Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Polimorfismo verticale vs. orizzontale
Introduzione ai tipi generici

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

IL TEMA DELLA GENERICITÀ

- Avere `Object` come classe base di ogni altra è servito storicamente anche a scrivere componenti software *generici rispetto al tipo*
  - strutture dati (array, liste, alberi, tabelle..) usabili su qualunque tipo di oggetto *[ma definite una sola volta]*
  - funzioni / metodi che accettano qualunque tipo di oggetto
- Nel tempo tale approccio ha però mostrato *evidenti limiti*
  - usare `Object` come tipo significa *abolire il controllo di tipo*
  - poiché «everything is an Object», in quelle strutture si può in realtà mettere *di tutto* → error-prone, problemi, esplosioni
- L'approccio moderno (da Java 5 in poi) è quindi diverso e basato sull'idea di *tipo parametrico («generico»)*

IL TEMA DELLA GENERICITÀ

- Avere `Object` come classe base di ogni altra è servito storicamente anche a scrivere componenti software *generici rispetto al tipo*
  - strutture dati (array, liste, alberi, ...) di qualunque tipo di oggetto
  - funzioni / metodi che accettano qualunque tipo di oggetto
- Nel tempo tale approccio ha però mostrato *evidenti limiti*
  - usare `Object` come tipo significa *abolire il controllo di tipo*
  - poiché «everything is an Object», in quelle strutture si può in realtà mettere *di tutto* → error-prone, problemi, esplosioni
- L'approccio moderno (da Java 5 in poi) è quindi diverso e basato sull'idea di *tipo parametrico («generico»)*

Ma *non tipi primitivi*, perché non sono oggetti! (l'ennesima tassa dovuta alla loro disuniformità..)

FUNZIONI (LIBRERIE) GENERICHE con `Object`

- Si supponga di voler scrivere una funzione che *stampi tutti gli elementi di un (generico) array*
- Perché funzioni per qualunque array, si può pensare di definirla in modo che accetti un *array di* ***`Object`***:

```java
public static void printAll(Object[] array){
   for (Object obj : array) System.out.println(obj);
}
```

Java

- Concretamente, sarà poi chiamata con *specifici array*

```java
String[]    arr1 = { "Pippo", "Alberto", "Enrico" };
Counter[]   arr2 = { new Counter(2), new Counter2(18)};
Frazione[]  arr3 = { new Frazione(2,3),new Frazione(5,7)};

printAll(arr1); printAll(arr2); printAll(arr3);
```

Java

FUNZIONI (LIBRERIE) GENERICHE con `Object`

- PROBLEMA: *l'intenzione* era «chiaramente» quella di passare un array specifico, contenente *oggetti omogenei*
- MA tecnicamente in un array di **`Object`** ci si potrebbe mettere *qualunque cosa*, *violando l'idea stessa di array come collezione di entità omogenee in tipo!*
- Ad esempio, nulla vieterebbe di fare questo:

Java

```
Object[] arr =
        { "Pippo", new Counter(2), new Frazione(2,3)};
printAll(arr);
```

- Tecnicamente ha senso e funziona, ma *equivale ad abolire il controllo di tipo:* dentro ad **`arr`** ci può essere qualunque «porcheria» e il compilatore non può obiettare! ☹☹

FUNZIONI (LIBRERIE) GENERICHE con `Object`

- A ben vedere, lo stesso problema era emerso tempo addietro, cercando di rendere generica la funzione **`idem`**:

Java ~C#

```
public static boolean idem(Object[] a, Object[] b){
   if (a.length != b.length) return false;
   for (int i=0; i<a.length; i++){
       if (!a[i].equals(b[i])) return false;
   }
   return true;
}
```

- L'uso inteso era chiamarla con due array *dello stesso tipo*:

```
System.out.println(idem(
  new Counter[]{new Counter(2),new Counter(3),new Counter(4)},
  new Counter[]{new Counter(2),new Counter(3),new Counter(4)} ));

System.out.println(idem(
  new String[]{"Pippo", "Alberto", "Enrico" },
  new String[]{"Denari", "Coppe", "Bastoni" } ));
```

