JAVA GENERICS

Java: Interfacce e astrazione

Supponiamo di voler definire liste di diversi tipi:

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

… e ListOfStrings e …

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

Usiamo I tipi!!!

List<Integer>

List<Number>

List<String>

List<List<String>>

…

Esempio: una coda

```
class CircleQueue { //coda di cerchi

    // rappresentazione come array
    private Circle[] circles;

    // costruttore
    public CircleQueue(int size) {...}

    // metodi
    public boolean isFull() {...}
    public boolean isEmpty() {...}
    public void enqueue(Circle c) {...}
    public Circle dequeue() {...}
}
```

Esempio: una coda e una seconda coda

```
class CircleQueue { //coda di cerchi

    // rappresentazione come array
    private Circle[] circles;

    // costruttore
    public CircleQueue(int size) {...}

    // metodi
    public boolean isFull() {...}
    public boolean isEmpty() {...}
    public void enqueue(Circle c) {...}
    public Circle dequeue() {...}
}
```

```
class PointQueue { //coda di punti

    // rappresentazione come array
    private Point[] circles;

    // costruttore
    public PointQueue(int size) {...}

    // metodi
    public boolean isFull() {...}
    public boolean isEmpty() {...}
    public void enqueue(Point p) {...}
    public Point dequeue() {...}
}
```

Esempio: una coda e una seconda coda

```
class CircleQueue { //coda di cerchi

    // rappresentazione come array
    private Circle[] circles;

    // costruttore
    public CircleQueue(int size) {...}

    // metodi
    public boolean isFull() {...}
    public boolean isEmpty() {...}
    public void enqueue(Circle c) {...}
    public Circle dequeue() {...}
}
```

```
class PointQueue { //coda di punti

    // rappresentazione come array
    private Point[] circles;

    // costruttore
    public PointQueue(int size) {...}

    // metodi
    public boolean isFull() {...}
    public boolean isEmpty() {...}
    public void enqueue(Point p) {...}
    public Point dequeue() {...}
}
```

Spot the differences!

Astrazione (da wikipedia)

Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts

Esempio: una terza coda (di Object)

```
class CircleQueue { //coda di cerchi

    // rappresentazione come array
    private Circle[] circles;

    // costruttore
    public CircleQueue(int size) {...}

    // metodi
    public boolean isFull()
    public boolean isEmpty()
    public void enqueue(Circ
    public Circle dequeue()
}
```

```
class PointQueue { //coda di punti

    // rappresentazione come array
    private Point[] circles;

    // costruttore
    public PointQueue(int size) {...}

    boolean isFull() {...}
    boolean isEmpty() {...}
    id enqueue(Point p) {...}
    int dequeue() {...}
```

```
class ObjectQueue { //coda di oggetti

    // rappresentazione come array
    private Object[] circles;

    // costruttore
    public ObjectQueue(int size) {...}

    // metodi
    public boolean isFull() {...}
    public boolean isEmpty() {...}
    public void enqueue(Object o) {...}
    public Object dequeue() {...}
}
```

Usiamo la (terza) coda

```
ObjectQueue cq = new ObjectQueue(10);
cq.enqueue(new Circle(new Point(0, 0), 10));
cq.enqueue(new Circle(new Point(1, 1), 5));
 :
```

Altre operazioni

Estrarre dalla coda un oggetto instanza di Circle è leggermente più complicato

Circle c = cq.dequeue();

NO! compilation error: non possiamo assegnare una espressione di tipo Object a una variabile di tipo Circle senza una operazione di casting esplicito

Circle c = (Circle) cq.dequeue();

Tuttavia …

…. potrebbe sollevare una ClassCastException se la coda contenesse oggetti che non sono di tipo Circle

Soluzione:

```
Object o = cq.dequeue();
  if (o instanceof Circle) {
    Circle c = (Circle)o;
}
```

E' una **soluzione complicata**, e che va fatta **ad ogni chiamata** a dequeue..

