Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Tipi parametrici varianti

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

TIPI GENERICI NELLA JCF

- Come già discusso, usare il tipo **`Object`** per fare *contenitori generici* causa *seri problemi*
  - equivale ad abolire il controllo di tipo
  - operazioni sintatticamente corrette possono risultare semanticamente errate, causando errori a run-time
- Per questo, la JCF adotta oggi il concetto di *tipo generico*
  - notazione **`<TIPO>`**

```
List<Integer> myList = new ArrayList<>();
myList.add(113);
myList.add("ahahahah"); // compile error
```

E GLI ARRAY..?

- Come noto, gli array sono presenti in Java dalle origini, ossia *ben prima della JCF*
- A differenza della JCF, il linguaggio consente da sempre di definire *array di uno specifico tipo*
  - dai tipi primitivi… `int[] v = new int[4];`
    (non è neppure necessario ricorrere ai tipi-wrapper)
  - …ai tipi-riferimento: `Frazione[] w = new Frazione[7];`
- Sembra dunque che la situazione sia *più chiara e semplice* rispetto a quella della JCF.

ARRAY vs JCF

- In effetti, se si violano vincoli di tipo, il compilatore se ne accorge immediatamente:

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

- L'analogia con la JCF tipizzata è del tutto evidente:

```
List<Integer> myList = new ArrayList<>();
myList.add(113);
myList.add("ahahahah"); // compile error
```

Si può quindi pensare che la JCF abbia semplicemente *avuto dopo* ciò che gli array avevano da subito (in realtà non è proprio così.. !)

ARRAY COME «CAVIE»

- Ha quindi senso *studiare inizialmente sugli array* alcune situazioni particolari che potrebbero verificarsi…
- .. per poi ragionare sulla JCF e comprenderne le scelte

Una nuova domanda:

Array di tipi diversi sono compatibili?

- Il problema si pone perché esiste l'ereditarietà:
  - se B eredita da A, sappiamo che un riferimento di tipo A può legittimamente puntare a un oggetto di tipo B
  - da qui la domanda: può un riferimento ad *array di A* puntare a un oggetto di tipo *array di B ?*

ARRAY COME «CAVIE»

- Ha quindi senso *studiare inizialmente sugli array* alcune situazioni particolari che potrebbero verificarsi…
- .. per poi ragionare sulla JCF e comprenderne le scelte

Una nuova domanda:

Array di tipi diversi sono compatibili?

- In altri termini:
  - ciò che vale «al singolare»…
  - ..ha senso che valga, *in generale*, anche «al plurale» ?

UN PRIMO ESPERIMENTO

- Consideriamo il seguente frammento di codice:

```
Integer[] arrayOfInt = new Integer[4];
arrayOfInt[0] = new Integer(12);
```

Fin qui, tutto tranquillo e tutto normale.

- Poiché però `Integer` deriva da `Object`, è lecito scrivere:

```
Object obj = arrayOfInt[0];
```

- Ergo, sorge spontanea la domanda: *avrebbe senso scrivere un analogo assegnamento «al plurale» ?*

```
Object[] arrayOfObjects = arrayOfInt; // ??
```

UN PRIMO ESPERIMENTO

- Consideriamo il seguente frammento di codice:

```
Integer[] arrayOfI
arrayOfInt[0] = ne
```

Fin qui, tutto tranquillo e t

- Poiché però `Inte`

```
Object obj = arrayOfInt[0];
```

Al singolare, sappiamo già che tutto funziona: grazie al polimorfismo,

- se si stampa `obj` viene stampato *un Integer*
- se si confronta `obj` con un altro oggetto tramite `equals` verrà usato il criterio di confronto *degli Integer*

- Ergo, sorge spontanea la domanda: *avrebbe senso scrivere un analogo assegnamento «al pl*

.. ma qui ..?

```
Object[] arrayOfObjects = arrayOfInt; // ??
```

UN NUOVO PROBLEMA

- Problema: se quella frase è lecita, ora si può accedere *allo stesso array (di Integer)* in due modi:
  - tramite il riferimento `arrayOfInt`
  - *tramite il nuovo riferimento* `arrayOfObjects`

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

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

- Ergo.. cosa succede adesso?

```
arrayOfObjects[1] = "ciao"; // ATTENZIONE!
```

IL PROBLEMA

- Poiché una stringa è un `Object`, l'assegnamento in Java è *formalmente corretto* → la compilazione ha successo
- MA poiché l'array sottostante è di `Integer`, tale assegnamento è in realtà *semanticamente assurdo*
  - un array di `Integer` non può ospitare stringhe!
- Di conseguenza, *a runtime accade il disastro:*

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

Un altro caso di TYPE UNSAFETY

- Forse poi non è così scontato che ciò che vale al singolare valga *in generale* anche al plurale*:* la parte non è il tutto!

UNA PRIMA CONCLUSIONE

- Come già accaduto in passato, anche qui *l'errore di progetto non è stato «smascherato» dal type system*

Gli array non sono "type safe"

- .. ma stavolta è accaduto *nonostante l'array fosse tipizzato!*
  - prima, mancava l'informazione di tipo → l'abbiamo aggiunta
  - qui invece c'era.. ma non è bastato!
- Siamo di fronte a un *problema diverso* dal precedente: un problema *più subdolo*, che ha che fare con le relazioni fra tipi diversi: *quanto in là si può spingere la compatibilità?*

È un problema di *varianza*

IL GRANDE TEMA DELLA VARIANZA

- C'è un errore di fondo nel considerare «compatibili in genera-le» array di tipi diversi *anche se i loro tipi «base» sono fra loro compatibili*
  - considerare array di `Object` e array di `Integer` «parenti» *solo perché lo sono i loro tipi base* è molto superficiale
  - ciò che vale «al singolare» *non necessariamente* vale anche «al plurale»
  - d'altronde, anche in matematica (o in filosofia..) *entità* e *insiemi di entità* sono *concetti diversi,* con *proprietà diverse*
- Questo problema si manifesta con *qualunque collection*

  → Bisogna guardarci in profondità!

VARIANZA: UN NUOVO ESPERIMENTO (parte 1)

- Ricordando che `Number` è la classe base di `Integer`, `Double`, etc., consideriamo il seguente codice:

```
Number[] numbers = { 3, 5, -2.1, 6.28, 2.0 };
Integer[] ints = { 2, 1, -5, 0 };
```

Il primo array contiene istanze di `Integer` e `Double`,
il secondo ovviamente solo di `Integer`

- Ora definiamo questa funzione di stampa ausiliaria:

```
public static void print(Number[] v){
  for (Number n : v) System.out.println(n);
  System.out.println("----------");
}
```

Funzionerà anche per l'array ***`ints`****?*

VARIANZA: UN NUOVO ESPERIMENTO (parte 1)

- In effetti, il programma si compila e gira:

```
Number[] numbers = { 3, 5, -2.1, 6.28, 2.0 };
Integer[] ints = { 2, 1, -5, 0 };
print(numbers);
print(ints);
```

```
3
5
-2.1
6.28
2.0
```

```
2
1
-5
0
```

- Dunque, una funzione che ***legge*** da un `Number[]` sembra **poter** lavorare correttamente anche con un `Integer[]`

VARIANZA: UN NUOVO ESPERIMENTO (parte 2)

- Consideriamo ora invece una funzione che modifichi una cella di un array di `Number` ricevuto come argomento:

```
public static void fillFirst(Number[] v) {
  v[0] = 1.11; // inserisce un Double
}
```

Di nuovo, chiediamoci: *funzionerà anche per l'array* ***ints****?*

```
Number[] numbers = { 3, 5, -2.1, 6.28, 2.0 };
Integer[] ints = { 2, 1, -5, 0 };
fillFirst(numbers); // { 1.11, 5, -2.1, 6.28, 2.0 }
fillFirst(ints); // BOOM! ESPLOSIONE A RUN TIME
```

```
java.lang.ArrayStoreException thrown: java.lang.Double
        at fill (#10:1)
```

VARIANZA: UN NUOVO ESPERIMENTO (parte 2)

- Sebbene sia passata la compilazione, a runtime è esploso

```
public static void fillFirst(Number[] v) {
  v[0] = 1.11; // inserisce un Double
}
```

  - passando un array di `Integer` in quanto «parente» dell'array di `Number`, abbiamo in realtà *violato un vincolo d'uso*
  - `fillFirst` aveva tutto il diritto di mettere un valore `Double` in un array di `Number`
  - ma ovviamente non si può metterlo in un array di `Integer`!

- Dunque, una funzione che ***scrive*** in un **`Number[]`** **non può** lavorare correttamente con un **`Integer[]`**

ARRAY & VARIANZA: CONCLUSIONE

In Java, con gli array:

- è lecito assegnare un array più specifico a uno più generico

```
Number[] res = ints;
```

- è sicuro passare un array più specifico a una funzione che ne attenda uno più generico *per estrarne* elementi:

```
print(ints);
```

- NON è affatto sicuro, anzi causa sicuramente il disastro, passare un array più specifico a una funzione che ne attenda uno più generico *per scriverci* elementi:

```
fillFirst(ints);
```

COVARIANZA

- Si riassume ciò dicendo che gli array Java sono ***covarianti***, ossia la loro compatibilità di tipo varia *nello stesso senso* di quella dei loro elementi
  - poiché **`Integer`** deriva da / è compatibile con **`Number`**, **`Integer[]`** è compatibile con **`Number[]`**
  - ma non viceversa
- Sebbene sembri logico, *non è una scelta type-safe*
  - a causa di ciò, diviene possibile *scrivere* in un array elementi *del tipo sbagliato*, causando *errore a runtime*
- La covarianza funziona solo *in lettura*, non *in scrittura*
  - ma un array per sua natura non è read-only → DISASTRO
  - *è una scelta figlia del suo tempo*: non c'erano i tipi generici, ergo la covarianza era il solo modo per usare decentemente gli array

UN ESEMPIO AL CONTRARIO

- Modifichiamo la funzione `fillFirst` perché accetti solo array di `Integer`:

```
public static void fillFirst(Integer[] v) {
  v[0] = 8; // inserisce certamente un Integer
}
```

Facciamoci ora la domanda inversa: *funzionerà anche per l'array* ***numbers****?*

```
fillFirst(ints); // { 8, 1, -5, 0 }
fillFirst(numbers); // FUNZIONERÀ?
```

Semanticamente avrebbe senso: è senz'altro sicuro inserire un valore `Integer` in un array di `Number`

UN ESEMPIO AL CONTRARIO

- Modifichiamo la funzione `fillFirst` perché accetti solo array di `Integer`:

```
public static void fillFirst(Integer[] v) {
  v[0] = 8; // inserisce certamente un Integer
}
```

Facciamoci ora la domanda inversa: *funzionerà anche per l'array* ***numbers****?* E invece… *non compila neppure!* ☹

```
fillFirst(ints); // { 8, 1, -5, 0 }
fillFirst(numbers); // E INVECE... NON COMPILA!
```

```
Error:
 incompatible types: java.lang.Number[]
cannot be converted to java.lang.Integer[]
```

CONTROVARIANZA

- È un caso in cui servirebbe *controvarianza,* ossia in cui la compatibilità di tipo «giusta» sarebbe *nel senso contrario* a quella degli elementi
- **MA gli array Java furono pensati covarianti**
  - per questo, seppur sia «semanticamente corretta», il compilatore rifiuta la chiamata
  - MOTIVO: all'inizio, il linguaggio non aveva i tipi parametrici, ergo l'invarianza avrebbe impedito qualunque uso del polimorfismo
- D'altronde, ovviamente, nessuna struttura dati può essere *contemporaneamente* covariante e controvariante
  - sono proprietà opposte!
  - e quindi..?

