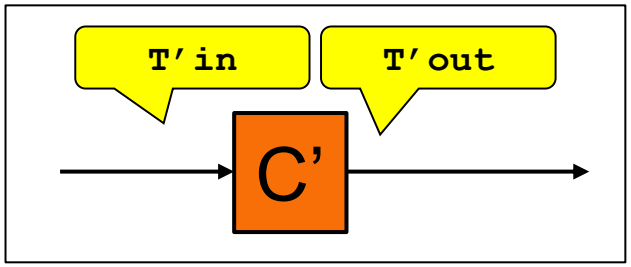


- SLOGAN
  - *Require less, Provide more*
  - *Demand no more, Promise no less*

SOSTITUIBILITÀ DEI COMPONENTI

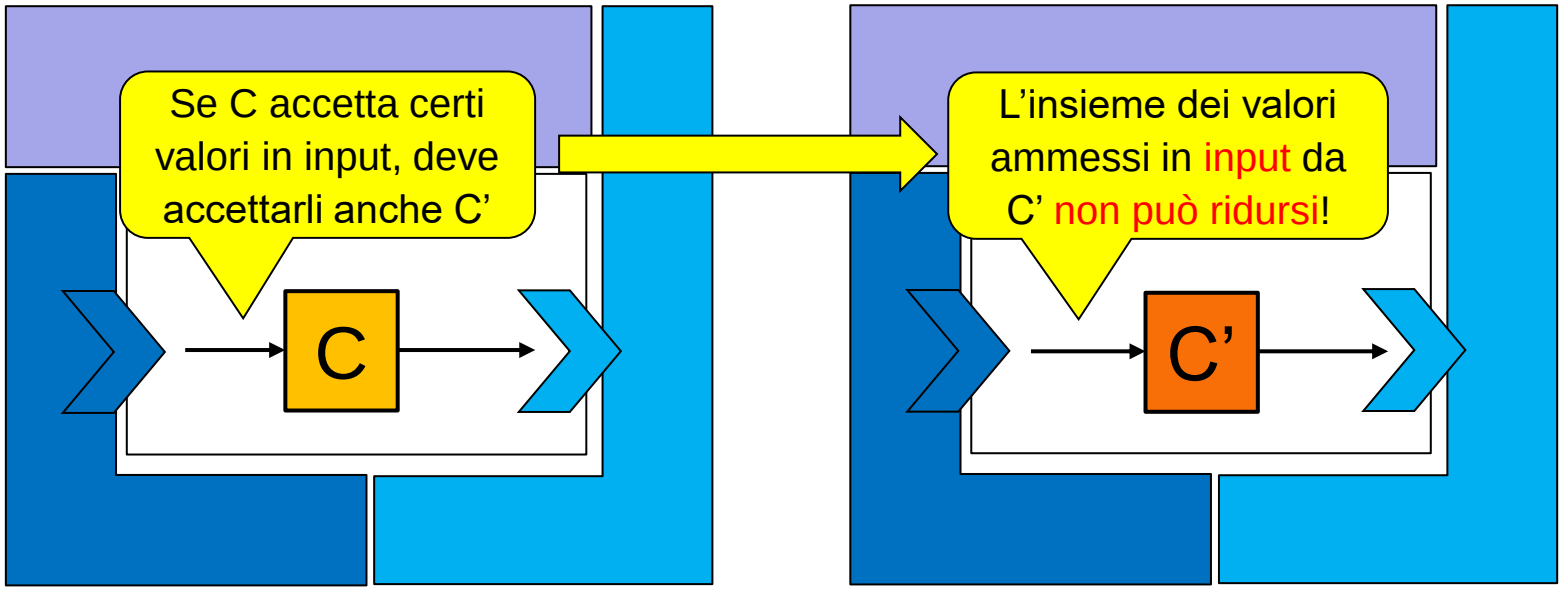
- Interpretazione: affinché C' sia sostituibile al posto di C, C' deve essere usabile *in ogni situazione* in cui si usava C
  - ogni argomento di ingresso valido per C dev'essere valido per C'
    → l'insieme dei tipi accettabili in ingresso a C' dev'essere *pari o più largo* rispetto all'insieme dei tipi accettabili in ingresso a C
  - ogni valore restituito da C' dev'essere un possibile valido valore di ritorno anche per C
    → il tipo restituito in uscita da C' dev'essere *pari o più stretto* del tipo restituito in uscita da C





