Java Generics

Java: Interfacce e astrazione

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

… e **ListOfStrings** e …

```
// 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>>**
- …

Parametri e parametri di tipo

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

- Dichiarazione del parametro formale
- Istanziato con una espressione che ha il tipo richiesto `(lst.add(new Integer(7)))`
- Tipo di `add`: `Integer -> boo`

```
interface List<E> {
  boolean add(E elt);
  E get(int index);
}
```

- Dichiarazione di un parametro di tipo
- Istanziabile con un qualunque tipo
  - `List<Integer>`

Una coda

```
class CircleQueue {
  private Circle[] circles;
    :
  public CircleQueue(int size) {...}
  public boolean isFull() {...}
  public boolean isEmpty() {...}
  public void enqueue(Circle c) {...}
  public Circle dequeue() {...}
}
```

Manca la specifica che possiamo immaginare: TDA standard

Una seconda coda

```
class PointQueue {
  private Point[] points;
    :
  public PointQueue(int size) {...}
  public boolean isFull() {...}
  public boolean isEmpty() {...}
  public void enqueue(Point p) {...}
  public Point dequeue() {...}
}
```

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

Una terza coda

```
class ObjectQueue {
  private Object[] objects;
    :
  public ObjectQueue(int size) {...}
  public boolean isFull() {...}
  public boolean isEmpty() {...}
  public void enqueue(Object o) {...}
  public Object dequeue() {...}
}
```

Come possiamo usare la (terza) coda?

Usiamo la (terza) coda

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


 :
```

Altre operazioni

Prendere un oggetto instanza di **`Circle`** è leggermente complicato

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

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

Errore di compilazione senza cast esplicito non possiamo assegnare una espressione con tipo statico **`Object`** ad una variabile di tipo dinamico **`Circle`**

Alternativamente

```
Object o = cq.dequeue();

if (o instanceof Circle ){
    Circle c = (Circle) o;
}
```

Solleviamo **`ClassCastException`** se la coda contenesse oggetti che non sono di tipo **`Circle`** !!!!

Un altro problema

```
ObjectQueue cq = new ObjectQueue(10);
cq.enqueue(new Circle(new Point(0, 0), 10));
cq.enqueue(new Circle(new Point(1, 1), 5));
cq.enqueue(new Point(1, 7));
```

Il codice funziona ma il tipo degli elementi non e' omogeneo
(sub-type polymorfism vs parametric polymorfism)