COVARIANZA, CONTROVARIANZA, INVARIANZA

In definitiva:

- in lettura, la type safety richiede covarianza
- in scrittura, la type safety richiede controvarianza

La situazione:

- negli array Java, la scelta originale fu di farli covarianti
  → *type safety solo in lettura* → *scelta infelice*
- nelle collection successive si è perciò cambiato approccio:
  «safety first» → né covarianti, né controvarianti

Tutta la JCF è progettata col vincolo che
le strutture dati siano ***invarianti***

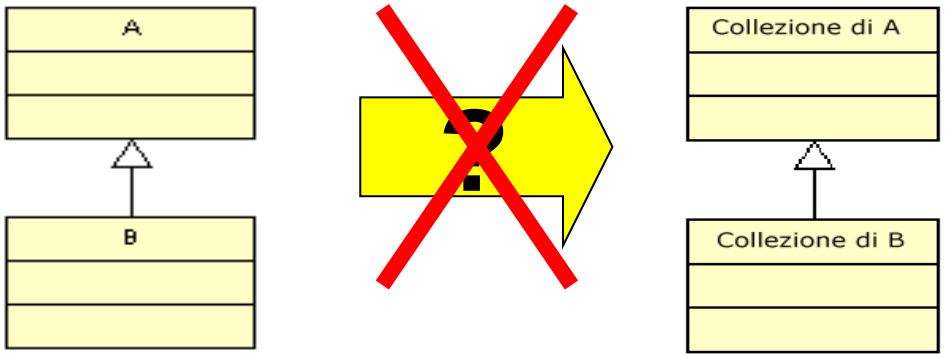
UN APPROCCIO TYPE-SAFE: INVARIANZA

- **Invarianza** = *collezioni di tipi diversi sono incompatibili*
  - può apparire limitativo, ma d'altronde.. la safety non è in vendita!
  - recupereremo per altra via la flessibilità che si serve ☺
- Perciò, *meglio non mischiare* array e JCF
  - gli array sono rimasti covarianti per retrocompatibilità, ma proprio per questo sono *intrinsecamente unsafe*
- Anzi, meglio *non usare proprio gli array* in situazioni in cui ci sia di mezzo la varianza: preferire `List` e collection!

COLLEZIONI INVARIANTI

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.



- Le collezioni Java sono deliberatamente *INVARIANTI*
  - sono *compatibili solo con loro stesse*: nient'altro

ESPERIMENTO CON LISTE

- Con gli array passava, ponendo le premesse del disastro:

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

- Con le liste, invece, *non passa la compilazione*:

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

```
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
```

NIENTE PIÙ PROBLEMI DI VARIANZA

- Riconoscendo l'errore di fondo nella scelta di array covarianti
  - «male minore» per l'epoca, unico modo per poter avere un po' di polimorfismo (verticale) con strutture dati (senza generici)
  - ormai da anni una scelta «fuori dal tempo», error-prone, causa di eccezioni a runtime → esistono alternative migliori
- nelle *collection* si è scelto di basarsi sull'assoluta *invarianza*
  - collezioni di tipi diversi sono *rigorosamente incompatibili anche se* i loro tipi «base» sono compatibili
  - massima sicurezza ☺
  - .. e massima rigidità ☹

- Quanto esattamente abbiamo perso?
- A cosa abbiamo rinunciato?
- C'è modo di attenuare le conseguenze negative..?

UN NUOVO ESPERIMENTO

- Supponiamo di voler scrivere una funzione **`copy`** che *copi tutti gli elementi di una collezione in un'altra.*
- Chiaramente, le due collezioni devono essere omogenee in tipo, quindi ha senso definirla così:

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

- Possibile uso:

```
List<Integer> list1 = …;
List<Integer> list2 = new ArrayList<>();
copy(list1, list2);
```

Qui, **`T = Integer`**

UN NUOVO ESPERIMENTO

- In realtà, questa definizione è *inutilmente limitativa*

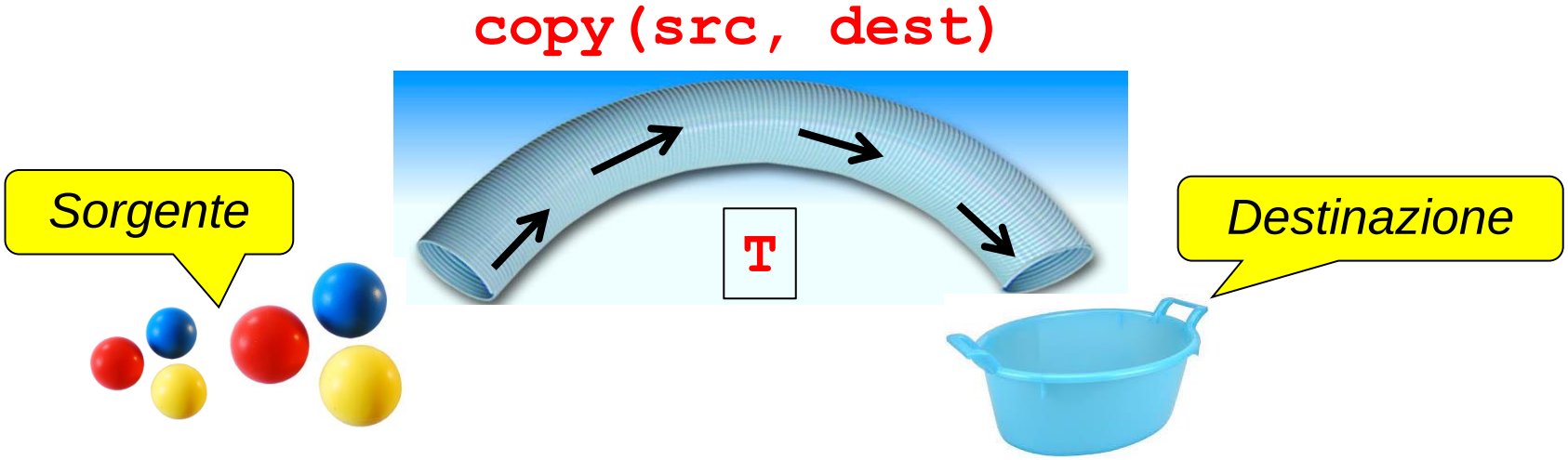
```
public static void copy( Collection<T> src,
                         Collection<T> dest){
   for (T elem : src) dest.add(elem);
}
```

Concettualmente equivalente a **dest[i] = elem = src[i]**

- Infatti:
  - il ciclo di copiatura richiede solo che `dest` accetti `elem`, che è un elemento della sorgente `src`
  - l'assegnamento a `dest` funziona purché `dest` sia di tipo più generale di `elem` (che assumiamo essere di tipo `T`)
  - a sua volta, l'assegnamento a `elem` da `src` è corretto purché quest'ultimo sia di tipo più specifico di `elem` (di tipo `T`)

INTERPRETAZIONE

- Si può immaginare la funzione `copy` come un tubo:

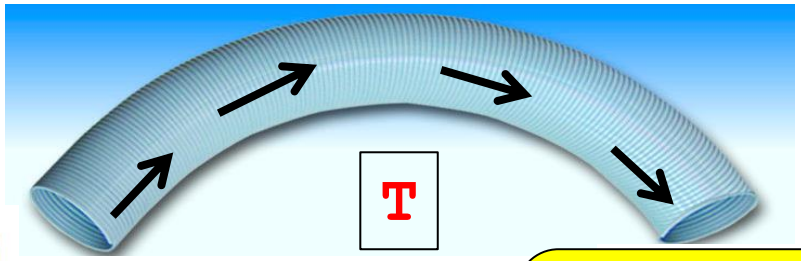


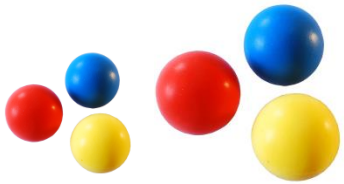
- Perché funzioni, non è realmente indispensabile che `src` e `dest` siano dello stesso *identico* tipo T (diametro del tubo)
- È sufficiente che il tipo sorgente "*entri nel tubo*" e il tipo destinazione possa "*accogliere ciò che esce dal tubo*"

INTERPRETAZIONE

- Si può immaginare la funzione `copy` come un tubo:

`copy(src, dest)`





Più «piccole» = più specifiche
Più «grandi» = più generali
(si riferisce all'insieme)

In realtà, quindi, sono accettabili:

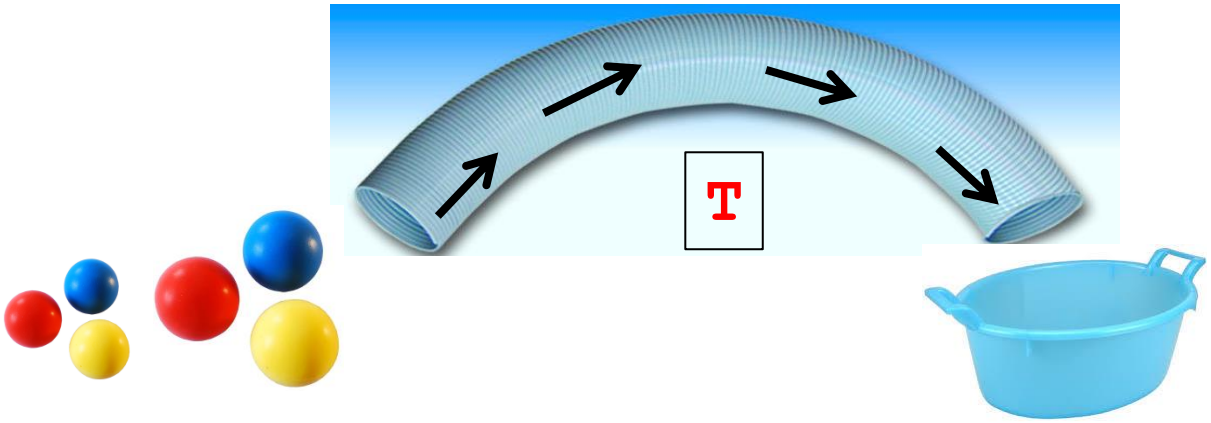
- come sorgente, collezioni di *cose «più piccole o pari a T»*
- come destinazione, collezioni di cose *«pari o più grandi di T»*

Infatti, cose più «piccole» entreranno nel tubo e potranno essere ospitate senza problemi nel cesto finale: ovviamente, non vale il viceversa!

INTERPRETAZIONE

- Si può immaginare la funzione `copy` come un tubo:

`copy(src, dest)`



In realtà, quindi, sono accettabili:

- *Più specifiche* te, collezioni di *cose «più piccole o pari a T»*
- *Più generali* zione, collezioni di cose *«pari o più grandi di T»*

VINCOLO DI ACCETTABILITÀ = VINCOLO DI VARIANZA

Il «grado di accettabilità» dei tipi dipende dalle situazioni:

- le «sorgenti» che ***producono*** oggetti devono essere di tipo *più specifico, o al più pari, a T* → **covarianti** rispetto a T

Producers** should be **covariant

- le «destinazioni» *che* ***consumano*** *(richiedono)* oggetti devono essere di tipo *più generale, o al più pari, a T* → **controvarianti** rispetto a T

Consumers** should be **contravariant

«Producers out, Consumers in»

Riusciamo a ottenere ciò, *senza rinunciare* al principio generale di invarianza delle collection?

SPECIFICARE LA VARIANZA

La **varianza** può essere specificata:

- quando il parametro-tipo viene ***dichiarato***
  → si parla di *declaration-site variance*
  - solo se il componente software (es. classe) è o tutto covariante, o tutto controvariante

C# Scala Kotlin

- quando il parametro-tipo viene ***usato nei vari metodi***
  → si parla di *use-site variance*
  - si specifica che quel dato elemento dev'essere covariante o controvariante in quello specifico caso d'uso
  - coniuga l'invarianza generale con la flessibilità d'uso

Java Scala Kotlin

TIPI PARAMETRICI VARIANTI «WILDCARD» IN JAVA

- Java offre solo ***use-site variance***, tramite la notazione dei *tipi parametrici varianti (wildcard)*
- Serve a *esprimere flessibilità* nei tipi di *argomenti o risultati* di *singoli metodi*
  - obiettivo: dire che *in quello specifico caso, in quello specifico punto e per quello specifico argomento*, è accettabile anche «qualcosa di più/meno» della pura «collezione di T»
- Notazione Java:
  - **`<E extends T>`** per *Producers* «più specifiche» di T
  - **`<E super T>`** per *Consumers* «più generali» di T
  - **`<?>`** per collezioni «qualsiasi» (su cui non si fa alcuna ipotesi)

TIPI PARAMETRICI VARIANTI «WILDCARD» IN JAVA

- Più precisamente:

  Producers **`<E extends T>`** (anonima: **`<? extends T>`**)
  specifica che è accettabile in quel punto un elemento *di tipo* **`E`** *covariante rispetto a* **`T`**, cioè *stia sotto* **`T`** nella tassonomia

  Consumers **`<E super T>`** (anonima: **`<? super T>`**)
  specifica che è accettabile in quel punto un elemento *di tipo* **`E`** *controvariante rispetto a* **`T`**, cioè *stia sopra* **`T`** nella tassonomia

  `<?>`
  specifica che è accettabile in quel punto un argomento di *tipo sconosciuto* (tipo qualunque = varianza totale)