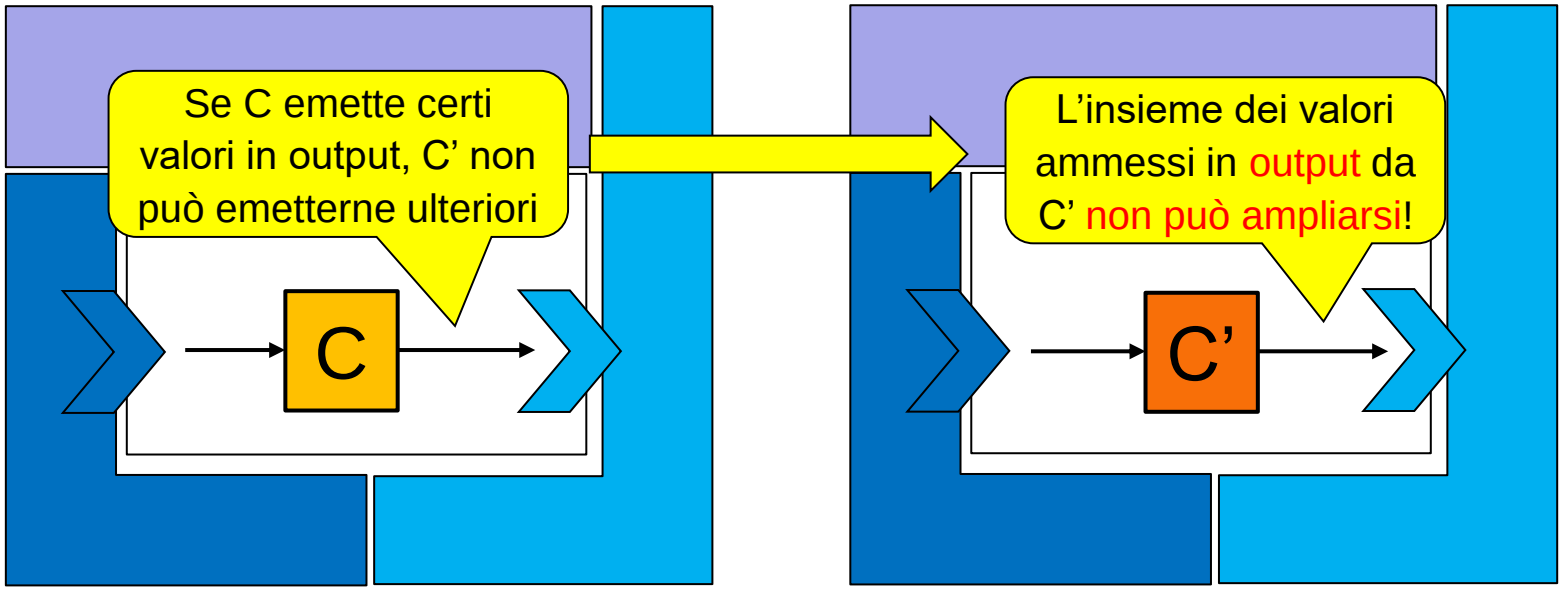
SOSTITUIBILITÀ DEI COMPONENTI

- Interpretazione: affinché C' sia sostituibile al posto di C, C' deve essere usabile *in ogni situazione* in cui si usava C
  - ogni valore di ingresso valido per C dev'essere valido anche per C'
  - ogni valore restituito da C' dev'essere valido anche per C
- Solo così il mondo circostante non si accorgerà di nulla!