FUNZIONI (LIBRERIE) GENERICHE con `Object`

- A ben vedere, lo stesso problema era emerso tempo addietro, cercando di rendere generica la funzione **`idem`**:

Java

~C#

```
public static boolean idem(Object[] a, Object[] b){
   if (a.length != b.length) return false;
   for (int i=0; i<a.length; i++){
       if (!a[i].equals(b[i])) return false;
   }
   return true;
}
```

- MA nulla poteva impedire di chiamarla con array disuniformi:
  - la chiamata è lecita e il risultato è certamente *false*
  - ma è insensato: due array di diverso tipo non sono mai uguali

```
System.out.println(idem(
  new Counter[]{new Counter(2),new Counter(3),new Counter(4)},
  new String[]{"Denari", "Coppe", "Bastoni" } ));
```

STRUTTURE DATI GENERICHE con `Object`

- Oltre agli array, si potrebbero desiderare altri tipi di strutture dati «generiche», come liste, alberi, tabelle, etc.
- Definirle come «contenitori di **`Object`**» permette di ottenere quanto voluto, *ma al prezzo – di nuovo – di poterci poi mettere dentro qualunque cosa*, violando così l'uniformità.
- Per meglio focalizzare la questione, definiamo e usiamo una classica lista «old style» a puntatori: **`edutils.List`**

STRUTTURE DATI GENERICHE con `Object`

- Esempio: la lista generica **`edutils.List`**

Java

```
package edutils;

public class List {
 private Object elem;
 private List next;

 public List(){ elem=null; next=null;}
 public List(Object e){ elem=e; next=null; }
 public List(Object e, List l){ elem=e; next=l; }

 public Object head(){ return elem; }
 public List tail(){ return next; }

 public boolean isEmpty(){ return elem==null; }

 public String toString(){ return isEmpty()? "" :
       elem.toString() +
       (next==null ? "" : ", " + next.toString()); }
}
```

Il tipo effettivo dell'elemento può essere *un qualsiasi Object* (ma non un tipo primitivo, perché essi non sono oggetti)

STRUTTURE DATI GENERICHE con `Object`

- Uso inteso: poter creare specifiche liste di ciò che ci serve

Java

```
List l1 = new List();
List l2 = new List("Pippo",
              new List("Alberto",
                  new List("Enrico")) );
List l3 = new List( new Counter(2),
              new List(new Counter2(18)) );
List l4 = new List( new Frazione(2,3),
              new List(new Frazione(5,7)) );

System.out.println(l1);
System.out.println(l2);
System.out.println(l3);
System.out.println(l4);
```

Lista vuota

Lista di stringhe

Lista di Counter

Lista di frazioni

```
Pippo, Alberto, Enrico
Counter di valore 2, Counter2 di valore 18
2/3, 5/7
```

STRUTTURE DATI GENERICHE con `Object`

- Ma purtroppo è legale anche un mix come questo:

Java

```
List l5 = new List( new Frazione(2,3),
                new List("Pippo",
                    new List(new Counter2(18))) );
System.out.println(l5);
```

```
Pippo, Alberto, Enrico
Counter di valore 2, Counter2 di valore 18
2/3, 5/7
2/3, Pippo, Counter2 di valore 18
```

- Di nuovo, definire strutture come «contenitori di `Object`» permette sì di ottenere contenitori «generici» (bene!)…
- …ma anche, purtroppo, di poterci poi mettere *dentro qualunque cosa (male!)*, violando così l'uniformità di tipo, che solitamente costituisce un requisito inteso

GENERICITÀ: UN PRIMO BILANCIO

- L'uso di `Object` come mezzo per scrivere componenti software *generici rispetto al tipo* è dunque un approccio *insoddisfacente, error-prone* e per questo *obsoleto*
  - si bypassa di fatto il controllo di tipo
  - si pongono le premesse per «esplosioni» a run-time dovute a errori non intercettati in fase di compilazione
- Il modo moderno per soddisfare la necessità di *genericità* è costituita dal concetto di *tipo parametrico (o generico)*
  - il tipo c'è, ma diventa un *parametro*
  - in tal modo sarà possibile *specificarlo al momento dell'uso* della funzione o del componente
  - RISULTATO: genericità senza compromessi ☺