- Acronimo **PECS** = *Producer Extends, Consumer Super*

COVARIANZA & CONTROVARIANZA IN JAVA, C#, SCALA & KOTLIN

Use-site variance

	Java	Scala	Kotlin
≤ T Covarianza	`<? extends T>`	`[_ <: T]`	`<out T>`
≥ T Controvarianza	`<? super T>`	`[_ >: T]`	`<in T>`
Invarianza	`<T>`	`[T]`	`<T>`

Le keyword «**extends**» e «**super**» si riferiscono alla *posizione nella tassonomia* (*sotto / sopra*)

I simboli «**minore**» e «**maggiore**» si riferiscono all' *ampiezza dell'insieme*

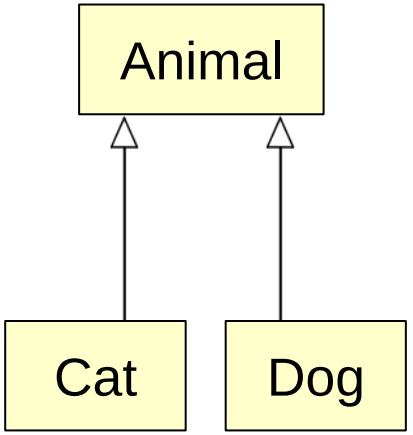
Le keyword «**in**» e «**out**» si riferiscono a *ciò che si può fare (leggere / scrivere) producer / consumer*

NB: Scala e Kotlin supportano anche la *declaration-site* variance
C# supporta in modo limitato solo la *declaration-site* variance

UN ESEMPIO IN JAVA

Al singolare

```
Animal a = new Cat();  // OK
Cat c =  new Animal(); // NO
Cat c = (Cat) new Animal(); // OK (downcast)
```



Al plurale, in lettura — *covariante*

```
Animal rd(List<? extends Animal> list){
  list.add(new Dog());          // NO
  list.add(new Animal());       // NO
  return list.get(0);           // OK
}
```

NO, perché la lista ricevuta, **`list`**, potrebbe essere una **`List<Cat>`** o di altri animali diversi da **`Dog`** e anche dal generico **`Animal`**

Al plurale, in scrittura — *controvariante*

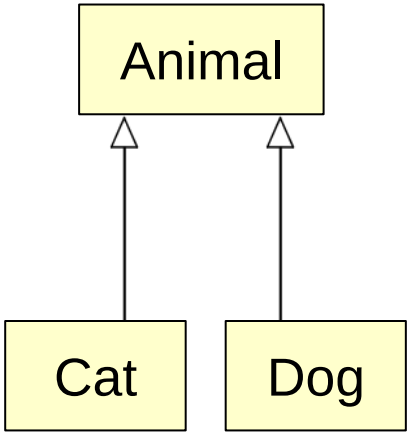
```
void wr(List<? super Animal> list){
 list.add(new Cat());
 list.add(new Dog());
}
```

SÌ, perché la lista ricevuta, **`list`**, può essere solo una **`List<Animal>`** o più su fino a **`List<Object>`**

UN ESEMPIO IN JAVA

Al singolare

```
Animal a = new Cat();  // OK
Cat c =  new Animal(); // NO
Cat c = (Cat) new Animal(); // OK (downcast)
```



Al plurale, in lettura

```
Animal rd(List<? extends Animal> list){
  list.add(new Dog());          // NO
  list.add(new Animal());       // NO
  return list.get(0);           // OK
}
```

Specifica un tipo covariante
Si può leggere, ma *non scrivere*

Al plurale, in scrittura

```
void wr(List<? super Animal> list){
 list.add(new Cat());
 list.add(new Dog());
}
```

Specifica un tipo controvariante
Si può scrivere, ma *non leggere*

RIPRENDENDO LA COPY…

- **Primo passo: allentare il vincolo *sulla sorgente***
  - sono accettabili non solo collezioni di "esattamente T", ma di *cose "più specifiche o pari a T" [provide more]*
  - la notazione wildcard **`<? extends T>`** afferma che lì è accettabile un argomento *di qualunque tipo estenda T*

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

- Possibile uso ampliato:

```
List<ExtendedPhonePlan> list1 = …;
List<PhonePlan> list2 = new ArrayList<>();
copy(list1, list2);
```

RIPRENDENDO LA COPY…

- **Secondo passo: allentare il vincolo *sulla destinazione***
  - sono accettabili non solo collezioni di "esattamente T", ma di *cose "più generali o pari a T" [require less]*
  - la notazione wildcard **`<? super T>`** afferma che **lì** è accettabile un argomento *di qualunque tipo stia sopra T*

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

- Possibile uso ampliato:

```
List<ExtendedPhonePlan> list1 = …;
List<AbstractPhonePlan> list2 = new ArrayList<>();
copy(list1, list2);
```

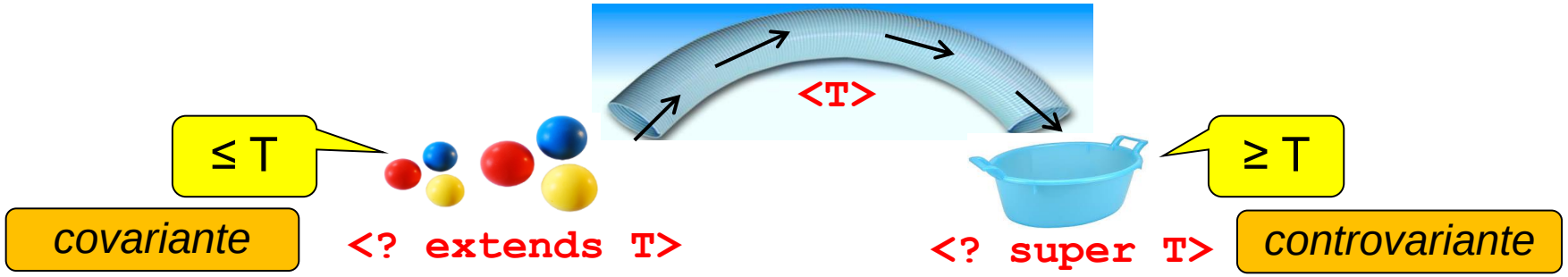
RIPRENDENDO LA COPY…

- **Secondo passo: allentare il vincolo *sulla destinazione***
  - sono accet
    ma di *cose*
  - la notazione
    accettabile un

Sono i due *vincoli minimi* che garantiscono la correttezza delle operazione di assegnamento:

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

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

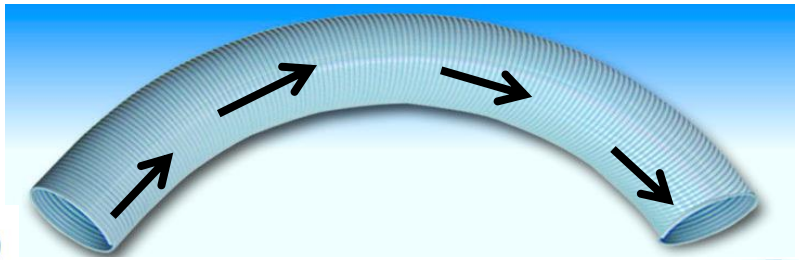


COPY CON WILDCARD

```
copy(src, dest)
```

covariante

controvariante



≤ T

≥ T

SORGENTE

- **elementi di tipo T**
- **o «più specifici» di T**

DESTINAZIONE

- **elementi di tipo T**
- **o «più generali» di T**

`Collection<? extends T>`

T è l'*upper bound* dei tipi accettabili:
o T, o tipi che "*stanno sotto*" T

`Collection<? super T>`

T è il *lower bound* dei tipi accettabili:
o T, o tipi che "*stanno sopra*" T

UPPER & LOWER BOUNDS

- Le notazioni **`<? extends T>`** e **`<? super T>`** si possono interpretare come *upper bound* e *lower bound* dei *tipi accettabili* in un dato punto (rispetto a una tassonomia)