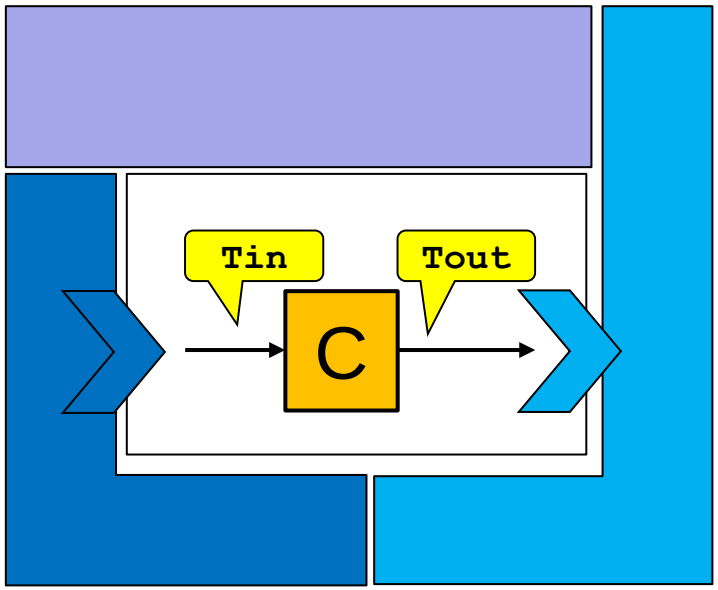
SOSTITUIBILITÀ DEI COMPONENTI

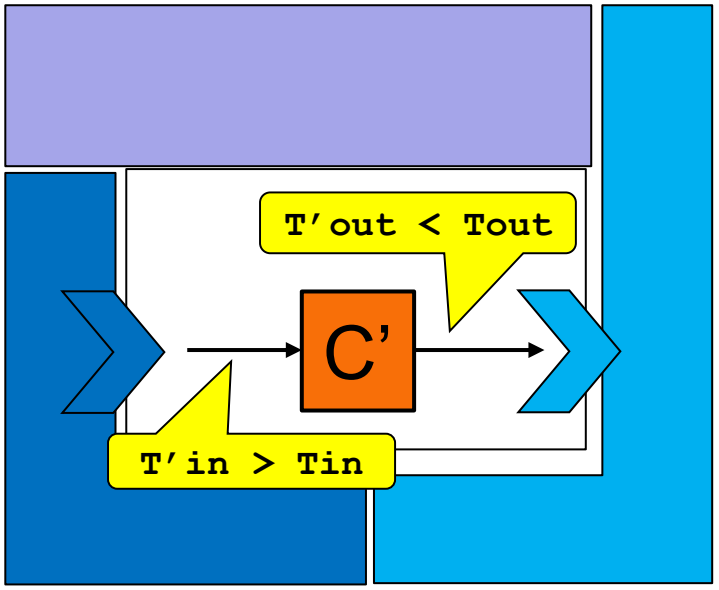
- Interpretazione: affinché C' sia sostituibile al posto di C, C' deve essere usabile *in ogni situazione* in cui si usava C
  - ogni valore di ingresso valido per C dev'essere valido anche per C'
  - ogni valore restituito da C' dev'essere valido anche per C
- Solo così il mondo circostante non si accorgerà di nulla!



SOSTITUIBILITÀ DEI COMPONENTI

- Interpretazione: affinché C' sia sostituibile al posto di C, C' deve essere usabile *in ogni situazione* in cui si usava C
  - ogni valore di ingresso valido per C dev'essere valido anche per C'
  - ogni valore restituito da C' dev'essere valido anche per C
- Solo così il mondo circostante non si accorgerà di nulla!





SOSTITUIBILITÀ DEI COMPONENTI

- Inoltre, C' **non** deve:
  - *introdurre nuove e più stringenti pre-condizioni* (ma può indebolirle)
  - *indebolire le post-condizioni* esistenti (mentre può rafforzarle)
  - introdurre metodi che *modifichino* proprietà osservabili

  Altrimenti, una proprietà vera su oggetti di tipo T cesserebbe di valere su oggetti del sottotipo S, violando il principio di sostituzione di Liskov.

- Ad esempio, C' non può
  - *dichiarare e lanciare nuove eccezioni*, non già previste da C
  - stabilire che un certo invariante di uscita (es. x==y) non sia più vero
  - aggiungere metodi come add, change, etc. se non già previsti

ESEMPIO HARDWARE

- L'alimentatore originale
  - accetta in input tensioni fra 220 e 240 V
  - emette in uscita una tensione fra 5 e 9V
- Qualunque alimentatore sostitutivo deve garantire di fare «almeno» questo, quindi:
  - accettare in input tensioni almeno da 220 a 240 V (o un range più ampio)
  - emettere in uscita una tensione che sia al più nel range 5-9V (ma potrebbe stare in un range più ristretto, ad es. 5-7V)
- MOTIVO: sotto tali condizioni, l'alimentatore sostitutivo *certamente non danneggia* il circuito a cui viene collegato
  - il quale, infatti, riceve valori sicuramente coerenti con quelli che accettava già prima

ALIMENTATORE ORIGINALE
- Ingresso 220-240V
- Uscita 5-9V

Possibili ricambi:
IN 220-240V, OUT 5-**8**V
IN **100**-240V, OUT 5-9V

Non accettabili come ricambi:
IN 220-240V, OUT 5-10V
IN 230-240V, OUT 5-9V

Non funzionerebbe con tensioni di ingresso di 220-230V, che l'originale accettava

Emettendo tensioni di uscita anche di 9-10V, potrebbe danneggiare il circuito a cui viene collegato

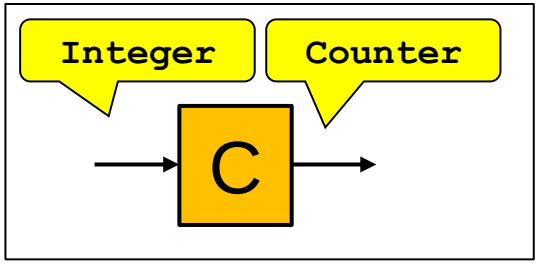
ESEMPIO SOFTWARE

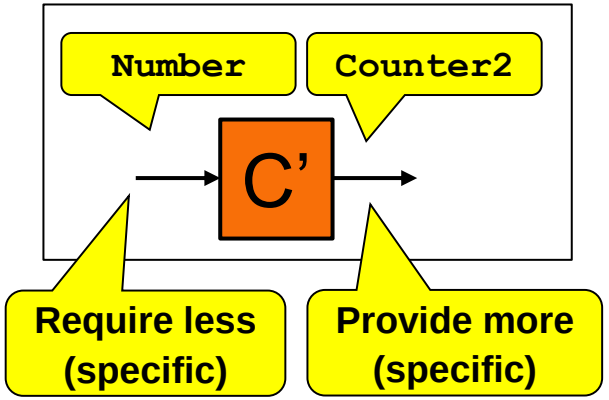
- Il componente originale
  - accetta in input un Integer
  - emette in uscita un Counter
- Qualunque componente sostitutivo lecito deve fare «almeno» questo, quindi:
  - accettare in input almeno gli Integer (magari anche altro)
  - emettere in uscita al più un Counter (magari però Counter più specifici)
- MOTIVO: sotto tali condizioni, il componente sostitutivo *certamente si integra* nel sistema software a cui viene collegato
  - il type system troverà sempre e solo tipi *coerenti* con quelli che gestiva già prima