A partire da Java 5

```
class Queue<T> {
  private T[] objects;
      :
  public Queue(int size) {...}
  public boolean isFull() {...}
  public boolean isEmpty() {...}
  public void enqueue(T o) {...}
  public T dequeue() {...}
}
```

```
Queue<Circle> cq = new Queue<Circle>(10);
cq.enqueue(new Circle(new Point(0, 0), 10));
cq.enqueue(new Circle(new Point(1, 1), 5));
Circle c = cq.dequeue();  // Niente cast!! Controllo statico
```

Da notare

```
Queue<Circle> cq = new Queue<Circle>(10);
cq.enqueue(new Point(1, 3));
```

Questo codice genera (giustamente) un errore a tempo di compilazione

Java Generics

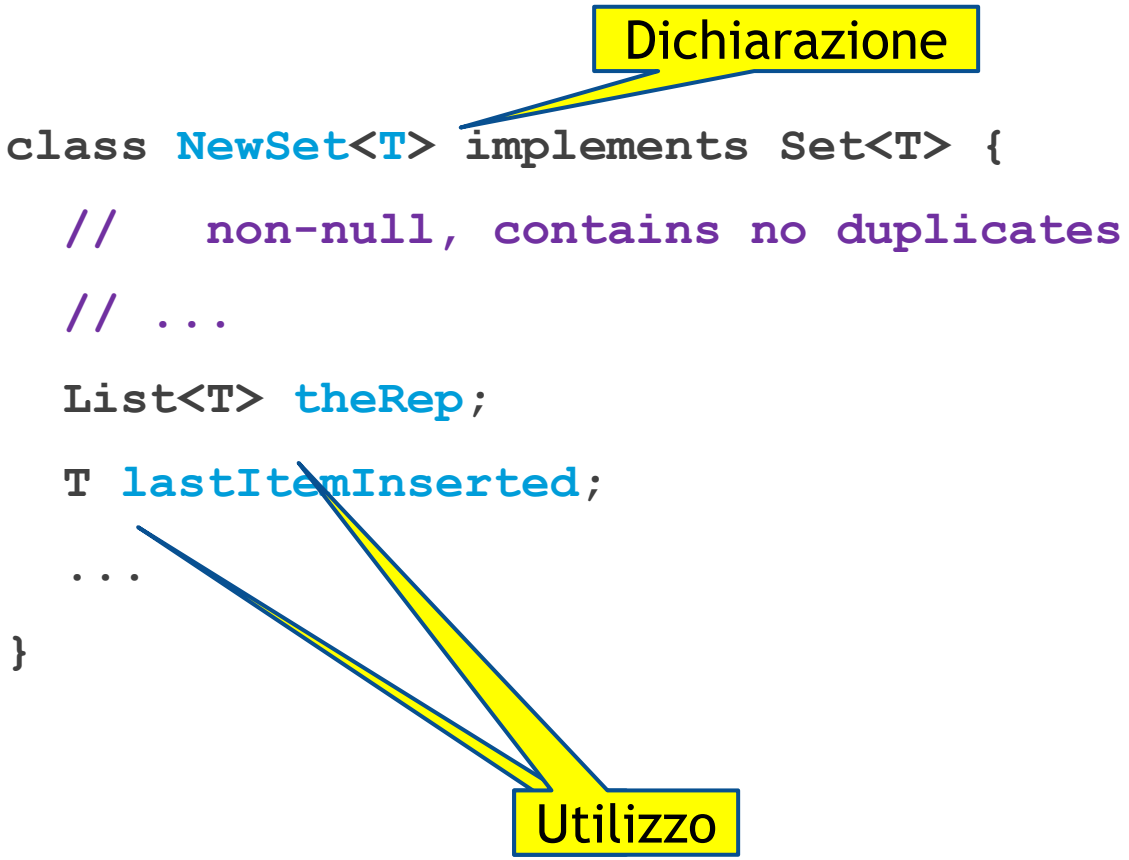
- Meccanismo di astrazione linguistica che consente la definizione di classi e metodi parametrici rispetto al tipo che utilizzano
- Astrazione che permette di definire algoritmi che si applicano in contesti diversi, ma in cui l'interfaccia e il funzionamento generale e l'implementazione possono essere definiti in modo tale da applicarsi indipendentemente dal contesto dell'applicazione.

Altro esempio

```
interface Collection<T> {
   boolean contains(T x);
   boolean remove(T x);
   boolean add(T x);
}
```

```
Collection<String> c = ...
  c.add("hi"); // OK!
  c.add(2); // static error
for (String s : c) {
// use s
}
```

Variabili di tipo



Dichiarare generici

```
class Name<TypeVar1, …, TypeVarN> {…}
```

```
interface Name<TypeVar1, …, TypeVarN> {…}
```

- convenzioni standard
  `T` per `Type`, `E` per `Element`,
  `K` per `Key`, `V` per `Value`, …

Istanziare una classe generica significa fornire un valore di tipo

```
Name<Type1, …, TypeN>
```