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() {...}
}
```

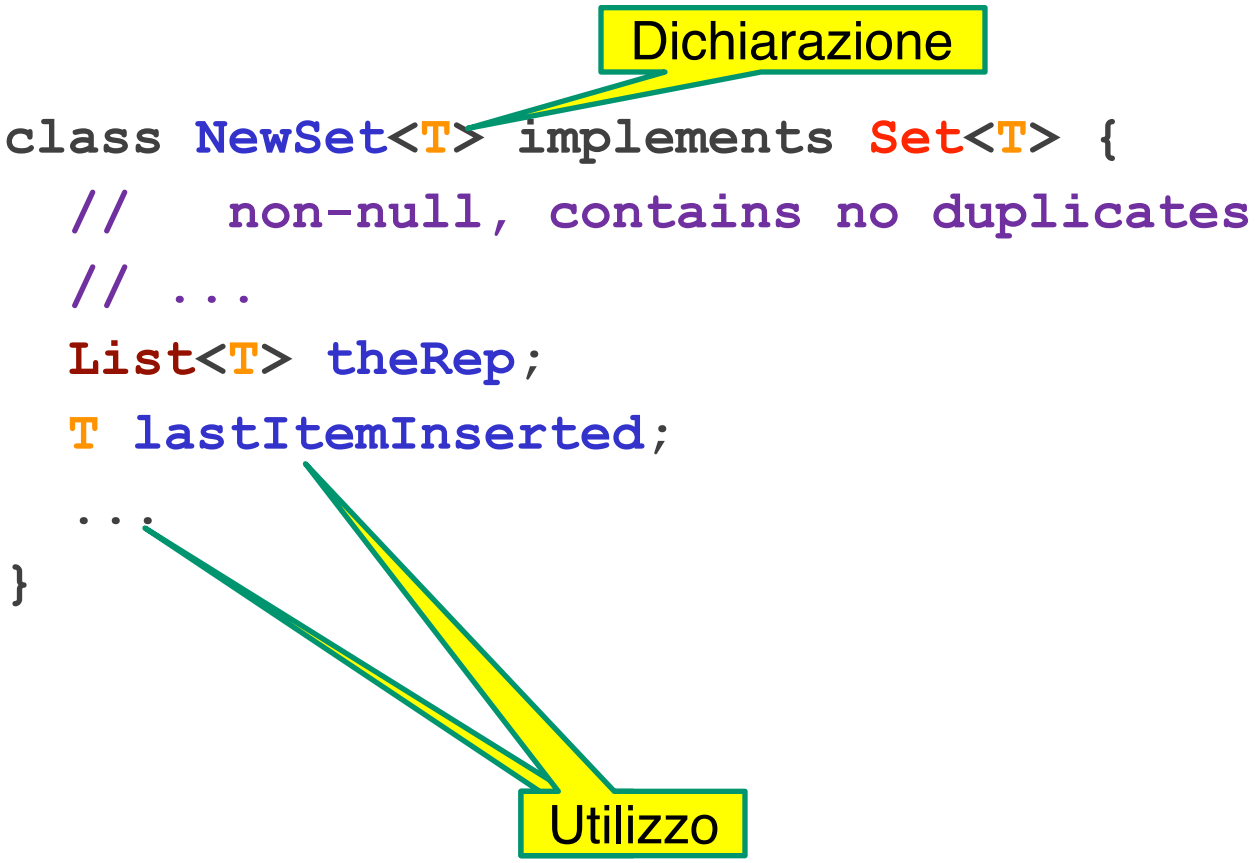
```
Queue<Circle> cq = new Queue<Circle>(10);
cq.enqueue(new Circle(new Point(0, 0), 10));
cq.enqueue(new Circle(new Point(1, 1), 5));
Circle c = cq.dequeue();
```

Da notare che…..

```
Queue<Circle> cq = new Queue<Circle>(10);
cq.enqueue(new Circle(new Point(0, 0), 10));
cq.enqueue(new Point(1, 1));
```

Errore a tempo di compilazione!!!!

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>
```

Istanziare tipi

```
boolean add1(Object elt);
boolean add2(Number elt);
add1(new Date( ));  // OK
add2(new Date( ));  // compile-time error

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
```

Limite superiore gerarchia

Visione effettiva dei generici

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

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

Istanziazione identica

```
Name<Type1, …, TypeN>
```

• Ma compile-time error se il tipo non è un sottotipo del corrispondente limite superiore della gerarchia

Usiamo le variabili di tipo

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

- concettualmente questo corrisponde a forzare una 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 super-tipo o uno dei suoi sotto-tipi

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

- Multiple upper bounds

`<TypeVar super SubType>`

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

Esempio

```
// strutture di ordine su alberi
public class TreeSet<T extends Comparable<T>> {
     ..}
```

Un esempio

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

Metodi generici

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

Esempio

```
class Utils {
  static double sumList(List<Number> lst) {
      double result = 0.0;
      for (Number n : lst)
           result += n.doubleValue( );

      return result;
  }
  static Number choose(List<Number> lst) {
      int i = ... // numero random < lst.size
      return lst.get(i);
  }
}
```

Soluzione efficace

```
class Utils {
  static <T extends Number>
  double sumList(List<T> lst) {
      double result = 0.0;
      for (Number n : lst) // anche T andrebbe bene
        result += n.doubleValue();

      return result;
  }
  static <T>
  T choose(List<T> lst) {
      int i = … // random number < lst.size
      return lst.get(i);
  }
}
```

Dichiarare
i vincoli sui tipi

Dichiarare
i vincoli sui tipi

Generici e la nozione di sotto-tipo

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

Quali sono le regole di Java?