ESEMPIO SOFTWARE

Funzione che abbia:

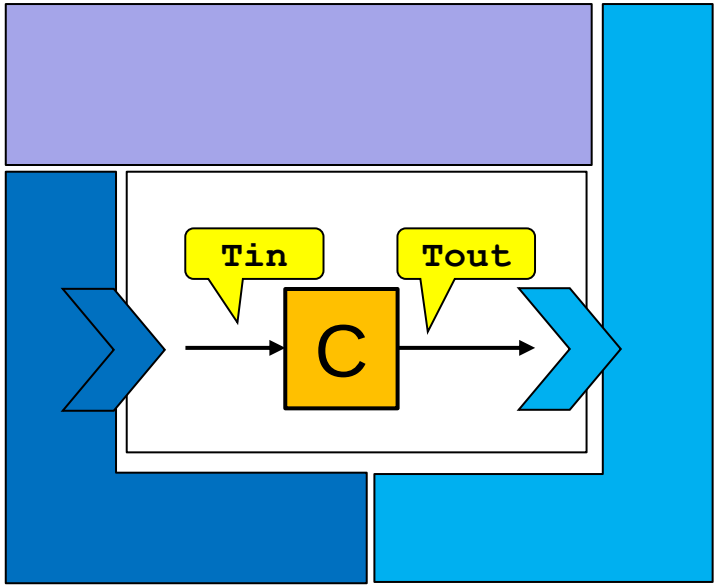
- Ingresso: un Integer
- Uscita: un Counter

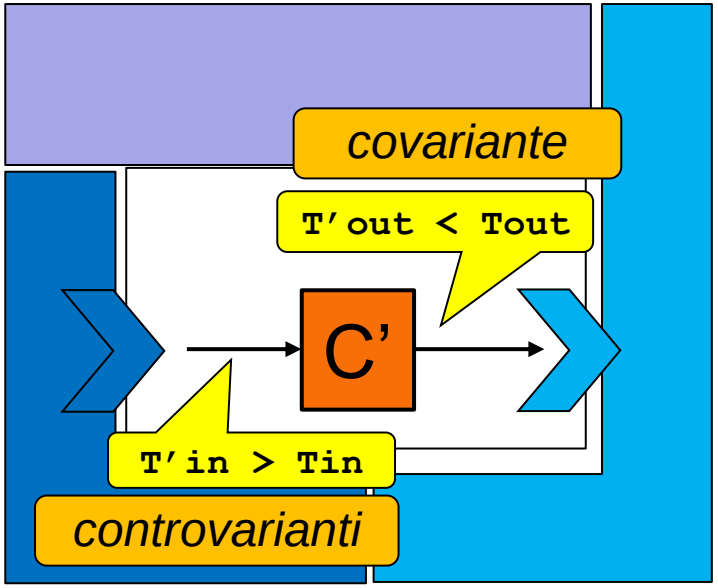




SOSTITUIBILITÀ DEI COMPONENTI

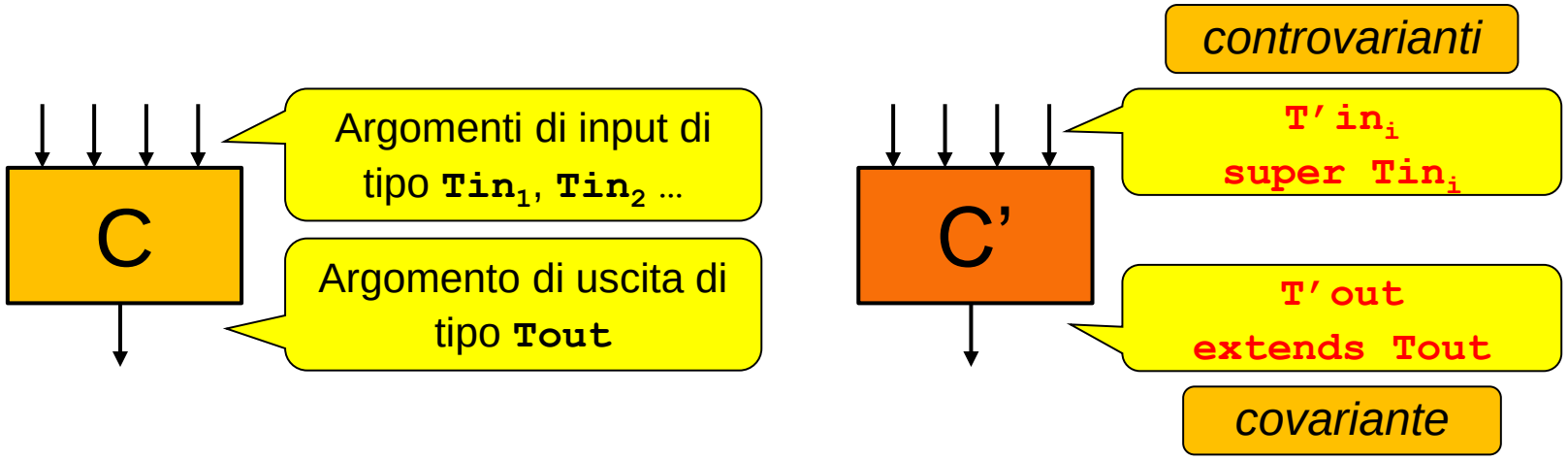
- In termini di vincoli in una gerarchia di ereditarietà, affinché C' sia sostituibile a C, occorre che sia *sottotipo* di C
- A tal fine:
  - i tipi accettabili in ingresso a C' devono essere controvarianti
  - i tipi producibili in uscita da C' devono essere covarianti