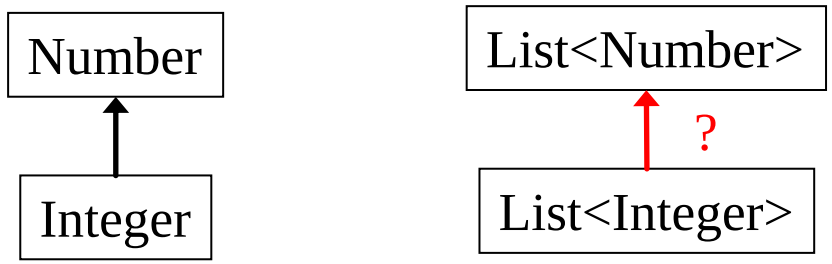
Lista Generica

```
class GenList<T> implements Collection<T> {
  int size = 0;
  Node<T> head = null; //riferiemnto al primo Node;
                       //null se lista vuota

  boolean contains(T x) {
    Node<T> n = head;
    while (n != null) {
      if (x.equals(n.data)) return true;
      n = n.next;
    }
    return false;
  }
```

Lista Generica (Cont)

```
  boolean add(T x) {
    head = new Node<T>(x, head);
    size++;
  }

  boolean remove(T x) {
    Node n = head, p = null;
    while (n != null && !x.equals(n.data)) {p = n; n = n.next; }
    if (n == null) return false;
    size--;
    if (p == null) head = n.next; else p.next = n.next;
    return true;
  }
}
```

Generici e la nozione di sottotipo



- **`Integer`** è un sottotipo di **`Number`**

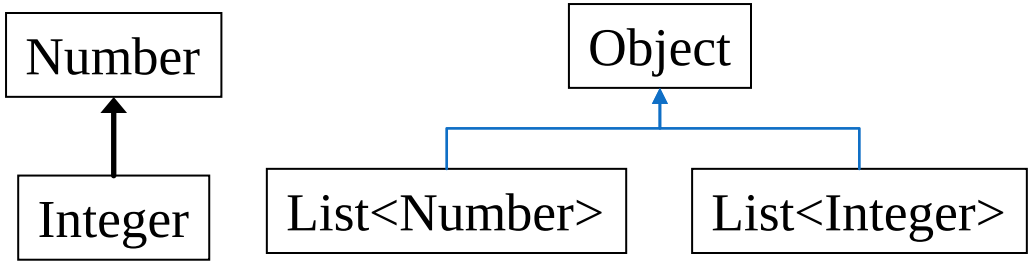
- **`List<Integer>`** è un sottotipo di **`List<Number>`** ?

Quali sono le regole di Java?

Type2 <: **Type3** $\not\Rightarrow$ **Type1<Type2>** <: **Type1<Type2>**

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

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



Varianza per tipi

Supponiamo che A(T) sia un opportuno tipo definito usando il Tipo T:

- A è *covariante* se T<:S implica A(T)<:A(S),
- A è *contravariante* se T<:S implica A(S)<:A(T),
- A è *bivariante* se è sia covariante che contravariante,
- A è *invariante* se non è covariante e controvariante

Esempi

Tipi e sottotipi

- Integer è un sottotipo di Number
- ArrayList<E> è un sottotipo di List<E> (poiché `ArrayList<E> implements List<E>`)
- List<E> è un sottotipo di Collection<E> (poiché `List<E> extends Collection<E>`)

Ma

- List<Integer> non è un sottotipo di List<Number>

Esempi

Tipi e sottotipi

- Integer è un sottotipo di Number
- ArrayList<E> è un sottotipo di List<E> (poiché ArrayList<E> implements List<E>)
- List<E>

Ma

- List<In

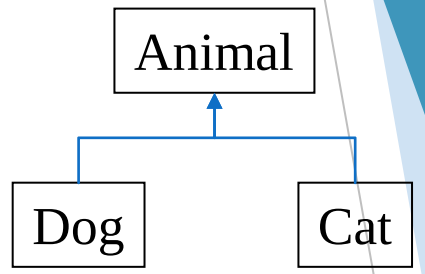
NOTA: OCaml ha regole diverse… (covarianza)

```
class c1 = object method m1 = 10 end;;
class c2 = object method m1 = 5 method m2 = 12 end;;
let lst1 = [new c1];; (* tipo c1 list *)
let lst2 = [new c2];; (* tipo c2 list *)
let lstlst = [lst1; (lst2 :> c1 list)];; (* tipo c1 list list *)
```

Funziona!