Se Type1 e Type3 sono differenti, anche se Type2 è un sottotipo di Type3, Type1 < Type2 > non è un sottotipo di Type1 < Type3 >

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

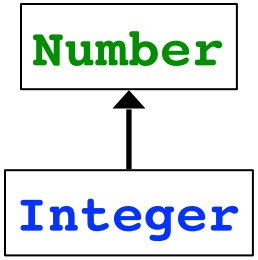
Esempi (da Java)

- Tipi e sottotipi
  - **`ArrayList<E>`** è un sottotipo di **`List<E>`**
  - **`List<E>`** è un sottotipo di **`Collection<E>`**
- **`Integer`** è un sottotipo di **`Number`** ma

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

List<Number> e List<Integer>

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



type `List<Number>` caratterizzata

```
boolean add(Number elt);
Number get(int index);
```

type `List<Integer>` caratterizzata

```
boolean add(Integer elt);
Integer get(int index);
```

`List<Integer>` non è un sottotipo di `List<Number>`!!!!

Esempio: `addAll`

```
interface Set<E> {
   // Aggiunge a this tutti gli elementi di c
   // (che non appartengono a this)
  void addAll(??? c);
}
```

Quale è il miglior tipo per il parametro formale?

- Il più ampio possibile …
- … che permette di avere implementazioni corrette

addAll

```
interface Set<E> {
   // Aggiunge a this tutti gli elementi di c
   // (che non appartengono a this)
  void addAll(??? c);
}
```

Una prima scelta è `void addAll(Set<E> c);`

Troppo restrittivo

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

addAll

```
interface Set<E> {
   // Aggiunge a this tutti gli elementi di c
   // (che non appartengono a this)
  void addAll(??? c);
}
```

Secondo tentativo: `void addAll(Collection<E> c);`

Troppo restrittivo

- il parametro attuale di tipo `List<Integer>` non va bene anche se `addAll` ha solo bisogno di leggere da c e non di modificarlo!!!
- questa è la principale limitazione della nozione di invarianza per i generici in Java

addAll

```
interface Set<E> {
    // Aggiunge a this tutti gli elementi di c
    // (che non appartengono a this)
  void addAll(??? c);
}
```

Proviamo ancora

```
<T extends E> void addAll(Collection<T> c);
```

Idea buona: un parametro generico ma vincolato

- `addAll` non può vedere nell'implementazione il tipo `T`, sa solo che è un sottotipo di `E`, e non può modificare la collection c

Un altro esempio

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

La soluzione va bene, ma ancora meglio

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

Wildcard

Sintassi delle wildcard

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

wildcard = una variabile di tipo anonima

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

`?` 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: `List<String>`non va bene

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

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

Wildcard

Quando si usano le wildcard?

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

Raw types

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

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

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

Type erasure

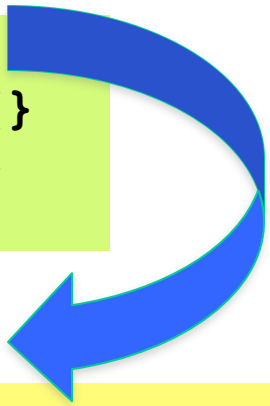
Tutti i tipi generici sono trasformati in `Object` nel processo di compilazione

- 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
```

Type erasure: problemi

```
class A {
  void foo(Queue<Circle> c) {}
  void foo(Queue<Point> c) {}
}
```

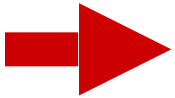


Type erasure porta a errore di compilazione

```
class A {
  void foo(Queue c) {}
  void foo(Queue c) {}
}
```

Esempio

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



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

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

Generici e Cast: `equals`

```
class Node<E> {
  ...
  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof Node<E>))
      return false;

    Node<E> n = (Node<E>) obj;
    return this.data( ).equals(n.data( ));
  }
 }
```

Java Generics (JG)

- Il compilatore verifica l'utilizzo corretto dei generici
- I parametri di tipo sono eliminati nel processo di compilazione e il "class file" risultante dalla compilazione è un normale class file senza polimorfismo parametrico

Considerazioni

- JG aiutano a migliorare il polimorfismo della soluzione
- Limite principale: il tipo effettivo è perso a runtime a causa della type erasure
- Tutte le istanze sono identificate
- Esistono altre implementazioni dei generici per Java