SOSTITUIBILITÀ DEI COMPONENTI CONDIZIONE

- Affinché il componente C' sia sostituibile al posto di C, C' deve essere usabile *in ogni situazione* in cui si usava C
  - nessun argomento di ingresso che fosse valido per C può essere *rifiutato* in C'
    → l'insieme dei tipi accettabili in ingresso a C' dev'essere *più largo*
  - C' non può fornire valori di ritorno *esterni* a quelli che avrebbe fornito C
    → il tipo accettabile in uscita da C' dev'essere *più stretto*



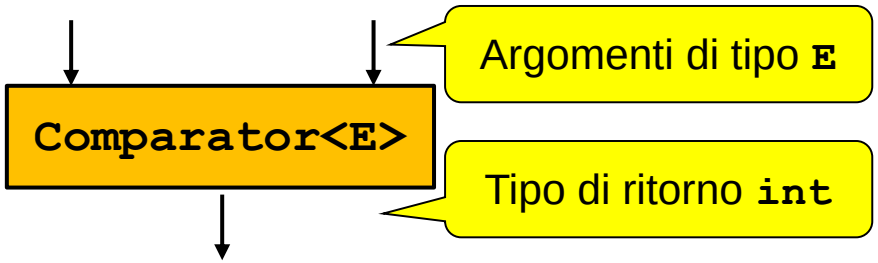
ESEMPIO 1

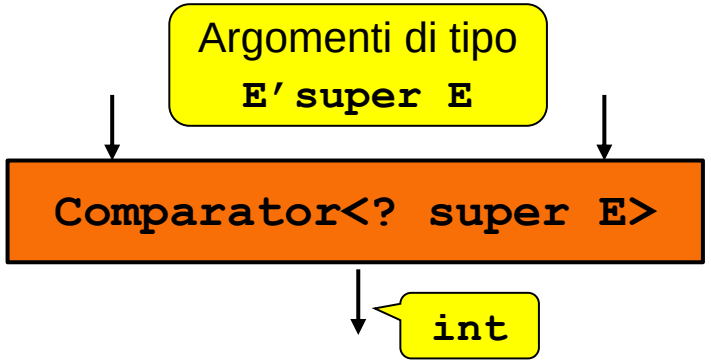
- Si consideri il metodo **`sort`** accetta un **`Comparator`**
  - per ordinare una lista di tipo T, serve un comparatore capace di confrontare due oggetti di tipo T
  - un **`Comparator<T>`** va sicuramente bene, ma non è indispensabile
  - va bene anche un comparatore *che accetti cose più generali di T,* che per ciò stesso sa certamente confrontare, come caso particolare, anche due oggetti di tipo T

ESEMPIO 1

- E infatti, l'argomento **`Comparator`** di **`sort`** per una **`List<E>`** è *controvariante* rispetto a **`E`**

```
void      sort(Comparator<? super E> c)
```





ESEMPIO 2

- Immaginiamo di voler definire un metodo **`filterList`** che filtri gli elementi di una lista secondo un certo filtro:

```
<T> List<T> filterList( List<T> list,
                        Predicate<T> acceptor)
```

```
List<Number> numbers =
       List.of(12, 13, 14, 15, 16.6, 17.7, 18.8);
List<Integer> ints =
       List.of(12, 13, 14, 15, 26, 27, 28);
System.out.println(numbers);
System.out.println(ints) ;
Predicate<Number>  numFilter = N -> N.intValue()%2==0;
Predicate<Integer> intFilter = N -> N%2==0;
System.out.println( filterList(ints,    intFilter) );
System.out.println( filterList(numbers, numFilter) );
```

ESEMPIO 2

- Osserviamo che tutto funzionerebbe bene anche se **`acceptor`** filtrasse *cose più generali* di T:

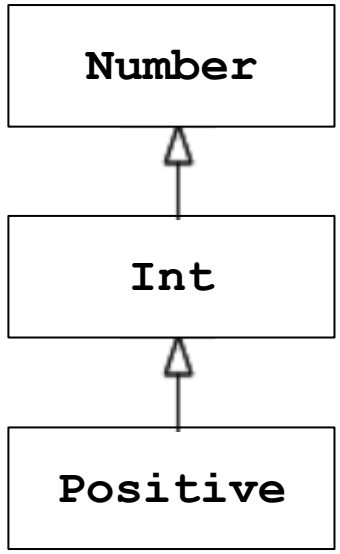
```
<T> List<T> filterList(List<T> list,
                       Predicate<? super T> acceptor)
```

```
List<Number> numbers =
       List.of(12, 13, 14, 15, 16.6, 17.7, 18.8);
List<Integer> ints =
       List.of(12, 13, 14, 15, 26, 27, 28);
System.out.println(numbers);
System.out.println(ints) ;
Predicate<Number>  numFilter = N ->
Predicate<Integer> intFilter = N ->
System.out.println( filterList(ints,    intFilter) );
System.out.println( filterList(numbers, numFilter) );
System.out.println( filterList(ints,    numFilter) );
```

Grazie a ciò, ora si può usare anche **`numFilter`** (anziché **`intFilter`**) per filtrare la lista di numeri

ESEMPIO 3

- Con riferimento alla gerarchia di ereditarietà a lato
- Si consideri il componente **`IntToIntConverter`** sotto illustrato, che espone il metodo **`convert(Int):Int`**