Perché i generici in Java non sono covarianti?

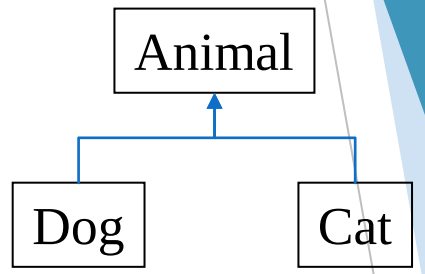
Se Java ammettesse la covarianza sui tipi generici, il seguente frammento di codice sarebbe corretto:




```
List<Dog> dogs = new ArrayList<Dog>(); // ArrayList implements List
List<Animal> animals = dogs; // Non consentito poiché non vale covarianza
animals.add(new Cat());
Dog dog = dogs.get(0); // Qual è il tipo effettivo di dog?
```

Perché i generici in Java non sono covarianti?

Se Java ammettesse la covarianza sui tipi generici, il seguente frammento di codice sarebbe corretto:




```
List<Dog> dogs = new ArrayList<Dog>(); // ArrayList implements List
List<Animal> animals = dogs; // Non consentito poiché non vale covarianza
animals.add(new Cat());
Dog dog = dogs.get(0); // Qual è il tipo effettivo di dog?
```

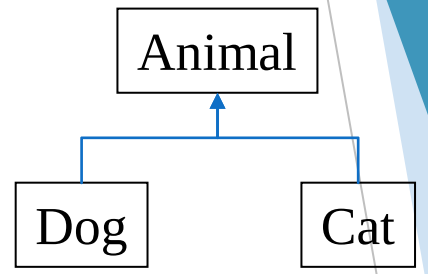
Il problema è che l'assegnamento crea un alias con un tipo diverso, e la lista è modificabile!

E con gli array? Sorpresa!

- Sappiamo che per i generici la nozione di sottotipo è invariante, pertanto, per analogia se **`Type1`** è un sottotipo di **`Type2`**, allora **`Type1[]`** e **`Type2[]`** non dovrebbero essere correlati
- Ma Java è strano, se **`Type1`** è un sottotipo di **`Type2`**, allora **`Type1[]`** è un sottotipo di **`Type2[]`**
  - Java (ma anche C#) ha fatto questa scelta prima dell'introduzione dei generici
  - cambiarla ora è un po' troppo invasivo per tutti i programmi Java già scritti (backward compatibility)

Ripetiamo l'esempio con gli array?

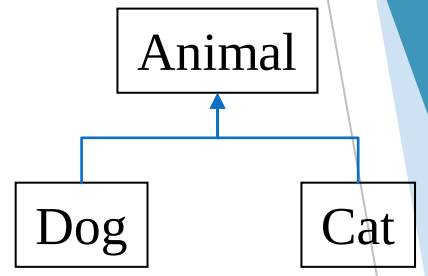
Il seguente esempio supera i controlli del compilatore:




```
Dog[] dogs = new Dog[10]; // array di Dog
Animal[] animals = dogs; // Consentito! Poiché per array vale covarianza
animals.add(new Cat());
Dog dog = dogs.get(0); // Eccezione a runtime: ArrayStoreException
```

Ripetiamo l'esempio con gli array?

Il seguente esempio supera i controlli del compilatore:

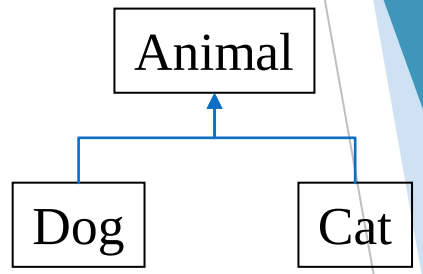



```
Dog[] dogs = new Dog[10]; // array di Dog
Animal[] animals = dogs; // Consentito! Poiché per array vale covarianza
animals.add(new Cat());
Dog dog = dogs[0]; // Eccezione a runtime: ArrayStoreException
```

Nel caso degli array questo esempio genera un errore rilevato solo a tempo di esecuzione!

Ripetiamo l'esempio con gli array?

Il seguente esempio supera i controlli del compilatore:




```
Dog[] dogs = new Dog[10]; // array di Dog
Animal[] animals = dogs; // Consentito! Poiché per array vale covarianza
animals.add(new Dog());
Dog dog = dogs[0]; // Eccezione a runtime: ArrayStoreException
```

In questo caso il programma compila e funziona correttamente (no eccezione)

Con gli array il controllo è dinamico: se i tipi effettivi (di **dog** e **animals[0]**) sono compatibili tutto funziona, altrimenti eccezione.