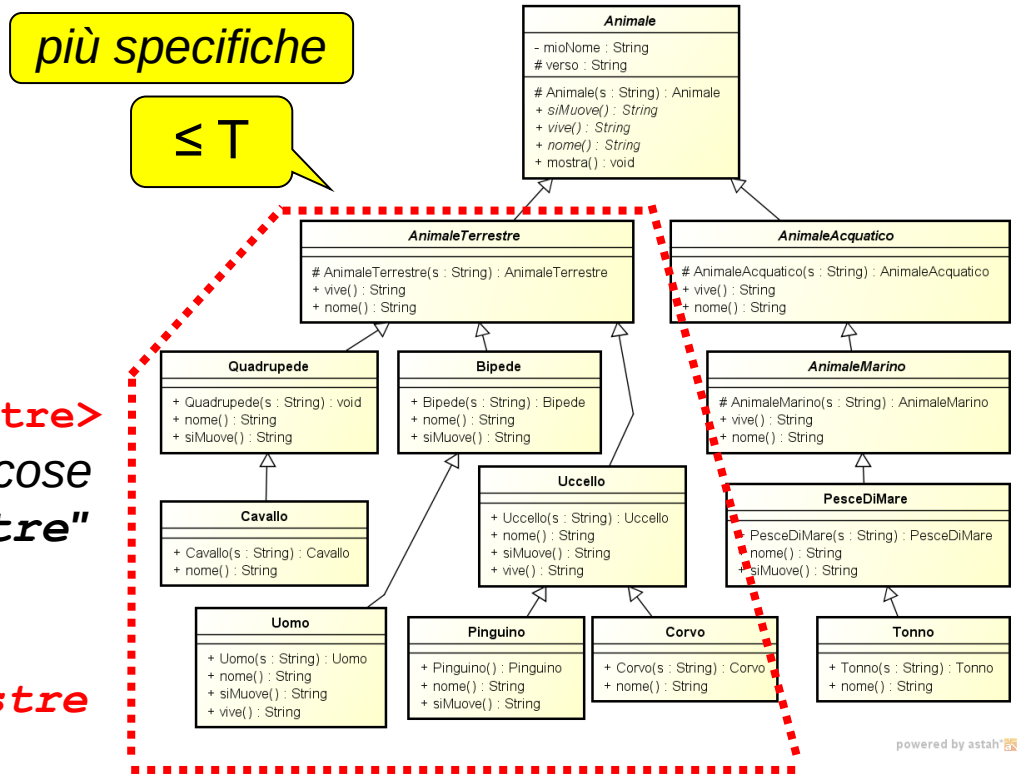
ESEMPI

- 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`

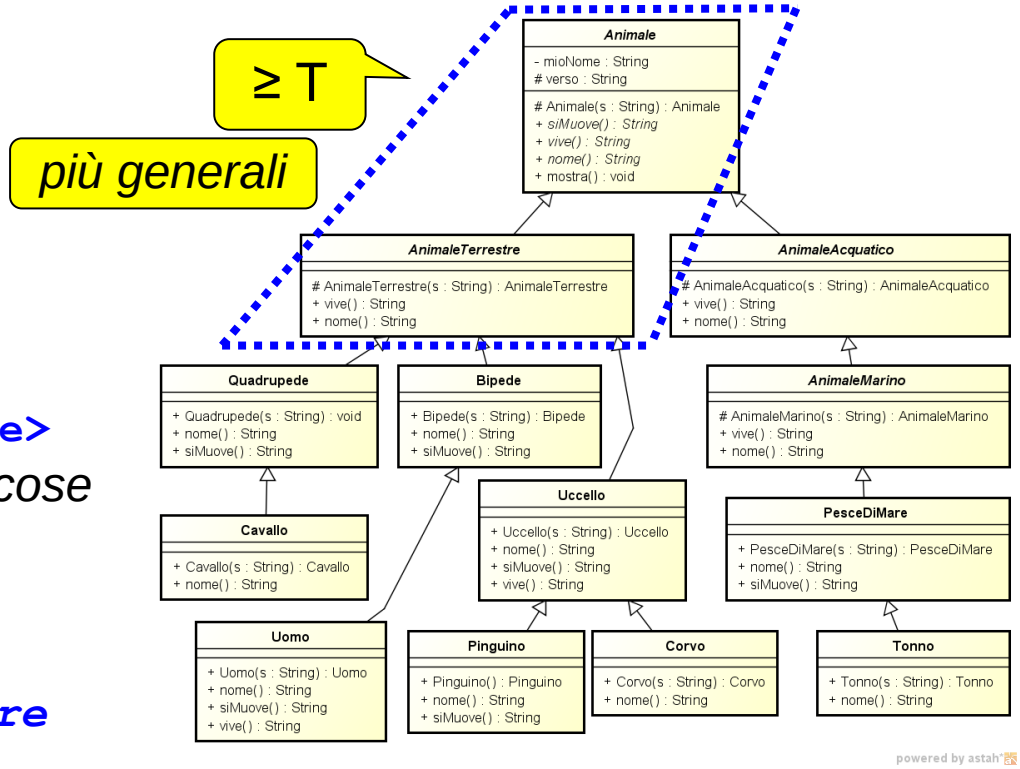


UPPER & LOWER BOUNDS

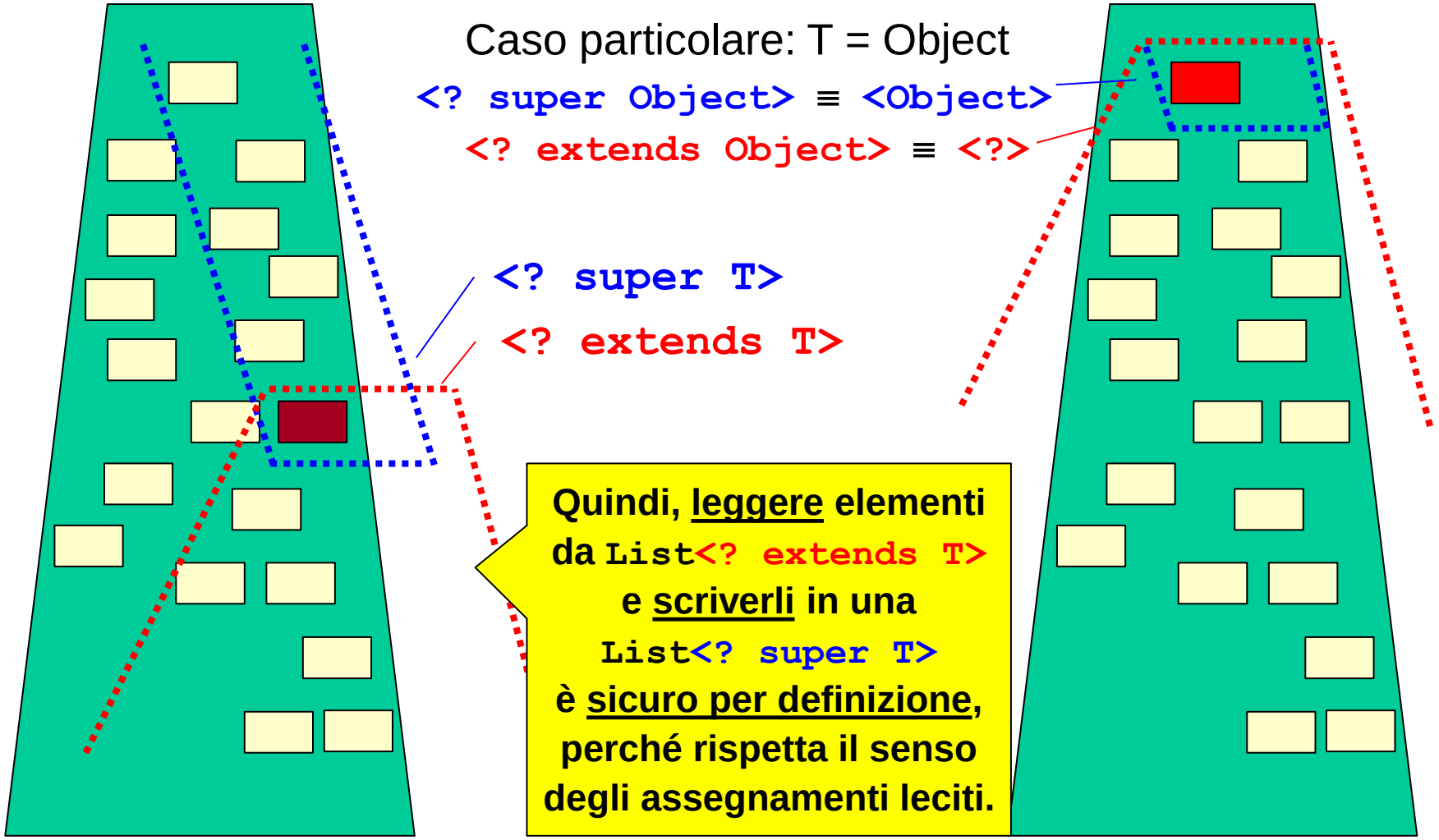
- Le notazioni **`<? extends T>`** e **`<? super T>`** si possono interpretare come *upper bound* e *lower bound* dei *tipi accettabili* in un dato punto (rispetto a una tassonomia)

ESEMPI

- 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 sopra* ***`AnimaleTerrestre`***"
  (in blu nel disegno)
  OVVERO
  liste di "al più" `AnimaleTerrestre`



UPPER & LOWER BOUNDS
SCHEMA RIASSUNTIVO



UNBOUNDED COLLECTION

- Upper / lower bound danno un limite superiore/inferiore al tipi che un certo argomento può accettare
- E se non volessimo dare *alcun limite?*
  - esempio: funzione che *stampi tutti gli elementi* di una *collezione*
  - ipotesi sugli tipi degli elementi: *nessuna*, perché qualunque oggetto di qualunque tipo può essere stampato (`toString`)
- Tale funzione dovrebbe poter operare su *qualsiasi* collection
  - unica ipotesi: che gli elementi siano **`Object`**
  - MA usare come tipo argomento **`Collection<Object>`** *sarebbe sbagliato,* perché sarebbe compatibile *solo con altre collezioni di* ***`Object`*** – e null'altro!
  - il tipo corretto è **`Collection<? extends Object>`**, abbreviabile come **`Collection<?>`**

`Collection<?>`

- Il tipo **`Collection<?>`** rappresenta il tipo «*collezione di oggetti di tipo sconosciuto*»
  - è il tipo «covariante per eccellenza»
- La *notazione* **`<?>`** indica un tipo *unbounded*, ossia che *non ha né upper né lower bound*
  - come tale, è *compatibile con qualunque collezione* effettivamente passata come argomento (accetta qualunque tipo)…
  - ..ma per lo stesso motivo *non consente alcuna modifica al contenuto della collezione* perché non ci sono ipotesi sul tipo degli elementi (se non, ovviamente, che siano… `Object`!)

NOTAZIONI <?> VS <>

ATTENZIONE a *non confondere*:

- la notazione per tipi invarianti **<?>**
- con il *diamond operator* **<>**

- La *notazione unbounded* specifica un tipo
  - indica una qualche "collezione di tipo sconosciuto"
  - in quanto tale, essa rappresenta *un ben preciso tipo*
- Il *diamond operator* invece è una scorciatoia per un tipo *definito altrove*
  - non specifica il tipo *non perché sia sconosciuto*,
  - ma perché *è già stato specificato altrove* (solitamente, poco prima)

RIASSUNTO: JAVA COVARIANZA & CONTROVARIANZA

`<T>` — tipo *invariante*

`<? super T>` — tipo *controvariante* rispetto a T *(lower bound)*

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

`<? extends T>` — tipo *covariante* rispetto a T *(upper bound)*

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

`<?>` — tipo *unbounded*

- 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>`**

COVARIANZA & CONTROVARIANZA IN JAVA, SCALA & KOTLIN

Use-site variance

		Java	Scala	Kotlin
≤ T	Covarianza	`<? extends T>`	`[_ <: T]`	`<out T>`
≥ T	Controvarianza	`<? super T>`	`[_ >: T]`	`<in T>`
	Invarianza	`<T>`	`[T]`	`<T>`

Le keyword «**extends**» e «**super**» si riferiscono alla *posizione nella tassonomia* (*sotto / sopra*)

I simboli «**minore**» e «**maggiore**» si riferiscono all' *ampiezza dell'insieme*

Le keyword «**in**» e «**out**» si riferiscono a *ciò che si può fare (leggere / scrivere) producer / consumer*

NB: Scala e Kotlin supportano anche la *declaration-site* variance
C# supporta in modo limitato solo la *declaration-site* variance

COVARIANZA & CONTROVARIANZA IN JAVA, SCALA & KOTLIN: ESEMPIO

Java

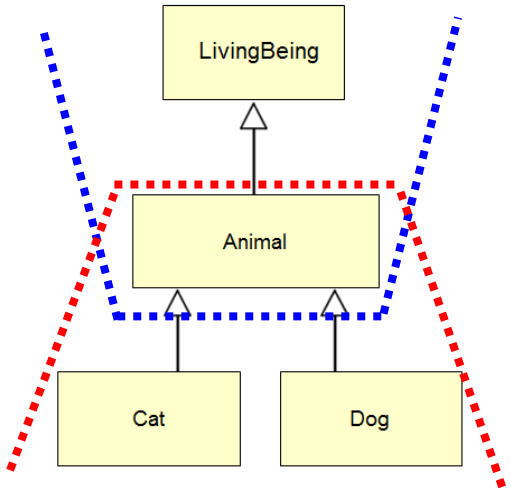
```
Animal rd(List<? extends Animal> list)
{ // list.add(new Dog());     // NO
  // list.add(new Animal()); // NO
  return list.get(0);         // OK
}
void wr(List<? super Animal> list){
 list.add(new Cat());
 list.add(new Dog());
}
```

Java

Kotlin

```
fun rd(list: MutableList<out Animal>): Animal
{ // list.add(Dog());     // NO
  // list.add(Animal()); // NO
  return list.get(0);     // OK
}
fun wr(list: MutableList<in Animal>): Unit {
   list.add(Cat());
   list.add(Dog());
}
```

Kotlin



Scala

```
def rd(list: ArrayDeque[_ <: Animal]): Animal = {
  // list += new Dog();          // NO
  // list += new Animal();       // NO
  return Animal = list(0);       // OK
}
def wr(list: ArrayDeque[_ >: Animal]): Unit = {
   list += new Cat();
   list += new Dog();
}
```

Scala

COVARIANZA & CONTROVARIANZA IN JAVA, SCALA & KOTLIN: ESEMPIO

Java

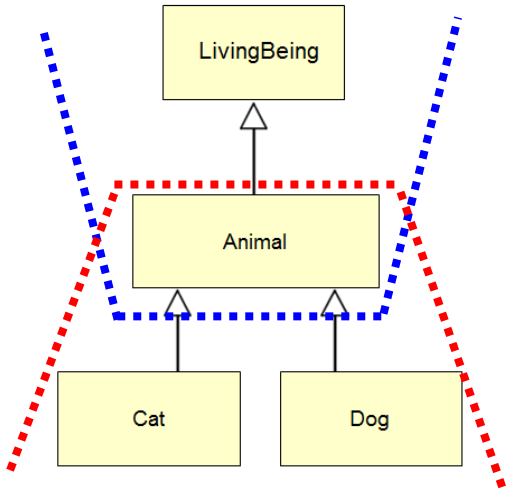
```
Animal rd(List<? extends Animal> l
{ // list.add(new Dog());     // NO
  // list.add(new Animal()); // NO
  return list.get(0);         // OK
}

void wr(List<? super Animal> list){
 list.add(new Cat());
 list.add(new Dog());
}
```

Accetta anche una **List<Dog>**

Infatti, da una tale lista si può benissimo estrarre un **Animal** (o volendo anche un **LivingBeing**)

Accetta anche una **List<LivingBeing>**

Infatti, in una tale lista si potrebbe benissimo scrivere un **Animal**, ma anche un **Dog** o un **Cat**



Scala

```
def rd(list: ArrayDeque[_ <: Animal]): Animal = {
  // list += new Dog();          // NO
  // list += new Animal();       // NO
  return Animal = list(0);       // OK
}
def wr(list: ArrayDeque[_ >: Animal]): Unit = {
   list += new Cat();
   list += new Dog();
}
```

Scala

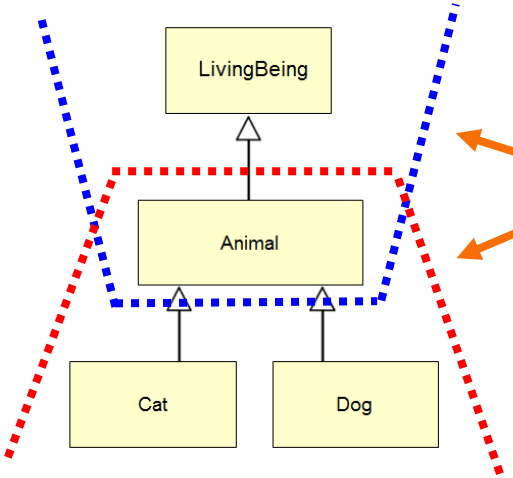
COVARIANZA & CONTROVARIANZA IN JAVA, SCALA & KOTLIN: ESEMPIO

Accetta anche una **`MutableList<Dog>`**

La notazione kotlin, **`out Animal`**, ricorda che da una tale lista si può estrarre un **`Animal`**