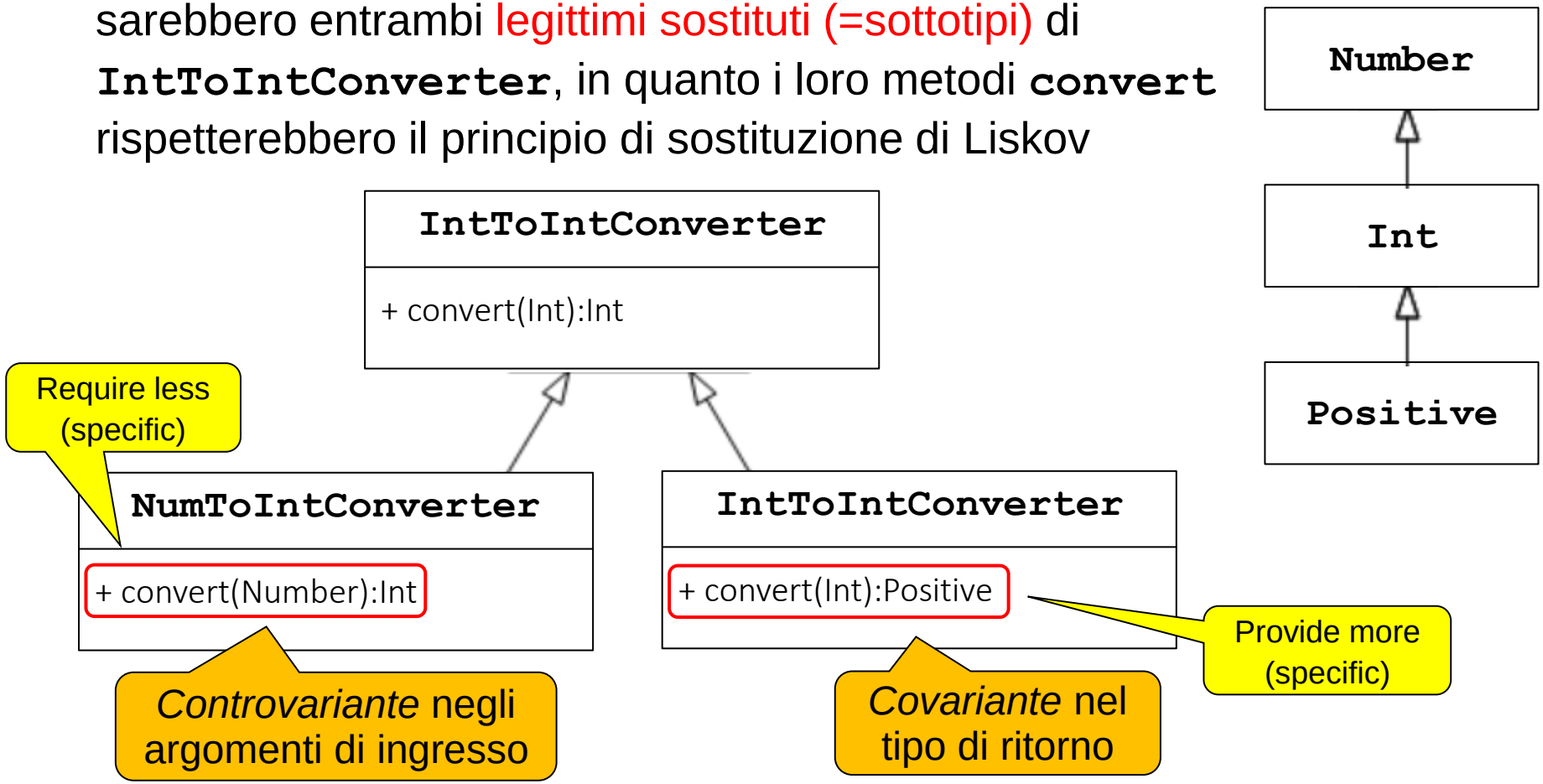
IntToIntConverter
+ convert(Int):Int

- Ci si chiede: due ipotetici componenti più specifici:
  - **`NumToIntConverter`**, il cui metodo **`convert`** accetti un **`Number`** (anziché solo un **`Int`**, quindi *allargando* il range dei valori ammessibili)
  - **`IntToPositiveConverter`**, il cui metodo **`convert`** emetta un **`Positive`** (anziché un generico **`Int`**, quindi *restringendo* l'insieme dei valori restituibili)

sarebbero sostituibili ad esso, senza colpo ferire?

ESEMPIO 3

- Sì: **`NumToIntConverter`** e **`IntToPositiveConverter`** sarebbero entrambi legittimi sostituti (=sottotipi) di **`IntToIntConverter`**, in quanto i loro metodi **`convert`** rispetterebbero il principio di sostituzione di Liskov



ESEMPIO 3: IL CODICE

```
public class Int extends Number {
    private int value;

    public Int(int v) {
        this.value=v;
    }

    public Int() {
        this.value=0;
    }

    public long longValue(){
        return 0L+value;
    }

    public double doubleValue(){
        return 0.0+value;
    }

    public float floatValue(){
        return 0.0F+value;
    }

    public String toString(){
        return ""+value;
    }
}
```

In questa classe ci sono ***due metodi convert*** con diversa signature: si applica il più specifico

```
class IntToIntConverter {
    public Int convert(Int i) {
        System.out.print("This is IntToIntConverter, ");
        return new Int(2*i.intValue());
    }
}

class NumToIntConverter extends IntToIntConverter {
    public Int convert(Number n) {
        System.out.print("This is NumToIntConverter, ");
        return new Int(3*n.intValue());
    }
}

class IntToPosConverter extends IntToIntConverter {
    public Positive convert(Int n) {
        System.out.print("This is IntToPosConverter, ");
        return new Positive(2*Math.abs(n.intValue()));
    }
}
```

Questo metodo *convert* **fa override** dell'omonimo metodo ereditato, perché la signature (return type a parte) è identica

```
public class Positive extends Int {
    public Positive(int v) {
        super(v);
        if (v<0) throw new IllegalArgumentException("Negative value for Positive: " + v);
    }
}
```

ESEMPIO 3: IL CODICE

```
public static void main(String[] args){

    Positive p = new Positive(4);
    Int i = new Int(-3);
    System.out.println(p);
    System.out.println(i);
    //
    Int iRes = new IntToIntConverter().convert(i);
    System.out.println(iRes);
    Positive pRes = new IntToPosConverter().convert(i);
    System.out.println(pRes);
    iRes = new NumToIntConverter().convert(6.28);
    System.out.println(iRes);
    //
    f(new IntToIntConverter(), new Int(-5));
    f(new IntToPosConverter(), new Int(-5));
    f(new NumToIntConverter(), new Int(-5)); // scatta il metodo ereditato da IntToIntConverter
    g(new NumToIntConverter(), new Int(-5)); // scatta il metodo di NumToIntConverter
    // g(new IntToIntConverter(), new Int(-5)); // NO, tipi incompatibili
    // g(new IntToPosConverter(), new Int(-5)); // NO, tipi inco

}

static void f(IntToIntConverter converter, Int n){
    System.out.println(converter.convert(n));
}

static void g(NumToIntConverter converter, Number n){
    System.out.println(converter.convert(n));
}
```

Collaudo di base, senza polimorfismo

Collaudo con principio di Liskov

NumToIntConverter espone **due** convert: quella ereditata, con argomento Int, e quella specifica, con Number.

Scatta la prima!

```
4
-3
This is IntToIntConverter, -6
This is IntToPosConverter, 6
This is NumToIntConverter, 18
This is IntToIntConverter, -10
This is IntToPosConverter, 10
This is IntToIntConverter, -10
This is NumToIntConverter, -15
```

CONTROESEMPIO: GLI ARRAY JAVA

- Gli array Java, notoriamente, sono *type unsafe*
- Infatti, averli definiti inizialmente covarianti
  - perché all'epoca non c'erano i tipi generici né le collection, quindi senza questo non si sarebbe potuto sfruttare il polimorfismo sull'unica «collection» (gli array, appunto) esistente
- ne ha minato per sempre la natura e le proprietà.
- È facile constatare che infatti essi *violano il principio di sostituzione di Liskov*
  - l'estrazione (lettura) di un dato è un metodo (*get*) covariante nel tipo del risultato (e fin qui ok)…
  - …ma la scrittura di un dato è un metodo (*set*) che *dovrebbe* essere invece *controvariante* nell'argomento, e non lo è!