IL TIPO PARAMETRICO (GENERICO)

- La nozione di *tipo generico* consente di *usare un simbolo per il tipo,* che diviene un *parametro* da specificare poi
  - in Java, C#, Kotlin: notazione **<T>**
  - in Scala: notazione **[T]**

L'identificatore può essere qualsiasi: i più usati sono **T** («Type») ed **E** («Element»)

Java

NON PIÙ

```
public static boolean idem(Object[] a, Object[] b)
```

MA

```
public static <T> boolean idem(T[] a, T[] b)
```

- Idea di fondo
  - non stabiliamo ora chi sia **T** (lo si dirà al momento dell'uso)
  - ma affermiamo che sarà comunque *lo stesso* per *entrambi gli argomenti:* non potranno più essere di tipi diversi!

CONFRONTO

- Finora, i due argomenti avevano come (unico) vincolo il fatto di essere *array di* **`Object`**
- L'idea era che fossero poi «*dello stesso*» tipo, ma questo vincolo non era in realtà specificato da nessuna parte
  - era un vincolo "farlocco": ogni array è «di oggetti»
  - nei fatti, il contenuto poteva essere qualunque cosa
- Ora invece i due argomenti hanno il vincolo di essere entrambi array della *stessa* cosa (**`T`**)
  - qualunque tipo sia/sarà T, i due array saranno *omogenei*
  - pasticci come quelli di prima diventano impossibili
  - la cura è *rafforzare* il controllo di tipo, *non eluderlo!*
  - il compilatore è un *alleato*, non un nemico!

POLIMORFISMO VERTICALE vs POLIMORFISMO ORIZZONTALE

- L'approccio «old style» che sfrutta `Object` per ottenere genericità si chiama *polimorfismo verticale*
  - segue la gerarchia di ereditarietà
  - sembra semplice, e lo è…. in effetti, è *fin troppo semplice!*
- L'approccio basato sul tipo generico si chiama invece *polimorfismo orizzontale*
  - opera "trasversalmente" rispetto alla gerarchia di ereditarietà
  - introduce il tipo come parametro
  - non influenza l'ereditarietà, non si basa su essa
  - è *l'approccio moderno* usato ormai "a tappeto" in tutti i linguaggi a oggetti con type system forti

SINTASSI: DEFINIZIONE

DEFINIZIONE di funzione generica in tipo

Java

```
public static <T> boolean idem(T[] a, T[] b)
```

C#

```
public static bool idem<T>(T[] a, T[] b)
```

Scala

```
def idem[T](a: Array[T], b: Array[T]) : Boolean
```

Kotlin

```
public fun <T> idem(a: Array<T>, b: Array<T>):Boolean
```

- i vari linguaggi si differenziano per la posizione del segnaposto **<T>** (o **[T]** in Scala) e dei relativi dettagli sintattici

SINTASSI: CHIAMATA

CHIAMATA di una funzione generica (con **T=Counter**)

Java

```
nomeclasse.<Counter>idem(array1, array2)
```

C#

```
nomeclasse.idem<Counter>(array1, array2)
```

Scala

```
nomeobject.idem(array1, array2)
```

Kotlin

```
nomeobject.idem(array1, array2)
```

Importante: in Java, omettere l'indicazione del tipo attuale `<Counter>` fa sì che venga assunto `<Object>`, *disattivando così di fatto il type check* !