Accetta anche **`MutableList<LivingBeing>`**

La notazione kotlin, **`in Animal`**, ricorda che in una tale lista si può scrivere un **`Animal`**



Kotlin

```
rd(list: MutableList<out Animal>): Animal
// list.add(Dog());          // NO
// list.add(Animal());       //
return list.get(0);          //
```

Animal Producer

```
wr(list: MutableList<in Animal>): Unit {
list.add(Cat());
list.add(Dog());
```

Animal Consumer

Scala

```
def rd(list: ArrayDeque[_ <: Animal]): Animal = {
  // list +=
  // list +=
  return Anim
}
def wr(list: ArrayDeque[_ >: Animal]): Unit = {
    list += ne
    list += ne
}
```

La notazione Scala, **`_ <: Animal`**, ricorda la relazione d'ordine fra i tipi

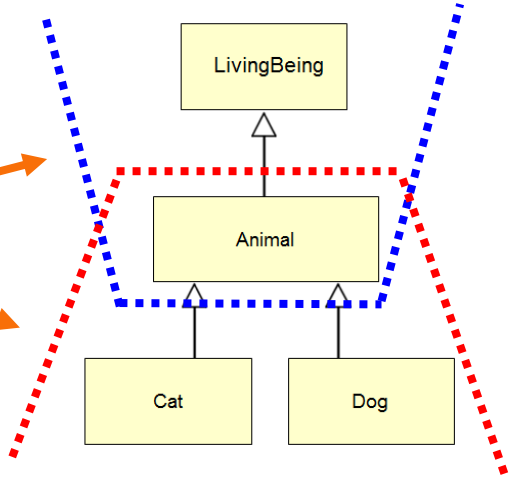
La notazione Scala, **`_ >: Animal`**, ricorda la relazione d'ordine fra i tipi

L'ESEMPIO IN JAVA con Jshell

Java

```
Animal rd(List<? extends Animal> list)
{ // list.add(new Dog());      // NO
  // list.add(new Animal());   // NO
  return list.get(0);          // OK
}

void wr(List<? super Animal> list){
 list.add(new Cat());
 list.add(new Dog());
}
```

Java




```
jshell> LivingBeing a = rd(List.of(new Animal(), new Animal()))
a ==> Animal@cd2dae5
```

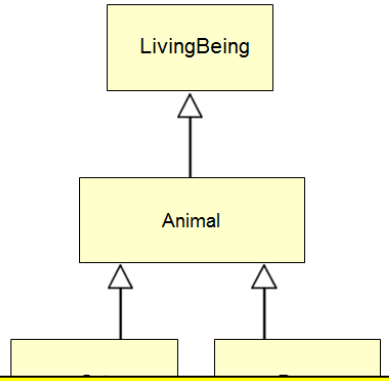
```
jshell> List<Animal> list = new ArrayList<Animal>()
list ==> []

jshell> wr(list)

jshell> list
list ==> [Cat@3b084709, Dog@3224f60b]
```

L'ESEMPIO IN JAVA con Jshell

Java

```
Animal rd(List<? extends Animal> list)
{ // list.add(new Dog());     // NO
  // list.add(new Animal()); // NO
  return list.get(0);         // OK
}

void wr(List<? super Animal> list){
 list.add(new Cat());
 list.add(new Dog());
}
```

Java



rd ha un argomento **covariante**
Accetterebbe una **List<Dog>**
ma non una **List<LivingBeing>**

```
jshell> var a = rd(liBe)
|  Error:
|  incompatible types: java.util.List<LivingBeing> cannot be converted
to java.util.List<? extends Animal>
            var a = rd(liBe);
                       ^--^
```

```
jshell> List<LivingBeing> liBe = new ArrayList<>()
liBe ==> []

jshell> wr(liBe)

jshell> liBe
liBe ==> [Cat@72b6cbcc, Dog@a7e666]
```

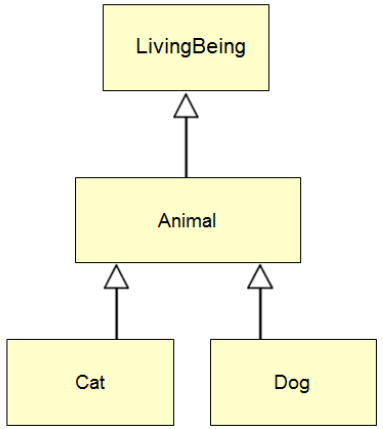
wr ha un argomento **controvariante**
Accetta una **List<LivingBeing>**
ma non accetterebbe **List<Dog>**

L'ESEMPIO IN JAVA con Jshell

Java

```
Animal rd(List<? extends Animal> list)
{ // list.add(new Dog());     // NO
  // list.add(new Animal()); // NO
  return list.get(0);         // OK
}

void wr(List<? super Animal> list){
 list.add(new Cat());
 list.add(new Dog());
}
```

Java




```
jshell> List<LivingBeing> liBe = new
liBe ==> []

jshell> wr(liBe)

jshell> liBe
liBe ==> [Cat@72b6cbcc, Dog@a7e666]
```

wr ha un argomento **controvariante**

Accetta una **List<LivingBeing>**
ma non accetterebbe **List<Dog>**

rd ha un argomento **covariante**

Accetterebbe una **List<Dog>**
ma non una **List<LivingBeing>**

```
jshell> var a = rd(liBe)
|  Error:
|  incompatible types: java.util.List<LivingBeing> cannot be converted
to java.util.List<? extends Animal>
            var a = rd(liBe);
                       ^--^
```

UN ESTRATTO DALLA LIBRERIA JAVA Collections

Java

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

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

Possibile scrittura equivalente (ma più prolissa):

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

Perché se confronta cose più generali di `T`, a maggior ragione è in grado di confrontare cose di tipo `T`

Compatibilità fra tipi varianti

COMPATIBILITÀ FRA TIPI VARIANTI

- Per completare lo schema riassuntivo dobbiamo chiederci *quali compatibilità ci siano fra tipi varianti stessi!*
- OVVERO: che relazione c'è, ad esempio, fra i due tipi Java **`List<? extends Number>`** e **`List<? extends Double>`** ?

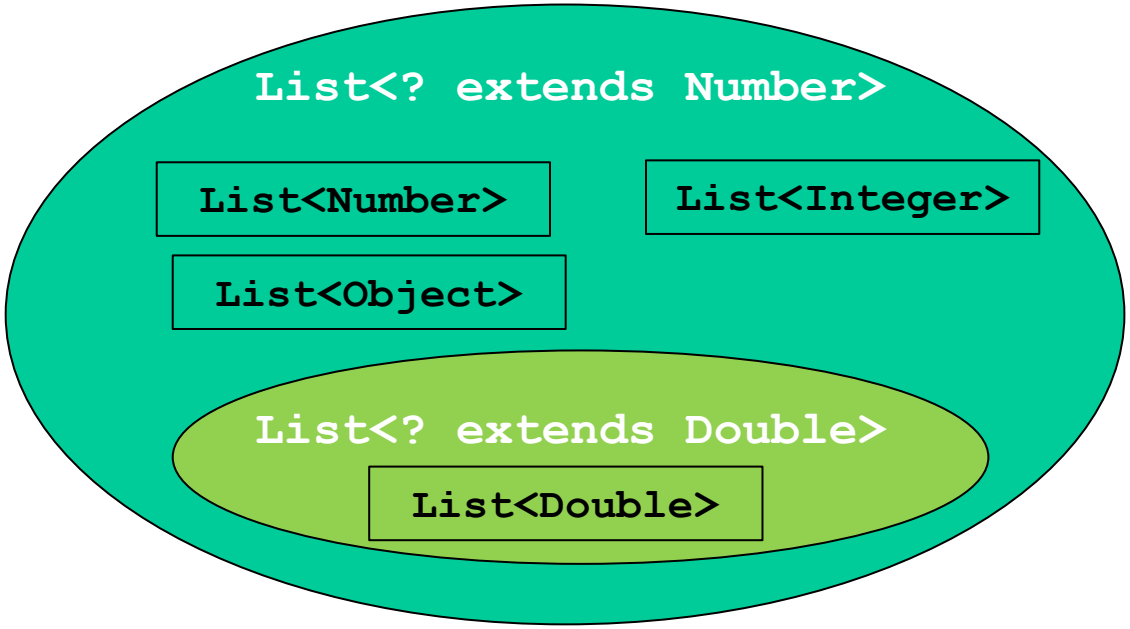
La risposta è nel significato stesso di notazione variante

- A un argomento di tipo **`List<? extends Number>`** si può passare *una lista di qualunque cosa estenda* ***`Number`***
- A un argomento di tipo **`List<? extends Double>`** si può passare una lista di qualunque cosa estenda **`Double`**
- Poiché **`Double`** estende **`Number`**, **`List<? extends Number>`** è più generale: *ogni argomento valido per* ***`List<? extends Double>`*** *lo è anche per* ***`List<? extends Number>`***, ma non viceversa.
- Discorso duale per i tipi della forma **`List<? super T>`**

Java

COMPATIBILITÀ FRA TIPI VARIANTI

- Per completare lo schema riassuntivo dobbiamo chiederci *quali compatibilità ci siano fra tipi varianti stessi*
- OVVERO: che relazione c'è, ad esempio, fra i due tipi **`List<? extends Number>`** e **`List<? extends Double>`** ?



`List<? extends Number>`
`List<Number>`
`List<Integer>`
`List<Double>`

`List<? extends Double>`
`List<Double>`

COMPATIBILITÀ FRA TIPI VARIANTI

- Per completare lo schema riassuntivo dobbiamo chiederci *quali compatibilità ci siano fra tipi varianti stessi*
- OVVERO: che relazione c'è, ad esempio, fra i due tipi **`List<? extends Number>`** e **`List<? extends Double>`** ?

ANALOGIA: disequazioni

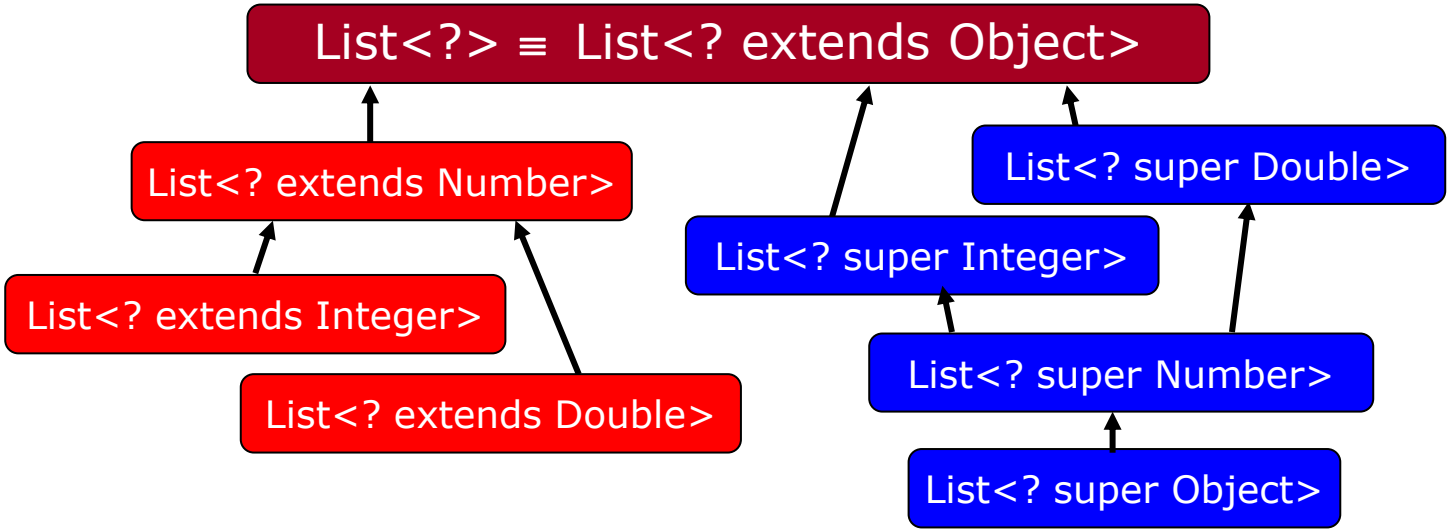
- le notazioni wildcard **`<? extends Number>`** e **`<? super Number>`** sono analoghe a vincoli numerici come **$x \leq$ `Number`** e **$x \geq$ `Number`**
- Quindi, chiedersi che relazione ci sia fra **`List<? extends Double>`** e **`List<? extends Number>`** è un po' come chiedersi che relazione ci sia fra **$x \leq 3$** e **$x \leq 5$** : *uno dei due vincoli ingloba l'altro*
- Vince il vincolo più stringente, cioè quello che *comprende meno valori:* con **`extends`** il più restrittivo è **`List<? extends Double>`**
- Discorso duale per **`<? super T>`** , che è analoga a **$x \geq N$**

SCHEMA RIASSUNTIVO (2)
CHI È COMPATIBILE CON CHI ?