Nel caso dei generici il controllo è statico: dato che non vale la covarianza già l'assegnamento **animals = dogs** viene rigettato a causa dei tipi statici diversi, indipendentemente da come potrebbe andare a tempo di esecuzione

Il motivo di questa differenza

- Già detto: gli array funzionano in questo modo fin dalle prime versioni di Java, mentre i generici sono stati introdotti dopo
  - Backward compatibility: allineare il funzionamento degli array a quello dei generici avrebbe causato problemi ai programmi già scritti
- Inoltre, gli array "conoscono" il proprio tipo dinamico
  - ogni array ha nel proprio descrittore l'informazione sul tipo dinamico (`Book[ ]`)
  - la Java Virtual Machine può fare un controllo a runtime ed eventualmente sollevare la **`ArrayStoreException`**
  - Nel caso delle classi generiche invece l'informazione sul tipo non è mantenuta a runtime (type erasure, vedremo…), quindi non c'è modo di fare un controllo dinamico e si ricorre al controllo statico (più vincolante)

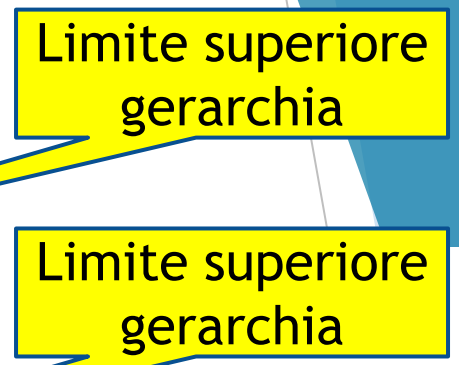
Generici e gerarchie

Con i generici niente covarianza… ma c'è un apposito costrutto linguistico per fare qualcosa di simile

Limite superiore gerarchia

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

List1<Date> // OK, Date è un sottotipo di Object

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

Visione effettiva dei generici




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

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

L'istanziazione resta identica

```
Name<Type1, …, TypeN>
```

- Ma *compile-time error* se il tipo non è un sottotipo del limite superiore della gerarchia
  - Attenzione: è sottotipo anche se il limite superiore è un'interfaccia che viene implementata

Usiamo le variabili di tipo

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

- concettualmente questo corrisponde a forzare una sorta di precondizione sulla istanziazione del tipo

```
class List1<E extends Object> {
  void m(E arg) {
    arg.asInt( );  // compiler error, E potrebbe
                   // non avere l'operazione asInt
  }
}

class List2<E extends Number> {
  void m(E arg) {
    arg.asInt( );  // OK, Number e tutti i suoi
                   // sottotipi supportano asInt
  }
}
```

Vincoli di tipo

`<TypeVar extends SuperType>`

- *upper bound*; va bene il tipo **`SuperType`** o uno dei suoi sottotipi

`<TypeVar extends ClassA & InterfB & InterfC & … >`

- *multiple* upper bounds (al più una classe e arbitrarie interfacce)

`<TypeVar super SubType>`

- *lower bound*; va bene il tipo **`SubType`** o uno dei suoi supertipi

Esempio

```
// insieme ordinato (la classe degli elementi deve
// implementare l'interfaccia Comparable)
public class TreeSet<T extends Comparable<T>> {
    ...
}
```

Altro esempio

Metodo che copia liste (di qualunque tipo)

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

La soluzione va bene, ma ancora meglio questa (più generica)

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

Oppure, analogamente:

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

Altro esempio

Le variabili di tipo possono essere dichiarate anche a livello di singolo metodo!

Metodo che copia liste (di qualunque tipo)

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

Consente di copiare una lista di Dogs in una lista di Animals (vedi esempi precedenti)

La soluzione va bene, ma ancora meglio questa (più generica)

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

Oppure, analogamente:

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

Wildcard: ?