CONTROESEMPIO: GLI ARRAY JAVA

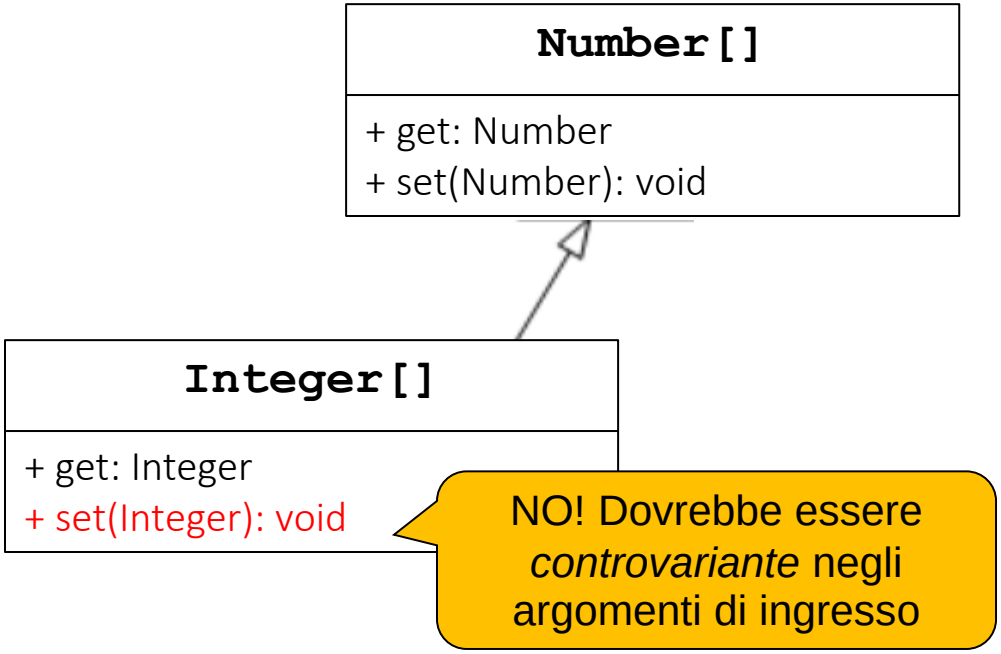
- Per fissare le idee, focalizziamoci sul tipo «array di Number»
  - schematizziamo con due metodi **`get`** e **`set`** l'uso dell'operatore **`[]`** rispettivamente in lettura (es. **`x=v[i]`**) e scrittura (es. **`v[i]=x`**)

`Number[]`
+ get: Number + set(Number): void

- Ora consideriamo il tipo «array di Integer» e chiediamoci: *affermare che esso è un sottotipo di «array di Number» (come fa Java) rispetta il principio di sostituzione di Liskov?*

CONTROESEMPIO: GLI ARRAY JAVA

- Se «array di Integer» è un sottotipo (=legittimo sostituto) di «array di Number», i suoi metodi devono garantire che
  - i tipi accettabili in ingresso a C' siano controvarianti
  - i tipi producibili in uscita da C' siano covarianti



È immediato vedere che non è così

- il metodo *get* è giustamente covariante nel tipo del risultato
- MA il metodo *set*, che dovrebbe essere controvariante, è invece *covariante* anch'esso!!

CONTROESEMPIO: GLI ARRAY JAVA

- Conseguenza: la scrittura in un array può dar luogo a errore («ArrayStoreException») per violazione di coerenza

```java
import java.util.*;

public class MyClass {
    public static void main(String args[]) {
      Integer[] arrayOfInt = new Integer[]{1,2,3,4,5};
      Number[]  arrayOfNum = new Number[5];
      Object[]  arrayOfObj = new Object[5];
      System.out.println("ArrayOfInt = " + Arrays.toString(arrayOfInt));
      filler(arrayOfNum);
      System.out.println("ArrayOfNum = " + Arrays.toString(arrayOfNum));
      filler(arrayOfInt); // ArrayStoreException!
      System.out.println("ArrayOfInt = " + Arrays.toString(arrayOfInt));
    }

    public static void filler(Number[] arr){
        arr[0] = 3.14; arr[1] = 2;
    }
}
```

Result

CPU Time: 0.09 sec(s), Memory: 33080 kilobyte(s)

```
ArrayOfInt = [1, 2, 3, 4, 5]
ArrayOfNum = [3.14, 2, null, null, null]

Exception in thread "main" java.lang.ArrayStoreException: java.lang.Double
    at MyClass.filler(MyClass.java:16)
    at MyClass.main(MyClass.java:11)
```

CONTROESEMPIO: GLI ARRAY JAVA

- DUBBIO: e se fosse il contrario?
- Se fosse «array di Object» a dover essere un sottotipo (=legittimo sostituto) di «array di Number»?

È immediato vedere che non è neppure così

  - il metodo *set* sarebbe ora (giustamente) controvariante nel tipo dell'argomento
  - MA il metodo *get*, che invece dovrebbe essere covariante nel tipo di ritorno, sarebbe *controvariante* anch'esso!!