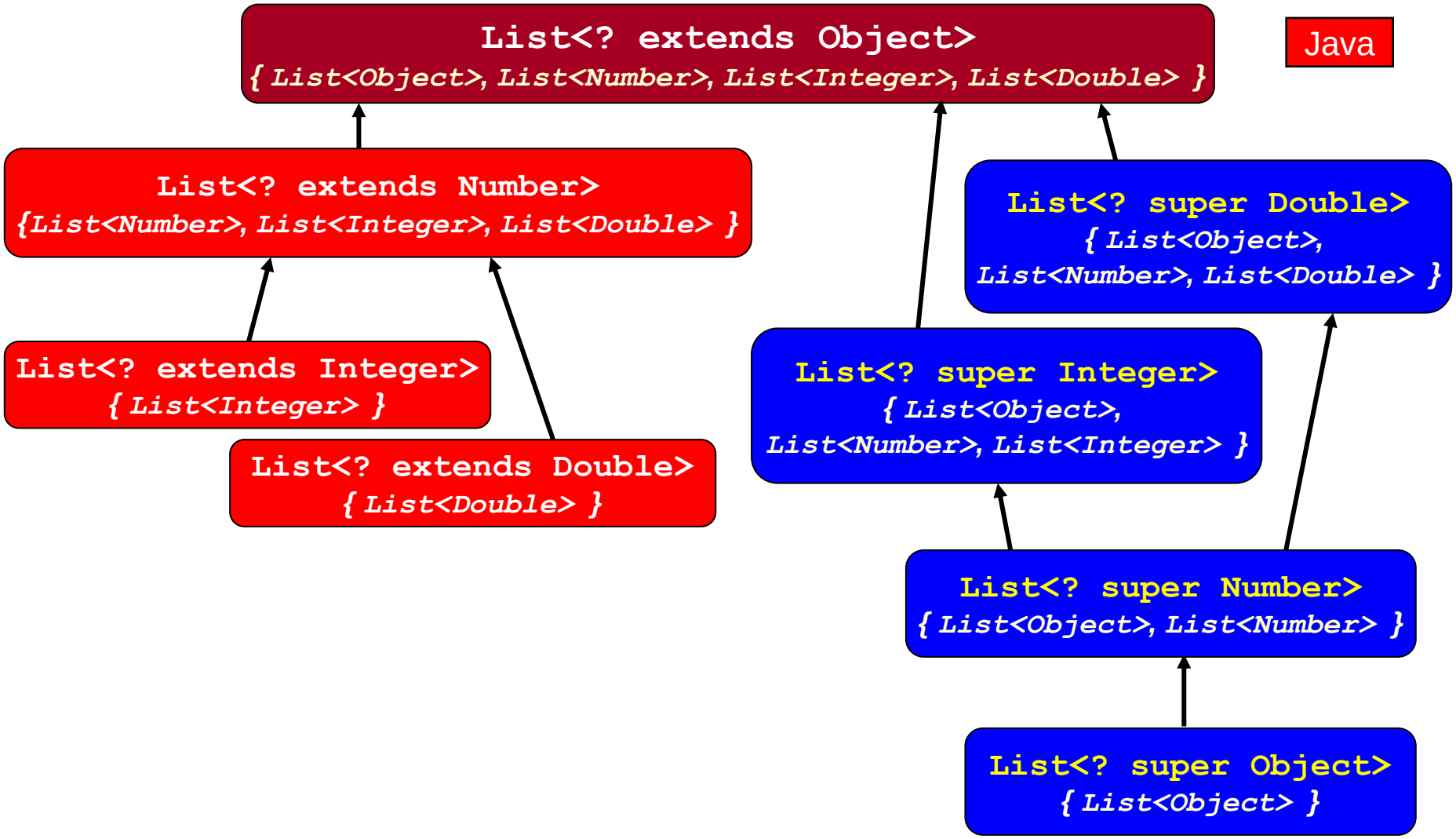
- Completando il ragionamento insiemistico:

Java

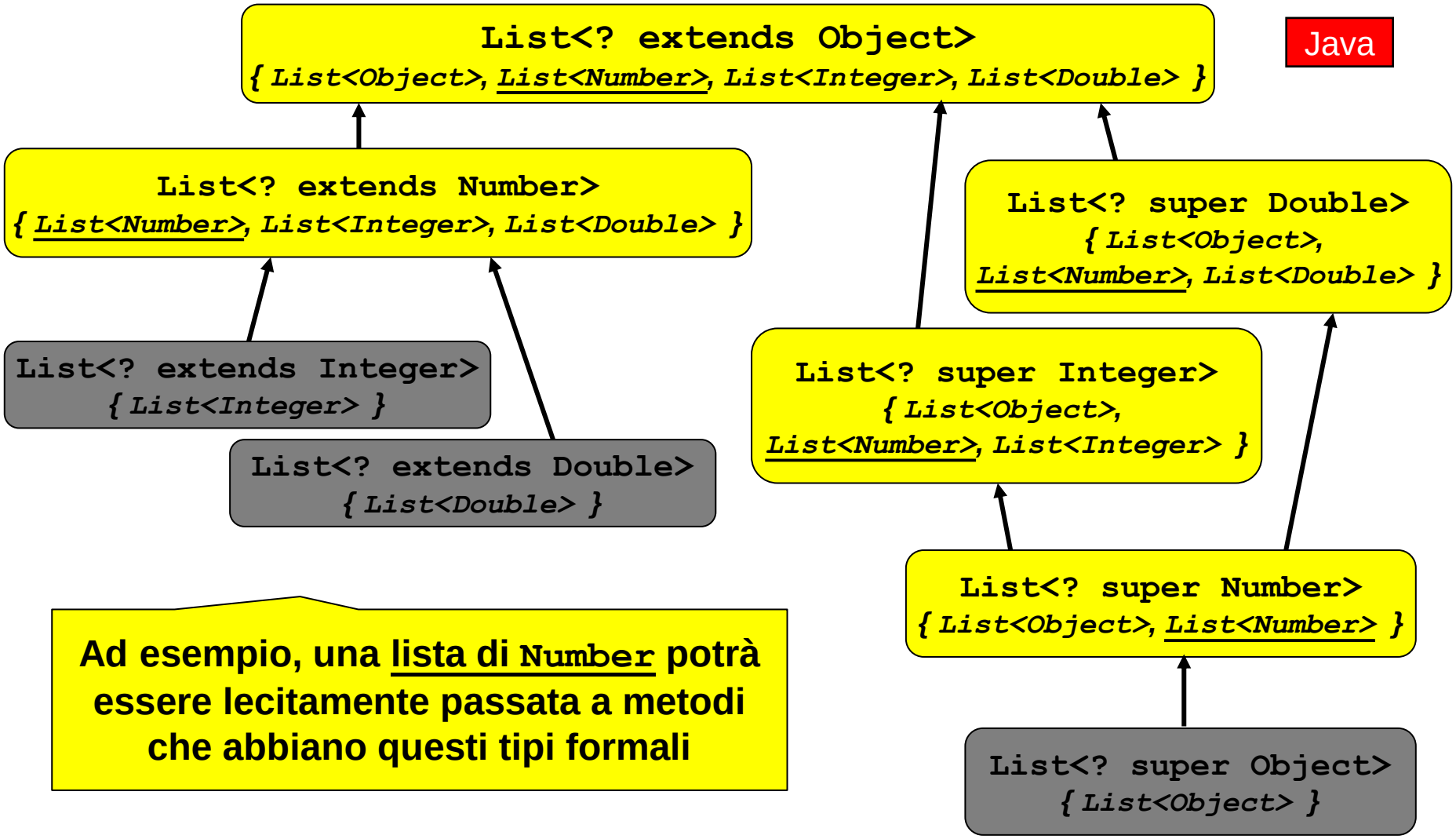
```
List<? extends Object>  = { List<Object>, List<Number>, List<Integer>, List<Double> }

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 ?



SCHEMA RIASSUNTIVO (5)
ESEMPIO DI CODICE

- Traducendo in codice la tassonomia precedente:

Java

```
List<Object>  lObj = new ArrayList<>();
List<Number>  lNum = new ArrayList<>();
List<Integer> lInt = new ArrayList<>();
List<Double>  lDbl = new ArrayList<>();

List<? extends Object>  lObjOrMore ; // { List<Object>, List<Number>, List<Integer>, List<Double> }
List<? extends Number>  lNumOrMore ; // { List<Number>, List<Integer>, List<Double> }
List<? extends Integer> lIntOrMore ; // { List<Integer> }
List<? extends Double>  lDblOrMore ; // { List<Double> }

List<? super Double>    lDblOrLess ; // { List<Object>, List<Number>, List<Double> }
List<? super Integer>   lIntOrLess ; // { List<Object>, List<Number>, List<Integer> }
List<? super Number>    lNumOrLess ; // { List<Object>, List<Number> }
List<? super Object>    lObjOrLess ; // { List<Object> }

lObjOrMore = lObj; lObjOrMore = lNum; lObjOrMore = lInt; lObjOrMore = lDbl;
lNumOrMore = lNum; lNumOrMore = lInt; lNumOrMore = lDbl; // lNumOrMore = lObj;
lIntOrMore = lInt; // lIntOrMore = lObj; lIntOrMore = lNum; lIntOrMore = lDbl;
lDblOrMore = lDbl; // lDblOrMore = lObj; lDblOrMore = lNum; lDblOrMore = lInt;

lDblOrLess = lObj; lDblOrLess = lNum; lDblOrLess = lDbl; // lDblOrLess = lInt;
lIntOrLess = lObj; lIntOrLess = lInt; lIntOrLess = lNum; // lIntOrLess = lDbl;
lNumOrLess = lObj; lNumOrLess = lNum; // lNumOrLess = lDbl;
lObjOrLess = lObj; // lNumOrLess = lObj; lNumOrLess = lNum; lNumOrLess = lInt;
```

SCHEMA RIASSUNTIVO (6)
ESEMPIO DI CODICE

- Traducendo in codice la tassonomia precedente:

Scala

```
import scala.collection.mutable.ListBuffer;
import java.lang.{Double => JDouble};

object EsempioSlide1 {
  def main(args: Array[String]): Unit = {

    val lObj: ListBuffer[Any]     = ListBuffer();
    val lNum: ListBuffer[Number]  = ListBuffer();
    val lInt: ListBuffer[Integer] = ListBuffer();
    val lDbl: ListBuffer[JDouble] = ListBuffer();

    var lObjOrMore: ListBuffer[_ <: Any]    = null ; // { Lis                                    Double> }
    var lNumOrMore: ListBuffer[_ <: Number] = null ; // { Lis
    var lIntOrMore: ListBuffer[_ <: Integer]= null ; // { Lis
    var lDblOrMore: ListBuffer[_ <: JDouble]= null;  // { Lis

    var lDblOrLess: ListBuffer[_ >: JDouble]= null ; // { Lis
    var lIntOrLess: ListBuffer[_ >: Integer]= null ; // { List<Object>, List<Number>, List<Integer> }
    var lNumOrLess: ListBuffer[_ >: Number] = null ; // { List<Object>, List<Number> }
    var lObjOrLess: ListBuffer[_ >: Any]    = null ; // { List<Object> }

    lObjOrMore = lObj; lObjOrMore = lNum; lObjOrMore = lInt; lObjOrMore = lDbl;
    lNumOrMore = lNum; lNumOrMore = lInt; lNumOrMore = lDbl; // lNumOrMore = lObj;
    lIntOrMore = lInt; // lIntOrMore = lObj; lIntOrMore = lNum; lIntOrMore = lDbl;
    lDblOrMore = lDbl; // lDblOrMore = lObj; lDblOrMore = lNum; lDblOrMore = lInt;

    lDblOrLess = lObj; lDblOrLess = lNum; lDblOrLess = lDbl; // lDblOrLess = lInt;
    lIntOrLess = lObj; lIntOrLess = lInt; lIntOrLess = lNum; // lIntOrLess = lDbl;
    lNumOrLess = lObj; lNumOrLess = lNum; // lNumOrLess = lDbl;
    lObjOrLess = lObj; // lNumOrLess = lObj; lNumOrLess = lNum; lNumOrLess = lInt;
  }
}
```

In Scala, la use-site variance usa le keyword **<:** e **>:** invece di **extends** e **super**

NB: necessario usare i numeri Java perché **Number** non esiste nella tassonomia Scala

SCHEMA RIASSUNTIVO (7)
ESEMPIO DI CODICE

- Traducendo in codice la tassonomia precedente:

Kotlin

In Kotlin, la use-site variance usa le keyword **`out`** e **`in`** invece di **`extends`** e **`super`**