- anche in fase di chiamata i vari linguaggi si differenziano per la posizione del segnaposto **<T>** (o **[T]**) e dei relativi dettagli sintattici

ESPERIMENTO in Java

Java

```
public class MyLib {
  public static <T> boolean idem(T[] a, T[] b){
    if (a.length != b.length) return false;
    for (int i=0; i<a.length; i++)
        if (!a[i].equals(b[i])) return false;
    return true;
  }
}
```

Java

```
public static void main(String[] args) {
  Counter[] x = {new Counter(2), new Counter(3), new Counter(4)};
  Counter[] y = {new Counter(2), new Counter(3), new Counter(4)};
  Counter[] z = {new Counter(2), new Counter(3), new Counter(5)};
  Counter[] w = {new Counter(2), new Counter(3)};

  System.out.println(MyLib.<Counter>idem(x,y));
  // true, se Counter ha equals polimorfa

  System.out.println(MyLib.<Counter>idem(x,z)); // false
  System.out.println(MyLib.<Counter>idem(z,y)); // false
  System.out.println(MyLib.<Counter>idem(x,w)); // false
}
```

OK: chiamata corretta

ESPERIMENTO in Java (BIS)

Java

```
public class MyLib {
  public static <T> boolean idem(T[] a, T[] b){
    if (a.length != b.length) return false;
    for (int i=0; i<a.length; i++)
        if (!a[i].equals(b[i])) return false;
    return true;
  }
}
```

Java

```
public static void main(String[] args) {
  Counter[] x = {new Counter(2), new Counter(3), new Counter(4)};
  Counter[] y = {new Counter(2), new Counter(3), new Counter(4)};
  Counter[] z = {new Counter(2), new Counter(3), new Counter(5)};
  Counter[] w = {new Counter(2), new Counter(3)};

  System.out.println(MyLib.idem(x,y));
  // true, se Counter ha equals polimorfa

  System.out.println(MyLib.idem(x,z)); // false
  System.out.println(MyLib.idem(z,y)); // false
  System.out.println(MyLib.idem(x,w)); // false
}
```

ATTENZIONE:
chiamata non qualificata
→ *pericolosa!*

ESPERIMENTO in Java (TER)

Java

```
public class MyLib {
  public static <T> boolean idem(T[] a, T[] b){
    if (a.length != b.length) return false;
    for (int i=0; i<a.length; i++)
        if (!a[i].equals(b[i])) return false;
    return true;
  }
}
```

Java

```
public static void main(String[] args) {
  Counter[] x = {new Counter(2), new Counter(3), new Counter(4)};
  Counter[] y = {new Counter(2), new Counter(3), new Counter(4)};
  Counter[] z = {new Counter(2), new Counter(3), new Counter(5)};
  String[]  q = {"ciao", "mondo"};
  System.out.println(MyLib.idem(x,q));
  // falso, ma perché compila?
}
```

DISASTRO!
Senza il type tag l'errore *non* viene rilevato, perché il compilatore ha assunto `<Object>`, *disattivando così di fatto il type check*

ESPERIMENTO in C#

C#

```
public class MyLib {
 public static bool idem<T>(T[] a, T[] b){
  if (a.Length != b.Length) return false;
  for (int i=0; i<a.Length; i++)
        if (!a[i].Equals(b[i])) return false;
  return true;
 }
}
```

C#

```
public static void Main(string[] args){
  Counter[] x = {new Counter(2), new Counter(3), new Counter(4)};
  Counter[] y = {new Counter(2), new Counter(3), new Counter(4)};
  Counter[] z = {new Counter(2), new Counter(3), new Counter(5)};
  Counter[] w = {new Counter(2), new Counter(3)};

  System.out.println(MyLib.idem<Counter>(x,y));
  // True, se Counter ha Equals polimorfa

  System.out.println(MyLib.idem<Counter>(x,z)); // False
  System.out.println(MyLib.idem<Counter>(z,y)); // False
  System.out.println(MyLib.idem<Counter>(x,w)); // False
}
```

OK: chiamata corretta

ESPERIMENTO in C# (BIS/TER)

C#

```
public static void Main(string[] args){
  Counter[] x = {new Counter(2), new Counter(3), new Counter(4)};
  Counter[] y = {new Counter(2), new Counter(3), new Counter(4)};
  Counter[] z = {new Counter(2), new Counter(3), new Counter(5)};
  Counter[] w = {new Counter(2), new Counter(3)};
  System.out.println(MyLib.idem(x,y));
  System.out.println(MyLib.idem(x,z));
  System.out.println(MyLib.idem(z,y));
  System.out.println(MyLib.idem(x,w));
}
```

In C#, la chiamata non qualificata giustamente NON COMPILA

Impossibile dedurre gli argomenti di tipo per il metodo MyLib.idem<T>(T[], T[])' dall'uso.