Wildcard = una variabile di tipo anonima

- **?** tipo non conosciuto
- si usano le wildcard quando si usa un tipo esattamente una volta ma non si conosce il nome
- l'unica cosa che si sa è l'unicità del tipo

Sintassi delle wildcard

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

Esempi

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

- maggiormente flessibile rispetto a
  `void addAll(Collection<E> c);`
- zucchero sintattico per
  `<T extends E> void addAll(Collection<T> c);`

Altro esempio:

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

- zucchero sintattico per
  `<T1, T2 extends T1> void copyTo(List<T1> dst, List<T2> src);`

? vs Object

? Tipo particolare anonimo

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

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

- possiamo istanziare **?** con un tipo qualunque: **Object**, **String**, …
- **List<Object>** è più restrittivo: non posso passare un **List<String>**

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

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

Too good to be true: type erasure

Tutti i tipi generici sono trasformati in **`Object`** nel processo di compilazione

- motivo: backward-compatibility con il codice vecchio
- morale: a runtime, tutte le istanziazioni generiche hanno lo stesso tipo

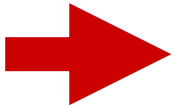
```
List<String>  lst1 = new ArrayList<String>( );
List<Integer> lst2 = new ArrayList<Integer>( );
lst1.getClass( ) == lst2.getClass( )  // true
```

getClass() restituisce il tipo effettivo

Esempio

Il compilatore usa i generici per i controlli statici e poi li elimina dal codice (non esistono nel bytecode):

```
class Vector<T> {
  T[] v; int sz;
  Vector() {
    v = new T[15];
    sz = 0;
  }
  <U extends Comparator<T>>
  void sort(U c) {
    …
    c.compare(v[i], v[j]);
    …
  }
}
…
Vector<Button> v;
v.addElement(new Button());
Button b = v.get(0);
```

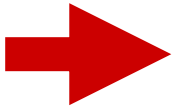


```
class Vector {
  Object[] v; int sz;
  Vector() {
    v = new Object[15];
    sz = 0;
  }

  void sort(Comparator c) {
    …
    c.compare(v[i], v[j]);
    …
  }
}
…
Vector v;
v.addElement(new Button());
Button b = (Button)b.get(0);
```

Esempio

Il compilatore usa i generici per i controlli statici e poi li elimina dal codice (non esistono nel bytecode):

```
class Vector<T> {
  T[] v; int sz;
  Vector() {
    v = new T[15];
    sz = 0;
  }
  <U extends Comparator<T>>
  void sort(U c) {
    …
    c.compare(v[i], v[j]);
    …
  }
}
…
Vector<Button> v;
v.addElement(new Button());
Button b = v.get(0);
```



```
class Vector {
  Object[] v; int sz;
  Vector() {
    v = new Object[15];
    sz = 0;
  }

  void sort(Comparator c) {
    …
    c.compare(v[i], v[j]);
    …
  }
}
…
Vector v;
v.addElement(new Button());
Button b = (Button)b.get(0);
```

Generici e casting

```
List<?> lg = new ArrayList<String>( );   // ok
List<String> ls = (List<String>) lg;     // warning
```

Dalla documentazione Java: “Compiler gives an unchecked warning, since this is something the run-time system *will not check for you*”

- Il controllo che può fare l'interprete a runtime è solo che **`lg`** sia una lista (**`List`**) non una lista di stringhe (**`List<String>`**)

Generici e casting

```
import java.util.ArrayList;

public class Prova<E extends Number> {
    public static void main(String[] args) {
        ArrayList<String> a = new ArrayList<String>();
        a.add("ciao");
        ArrayList<Integer> b = magic(a);  // fa magicamente il cast...
        b.add(100);
        System.out.println(b.get(0));  // stampa "ciao"
        System.out.println(b.get(1));  // stampa 100
    }

    public static ArrayList<Integer> magic(ArrayList<?> a) {
        return (ArrayList<Integer>) a;
    }

}
```

Generici e casting

```
import java.util.ArrayList;

public class Prova<E extends Number> {
    public static void main(String[] args) {
        ArrayList<String> a = new ArrayList<String>();
        a.add("ciao");
        ArrayList<Integer> b = magic(a);  // fa magicamente il cast...
        b.add(100);
        System.out.println(b.get(0));  // stampa "ciao"
        System.out.println(b.get(1));  // stampa 100
    }

    public static ArrayList<Integer> magic(ArrayList<?> a) {
        return (ArrayList<Integer>) a;
    }

}
```

Il fatto che b contenga elementi di tipo diverso mostra che in realtà è di tipo ArrayList<Object> (type erasure)

Il compilatore segnala un warning (unchecked cast)

Java Generics (JG), considerazioni finali

- Il compilatore verifica l'utilizzo corretto dei generici
- I parametri di tipo sono eliminati nel processo di compilazione e il "class file" risultante dalla compilazione è un normale class file senza poliformismo parametrico
- JG aiutano a migliorare il polimorfismo della soluzione
- Limite principale: il tipo effettivo è perso a runtime a causa della type erasure