```
fun main(args: Array<String>): Unit {

    val lObj: MutableList<Any>     = mutableListOf();
    val lNum: MutableList<Number>  = mutableListOf();
    val lInt: MutableList<Int>     = mutableListOf();
    val lDbl: MutableList<Double>  = mutableListOf();

    var lObjOrMore: MutableList<out Any>     ; // { List<Object>, List<Number>, List<Integer>, List<Double> }
    var lNumOrMore: MutableList<out Number>  ; // { List<Number>, List<Integer>, List<Double> }
    var lIntOrMore: MutableList<out Int>     ; // { List<Integer> }
    var lDblOrMore: MutableList<out Double> ; // { List<Double> }

    var lDblOrLess: MutableList<in Double>   ; // { List<Object>, List<Number>, List<Double> }
    var lIntOrLess: MutableList<in Int>  ; // { List<Object>, List<Number>, List<Integer> }
    var lNumOrLess: MutableList<in Number>   ; // { List<Object>, List<Number> }
    var lObjOrLess: MutableList<in Any>  ; // { List<Object> }

    lObjOrMore = lObj; lObjOrMore = lNum; lObjOrMore = lInt; lObjOrMore = lDbl;
    lNumOrMore = lNum; lNumOrMore = lInt; lNumOrMore = lDbl; // lNumOrMore = lObj;
    lIntOrMore = lInt; // lIntOrMore = lObj; lIntOrMore = lNum; lIntOrMore = lDbl;
    lDblOrMore = lDbl; // lDblOrMore = lObj; lDblOrMore = lNum; lDblOrMore = lInt;

    lDblOrLess = lObj; lDblOrLess = lNum; lDblOrLess = lDbl; // lDblOrLess = lInt;
    lIntOrLess = lObj; lIntOrLess = lInt; lIntOrLess = lNum; // lIntOrLess = lDbl;
    lNumOrLess = lObj; lNumOrLess = lNum; // lNumOrLess = lDbl;
    lObjOrLess = lObj; // lNumOrLess = lObj; lNumOrLess = lNum; lNumOrLess = lInt;

}
```

Varianza in classi generiche

Use-site vs. Declaration-site variance

SVILUPPARE CLASSI (e interfacce) GENERICHE

- Finora ci siamo occupati di *funzioni* con argomenti generici

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

Java

- Ora dobbiamo però porci «dall'altra parte», ossia nei panni di chi deve *definire* tali classi o interfacce generiche
  - Come ha ragionato lo sviluppatore delle Collections?
  - Come si ragiona per definire *nuovi contenitori*?
- Anche qui bisogna affrontare la questione di *quale sia* la *corretta varianza* e di *come specificarla*

SVILUPPARE CLASSI (e interfacce) GENERICHE

- Ricordiamo la regola aurea:
  - *Producers should be covariant*
  - *Consumers should be contravariant*

PECS

«Producers out, Consumers in»

- Prima cosa: *quale componente software* ci serve?
  - un'entità che sia *entrambe le cose* → invarianza
  - un *puro produttore* di elementi → covarianza
  - un *puro consumatore* di elementi → controvarianza
- Studieremo tre esempi
  - **`MyContainer<T>`**, un *contenitore modificabile* di elementi T
  - **`MyProducer<T>`**, *puro produttore* di elementi T
  - **`MyWriter<T>`**, un *puro consumatore* di elementi T

MyContainer

- Un contenitore di elementi di tipo T
  - invariante rispetto a **`T`**
  - **`get`** e **`put`** leggono/scrivono un elemento di tipo **`T`**

Java

```
public class MyContainer<T> {
  private List<T> container;
  public MyContainer(){ container = new ArrayList<>(); }
  public T get(){ return container.remove(0); }
  public void put(T element){ container.add(element); }
  public boolean isEmpty(){ return container.isEmpty(); }
  public int size() { return container.size();}
  public String toString(){ return container.toString();}
}
```

MyContainer: TEST

- Mini-main di test

Java

```
public static void main(String[] args){
  MyContainer<Number> numbers = new MyContainer<>();
  numbers.put(18);
  numbers.put(18.5);
  numbers.put(17.5F);
  System.out.println(numbers);
}
```

Il metodo **put** accetta qualsiasi **Number**

Output

```
[18, 18.5, 17.5]
```

SPECIALIZZAZIONI DI MyContainer

- Versioni specializzate di MyContainer

```
public class MyIntContainer extends MyContainer<Integer> {}
public class MyDoubleContainer extends MyContainer<Double> {}
```

Java

Il metodo **put** accetta rispettivamente solo **Integer** e solo **Double**

- Mini-main di test

```
public static void main(String[] args){
  MyIntContainer ints = new MyIntContainer();
  ints.put(18);    // ints.put(18.5); // ints.put(17.5F); // NO!
  System.out.println(ints);

  MyDoubleContainer reals = new MyDoubleContainer();
  reals.put(18.5); // reals.put(18); // reals.put(17.5F); // NO!
  System.out.println(reals);
}
```

Java

SPECIALIZZAZIONI DI MyContainer

- Versioni specializzate di MyContainer

```
public class MyIntContainer extends MyContainer<Integer> {}
public class MyDoubleContainer extends MyContainer<Double> {}
```
Java

- Mini-main di test

```
public static void main(String
  MyIntContainer ints = new
  ints.put(18);     // ints.p
  System.out.println(ints);
  MyDoubleContainer reals =
  reals.put(18.5); // real
  System.out.println(real
  // numbers = ints;          // NO, MyContainer è invariante
  // numbers = reals;         // NO, MyContainer è invariante
}
```

Giustamente `MyContainer` è del tutto *incompatibile* con le sue specializzazioni: d'altronde, manipola elementi *sia per leggerli che per scriverli.*

Si può solo specificare la varianza dei singoli metodi `put` e `get`, ma non del contenitore globalmente inteso.

MyProducer

- Un puro produttore
  - implementativamente, al suo interno sfrutta un **`MyContainer`**

Java

```
public class MyProducer<T> {
  private List<T> container;
  private int i=0;
  public MyProducer(List<T> data){ container = data; i=0;}
  public T get (){
   i = (i+1) % container.size();
   return container.get(i);
  }
  public boolean isEmpty(){ return container.isEmpty(); }
  public String toString() { return container.toString(); }
}
```

È preconfigurato con una lista fissa di dati

Di fatto, un adapter per **`MyContainer`** che maschera il metodo **`put`**

Il metodo **`get`** restituisce uno dei dati, a rotazione

MyProducer: TEST

- Test del puro produttore

Java

```
public static void main(String[] args){
   MyProducer<Number> producer = new MyProducer<>(
                                    List.of(22, 7, -13.3, 1.1));
    testWithNums(producer);
}

public static void testWithNums(MyProducer<Number> producer){
   System.out.println(producer.get());
   System.out.println(producer.get());
   System.out.println(producer.get());
   System.out.println(producer.get());
   System.out.println(producer.get());
   System.out.println(producer);
}
```

Un produttore di **Number**

```
7
-13.3
1.1
22
7
[22, 7, -13.3, 1.1]
```

MyWriter

- Un puro consumatore
  - implementativamente, al suo interno sfrutta un **`MyContainer`**

```
public class MyWriter<T> {
  private List<T> container;
  public MyWriter(){ container = new ArrayList<>(); }
  public void put(T element){ container.add(element); }
  public boolean isEmpty(){ return container.isEmpty(); }
  public String toString() { return container.toString(); }
}
```

Java

Di fatto, un adapter per **`MyContainer`** che maschera il metodo **`get`**

- Mini-main di test

```
public static void main(String[] args){
  MyWriter<Number> writer = new MyWriter<>();
   writer.put(18); writer.put(18.5); writer.put(17.5F);
}
```

Java

Ci possiamo scrivere solo dei **`Number`**

DUE POSSIBILI SPECIALIZZAZIONI
Produttore e consumatore di interi

- Un puro produttore di *soli interi*

```
public class MyIntProducer extends MyProducer<Integer> {
  public MyIntProducer(List<Integer> data) { super(data);}
}
```

Java

Produce **solo** `Integer`

- Un puro consumatore di *soli interi*

```
public class MyIntWriter extends MyWriter<Integer> {
}
```

Java

Ci possiamo scrivere **solo** `Integer`

- DOMANDE
  - Come collaudiamo queste specializzazioni?
  - Ma soprattutto: che varianza dovrebbero avere?

SPECIALIZZAZIONI: TEST

- Specifica di test
  - il consumatore di **`Integer`** dovrebbe accettare solo interi
  - il produttore di **`Integer`** dovrebbe produrre solo interi

```
public static void main(String[] args){
  MyIntWriter intWr = new MyIntWriter();
  intWr.put(18); // intWr.put(18.5); // intWr.put(17.5F);}
  MyIntProducer producer = new MyIntProducer(List.of(3,6,9,-4));
  testWithInt(producer);
}
```

Ci possiamo scrivere **solo `Integer`**

Produce **solo `Integer`**

Metodo analogo al precedente **`testWithNums`**

- Rimane la domanda: *quale varianza?*

QUALE VARIANZA PER PRODUTTORI E CONSUMATORI?

- Il contenitore di base era invariante
  - giusto così, avendo sia metodi di lettura che si scrittura
- In teoria, per il puro consumatore e il puro produttore la questione potrebbe (dovrebbe?) essere diversa:
  - il puro *produttore* potrebbe utilmente essere *covariante*
  - il puro *consumatore* potrebbe utilmente essere *controvariante*
- MA per esprimere bene ciò servirebbe la *declaration-site variance*
  - bisognerebbe cioè poter «etichettare» *l'intera classe* come *covariante* o *controvariante*
  - purtroppo, Java NON offre questo livello di espressività
  - ci si deve accontentare della use-site variance.. con alcuni limiti
  - in Scala o Kotlin, invece, ci riusciremmo (in C#, con qualche limite)

LIMITI DELLA USE-SITE VARIANCE IN JAVA

- In Java, con la *use-site variance,* possiamo marcare *specifici metodi/funzioni* come covarianti/controvarianti
  - ad esempio, possiamo generalizzare i due test precedenti:

```
static void testProducer(MyProducer<? extends Number> producer){
   System.out.println(producer.get());
   ...
   System.out.println(producer);
}
```

Funziona con **Producer<Number>** *e tutte le sue specializzazioni*

Java

```
static void testWriter(MyWriter<? super Integer> writer){
   System.out.print("testWriter:");
   writer.put(18);
   writer.put(19);
   writer.put(17);
   System.out.println(writer);
}
```

Funziona con **Writer<Integer>** *e tutte le sue generalizzazioni*

Java

LIMITI DELLA USE-SITE VARIANCE IN JAVA

- MA non riusciamo a esprimere l'idea che un componente potrebbe essere covariante o controvariante *in quanto tale:*
  - il puro consumatore potrebbe essere controvariante in modo safe:

Java

```
public static void main(String[] args){
   MyWriter<Number> numWriter = …;
   // numWriter = new MyIntWriter(); // NO, giustamente

   MyIntWriter intWriter = numWriter;
   // NON COMPILA, MA sarebbe controvariante e avrebbe senso!
}
```

  - il puro produttore potrebbe essere covariante in modo safe:

Java

```
public static void main(String[] args){
   MyIntProducer intProducer = …;
   // intProducer = new MyProducer<Number(); // NO, giustamente
   MyProducer<Number> producer = intProducer;
   // NON COMPILA, MA sarebbe covariante e avrebbe senso!
}
```

USE-SITE VARIANCE vs. DECLARATION-SITE VARIANCE

- Negli esempi visti finora, abbiamo sempre e solo usato la *use-site variance*
  - si specifica che un dato metodo dev'essere covariante o controvariante in quello specifico caso d'uso
  - è il solo schema di varianza possibile in Java

Java
Scala
Kotlin

- Anche se non supportato in Java, esiste però anche lo schema alternativo della *declaration-site variance*
  - si specifica che un componente software (es. classe) debba essere, nella sua totalità, o tutto covariante, o tutto controvariante
  - è il solo schema di varianza possibile in C# (con limiti)
  - Scala e Kotlin li supportano entrambi

C#
Scala
Kotlin

USE-SITE VARIANCE vs. DECLARATION-SITE VARIANCE

Use-site variance

	Java	Scala	Kotlin
≤ T Covarianza	`<? extends T>`	`[_ <: T]`	`<out T>`
≥ T Controvarianza	`<? super T>`	`[_ >: T]`	`<in T>`
Invarianza	`<T>`	`[T]`	`<T>`

NB: C# non supporta la *use-site* variance

Declaration-site variance

	Java	Scala	Kotlin	C#
≤ T Covarianza	*non supportata*	`[+T]`	`<out T>`	
≥ T Controvarianza	*non supportata*	`[-T]`	`<in T>`	
Invarianza	`<T>`	`[T]`	`<T>`	

USE-SITE VARIANCE IN JAVA, SCALA & KOTLIN

Use-site variance

		Java	Scala	Kotlin
≤ T	Covarianza	`<? extends T>`	`[_ <: T]`	`<out T>`
≥ T	Controvarianza	`<? super T>`	`[_ >: T]`	`<in T>`
	Invarianza	`<T>`	`[T]`	`<T>`

Le keyword «**extends**» e «**super**» si riferiscono alla *posizione nella tassonomia* (*sotto / sopra*)

I simboli «**minore**» e «**maggiore**» si riferiscono all' *ampiezza dell'insieme*

Le keyword «**in**» e «**out**» si riferiscono a *ciò che si può fare (leggere / scrivere) producer / consumer*

DECLARATION-SITE VARIANCE in SCALA & KOTLIN

Declaration-site variance

		Java	Scala	Kotlin	C#
≤ T	Covarianza	*non supportata*	`[+T]`	`<out T>`	`<out T>`
≥ T	Controvarianza	*non supportata*	`[-T]`	`<in T>`	`<in T>`
	Invarianza	`<T>`	`[T]`	`<T>`	`<T>`

Java non consente la specifica di varianza all'atto della dichiarazione di un componente

In Scala la declaration-site variance adotta simboli specifici, +/–, diversi dalle keyword «`in`» e «`out`» adottate per la use-site variance

In Kotlin si usano invece le stesse keyword «`in`» e «`out`» già viste per la use-site variance

In C# la declaration-site variance è ammessa solo su interfacce e purché non usino value types; le keyword sono «`in`» e «`out`»

DECLARATION-SITE VARIANCE IN KOTLIN

- Il puro consumatore controvariante (keyword **`in`**):

```
open public class MyWriter<in T> {
  private val container : MutableList<T> = mutableListOf<T>();
  fun put(element: T):Unit { container.add(element); }
  fun isEmpty():Boolean { return container.isEmpty(); }
  override fun toString():String { return container.toString();}
}
```

Kotlin

- Mini-test:

```
val numWriter = MyWriter<Number>();
val intWriter : MyWriter<Int> = numWriter;
```

Kotlin

In Java non compilava, **qui invece sì** perché il writer è dichiarato come controvariante

L'opposto, ovviamente, non compila

```
val numWriter : MyWriter<Number> = new MyWriter<Int>(); //NO
```

Kotlin

DECLARATION-SITE VARIANCE IN KOTLIN

- Il puro produttore covariante (keyword **`out`**):

```
open public class MyProducer<out T> {
  private val container : MutableList<T>();
  fun get ():T { return container.get(0); }
  fun isEmpty():Boolean { return container.isEmpty(); }
  override fun toString():String { return container.toString();}
}
```

NB: è uno scheletro (manca il costruttore)

Kotlin

- Mini-test:

```
var numProducer = MyProducer<Number>();
var intProducer = MyProducer<Int>();
numProducer = intProducer; // covariante
// IN ALTERNATIVA
// intProducer = numProducer; // NO, controvariante
```

In Java non compilava, **qui invece sì** perché il producer è dichiarato come covariante

Kotlin

DECLARATION-SITE VARIANCE IN SCALA

- Il puro consumatore controvariante (keyword –) e relativi test:

Scala

```
import scala.collection.mutable.ListBuffer;

class MyWriter[-T] {
  private val container : ListBuffer[T] = ListBuffer[T]();
  def put(element: T):Unit = { container += element; }
  def isEmpty():Boolean = container.isEmpty;
  override def toString():String = container.toString();
}

// esempio coi tipi Number e Integer di Java (Number non esiste in Scala)
val numWriter : MyWriter[Number] = new MyWriter[Number]();
val intWriter : MyWriter[Integer] = numWriter;

// esempio coi tipi AnyVal e Int di Scala
val numWriter2 : MyWriter[AnyVal] = new MyWriter[AnyVal]();
val intWriter2 : MyWriter[Int] = numWriter2; // con i value type di Scala

// val numWriter3 : MyWriter[Number] = new MyWriter[Integer](); // NO
// val numWriter3 : MyWriter[AnyVal] = new MyWriter[Int](); // NO
```

DECLARATION-SITE VARIANCE IN SCALA

- Il puro produttore covariante (keyword **+**) e relativi test:

Scala

```
class MyProducer[+T] {
  private val container : ListBuffer[T] = ListBuffer[T]();
  def get ():T = container(0);
  def isEmpty():Boolean = container.isEmpty;
  override def toString():String = container.toString();
}

var numProducer = MyProducer[Number]();
var intProducer = MyProducer[Integer]();

var numProducer2 = MyProducer[AnyVal]();
var intProducer2 = MyProducer[Int]();

numProducer  = intProducer;   // covariante
numProducer2 = intProducer2; // covariante

// intProducer = numProducer; // NO, controvariante
// intProducer2 = numProducer2; // NO, controvariante
```

DECLARATION-SITE VARIANCE IN C#

- Premessa: poiché in C# la declaration-site variance è ammessa
  - solo su interfacce (non nelle dichiarazioni di classi)
  - solo su tipi-riferimento, esclusi i value types (quindi niente int, etc.),
- occorre preliminarmente ricostruirsi alcune classi analoghe a quelle di Java, Scala e Kotlin, a fini di test
  - ricostruiamo **`Number`**, **`Int`** e **`Double`**

C#

```
using System;
using System.Collections.Generic;

public abstract class Number {
    public abstract int intValue();
    public abstract double doubleValue();
}

public class Int : Number {
    private int value;
    public Int(int v) { this.value=v;}
    public override int intValue() { return value; }
    public override double doubleValue() { return (double) value; }
}

public class Double : Number {
    private double value;
    public Double(double v) { this.value=v;}
    public override int intValue() { return (int) Math.Round(value); }
    public override double doubleValue() { return value; }
}
```

DECLARATION-SITE VARIANCE IN C#

- Il puro consumatore controvariante (keyword **`in`**) :

```
interface IMyWriter<in T> {
    void Put(T element);
    bool IsEmpty();
}

public class MyWriter<T> : IMyWriter<T> {
    private List<T> container = new List<T>();
    public void Put(T element) { container.Add(element); }
    public bool IsEmpty() { return container.Count==0; }
    override public String ToString() { return container.ToString();}
}
```

C#

Keyword **`in`**

- Il puro produttore covariante (keyword **`out`**) :

```
interface IMyProducer<out T> {
    T Get();
    bool IsEmpty();
}

public class MyProducer<T> : IMyProducer<T> {
    private List<T> container = new List<T>();
    public T Get () { return container[0]; }
    public bool IsEmpty() { return container.Count==0; }
    override public String ToString() { return container.ToString();}
}
```

C#

Keyword **`out`**

In C#, *solo le interfacce* hanno la specifica di varianza:

le classi no, sono *sempre invarianti*

DECLARATION-SITE VARIANCE IN C#

- Test:

C#

```
public static void Main() {
    Console.WriteLine("Hello World");
    IMyWriter<Number> numWriter = new MyWriter<Number>();
    IMyWriter<Int> intWriter = numWriter;
    //IMyWriter<Number> numWriter2 = new MyWriter<Int>(); //NO
    //--------
    IMyProducer<Number> numProducer = new MyProducer<Number>();
    IMyProducer<Int> intProducer = new MyProducer<Int>();
    numProducer = intProducer; // covariante
    // intProducer = numProducer; // NO, controvariante
}
```

Varianza in classi generiche

MyStack refactored

ESEMPIO FINALE REFACTORING DI `MyStack`

- Ricordate il nostro stack (con la push in cascading) ?

Java

```
public class MyStack<T> {
  private List<T> storage;
  public MyStack(){ storage = new ArrayList<>(); }
  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(); }
}
```

- Così definito, è *invariante*: uno stack di un tipo è rigorosamente incompatibile con uno stack di un altro
- Ma.. è proprio indispensabile?

ESEMPIO FINALE
REFACTORING DI `MyStack`

- Situazione attuale:

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

Java

- Essendo invariante, il controllo di tipo è stringente:
  - **`stack1`** accetta solo **`int`** e null'altro (né **`float`**, né **`double`**)
  - **`stack2`** accetta solo **`double`** e null'altro (né **`float`**, né **`int`**)

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

Java

NO! 18 è `int`

NO! 22.0F è `float`

MyStack
INVARIANZA = INCOMPATIBILITÀ

- Ricordate **`MyStack`** (era invariante) ?

Java

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

- essendo invariante, il controllo di tipo è stringente:
  - **`stack1`** accetta solo **`int`** e null'altro (né **`float`**, né **`double`**)
  - **`stack2`** accetta solo **`double`** e null'altro (né **`float`**, né **`int`**)
  - i due stack sono totalmente incompatibili fra loro

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

MyStack
INVARIANZA = INCOMPATIBILITÀ

- Ovviamente, *ogni tentativo di mischiarli è stroncato:*

Java

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

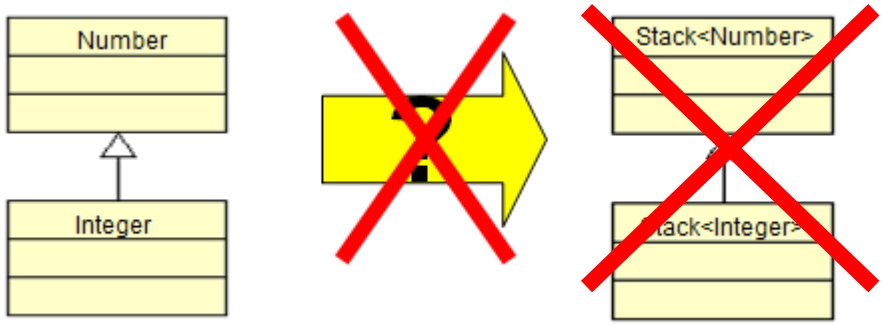
MyStack
INVARIANZA = INCOMPATIBILITÀ

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

Java

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

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



MyStack
REVISIONE & ESTENSIONE

- Aggiungiamo un metodo **moveFrom** per spostare elementi da uno stack a un altro:

Java

```
public class MyStack<T> {
  private List<T> storage;
  public MyStack(){ storage = new ArrayList<>(); }
  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());
  }
}
```

<T> necessario

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

- Perché limitare **moveFrom** a operare solo su stack di **T** ?

MyStack
REVISIONE & ESTENSIONE

- In questo caso, **moveFrom** avrebbe funzionato tranquillamente, ma *la rigidità dell'invarianza blocca la compilazione*

Java

```
MyStack<Integer> stack1 = new MyStack<>();
MyStack<Integer> stack2 = new MyStack<>();
MyStack<Number>  stack0 = new MyStack<>();
stack1.moveFrom(stack2); // ovvio che funzioni,,
stack0.moveFrom(stack1); // STRONCATO INUTILMENTE
```

- È un peccato che questa chiamata a **moveFrom** venga stroncata, perché, *considerato quel che deve fare,* sarebbe stata perfettamente type-safe e in grado di funzionare
- *Possiamo ovviare inserendo un tipo wildcard*

UN `MyStack` CON WILDCARD

- Riformuliamo **`moveFrom`**

Java

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

- Con questa modifica, **`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 il vincolo che garantisce che **`moveFrom`** sia sicura e sensata ☺

UN `MyStack` CON WILDCARD

- Si può abbreviare la sintassi *evitando di introdurre il tipo* ***E***

Java

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

- Dice la stessa cosa in modo più diretto e compatto
  - come `source` va bene un qualunque stack di "cose che estendano T"

UN `MyStack` CON WILDCARD

- Se volessimo anche *svuotare e stampare lo stack*, potremmo definire una funzione **`emptyAndPrintStack`**
- Che argomento dovrebbe avere?
  - per stampare non serve alcuna ipotesi: ogni **`Object`** va bene
  - usiamo il tipo unbounded **`MyStack<?>`**, che come noto è «massimamente covariante»

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

Java

- In quanto tale, si può invocare su qualunque stack:
  - **`stack0`**, **`stack1`**, **`stack2`**, **`MyStack<PhonePlan>`**,…