C#

```
public static void Main(string[] args){
  Counter[] x = {new Counter(2), new Counter(3), new Counter(4)};
  Counter[] y = {new Counter(2), new Counter(3), new Counter(4)};
  Counter[] z = {new Counter(2), new Counter(3), new Counter(5)};
  Counter[] w = {new Counter(2), new Counter(3)};
  String[]  q = {"ciao", "mondo"};
  System.out.println(MyLib.idem<Counter>(x,y));
}
```

Mischiando tipi di oggetti diversi *non compila*, come è giusto

ESPERIMENTO in Scala

Scala

```
def idem[T](a: Array[T], b: Array[T]) : Boolean = {
  if (a.size != b.size) return false;
  for (i <- 0 until a.size)
        if (!a(i).equals(b(i))) return false;
  return true;
}
```

Scala

```
def main(args: Array[String]) : Unit = {
  val x = Array(new Counter(2),new Counter(3),new Counter(4));
  val y = Array(new Counter(2),new Counter(3),new Counter(4));
  val z = Array(new Counter(2),new Counter(3),new Counter(5));
  val w = Array(new Counter(2), new Counter(3));
  println(idem(x,y)); // true, se Counter ha equals polimorfa
  println(idem(x,z)); // false
  println(idem(z,y)); // false
  println(idem(x,w)); // false
}
```

In Scala la chiamata non richiede qualificatori: il compilatore non fa assunzioni su **`Object`**
Eventuali usi errati, come **`idem(x,q)`**, vengono identificati a compile time

ESPERIMENTO in Kotlin

Kotlin

```
public fun <T> idem(a: Array<T>, b: Array<T>) : Boolean {
  if (a.size != b.size) return false;
  for (i in 0..a.size-1)
        if (!a[i]!!.equals(b[i])) return false;
  return true;
}
```

`!!` non-null assertion operator: converte un valore *eventualmente nullo* in uno *certamente non nullo*, lanciando eccezione (NPE) in caso sia nullo

Kotlin

```
public fun main(args: Array<String>) : Unit {
  val x = arrayOf(Counter(2), Counter(3), Counter(4));
  val y = arrayOf(Counter(2), Counter(3), Counter(4));
  val z = arrayOf(Counter(2), Counter(3), Counter(5));
  val w = arrayOf(Counter(2), Counter(3));

  println(idem(x,y)); // true, se Counter ha equals polimorfa
  println(idem(x,z)); // false
  println(idem(z,y)); // false
  println(idem(x,w)); // false
}
```

In Kotlin la chiamata non richiede qualificatori: il compilatore non fa assunzioni su `Object`
Eventuali usi errati, come `idem(x,q)`, vengono identificati a compile time

STRUTTURE DATI GENERICHE CON TIPO PARAMETRICO

- Il tipo parametrico si può usare anche per definire nuove strutture dati: liste, alberi, tabelle…
- Non più «contenitori di **`Object`**» MA «contenitori di **`T`**»
  - rimane possibile specializzarli per qualunque tipo concreto **`T`**
  - *MA diventa impossibile poterci mettere dentro qualunque cosa senza controllo*, violando l'uniformità di tipo
- Revisioniamo quindi la precedente definizione di lista a puntatori (**`edutils.List`**) usando il tipo parametrico come tipo dell'elemento, al posto di **`Object`**

STRUTTURE DATI GENERICHE CON `Object`

- Lista generica **`edutils.List`** – vecchia versione

Java

```
package edutils;

public class List {
 private Object elem;
 private List next;

 public List(){ elem=null; next=null;}
 public List(Object e){ elem=e; next=null; }
 public List(Object e, List l){ elem=e; next=l; }

 public Object head(){ return elem; }
 public List tail(){ return next; }

 public boolean isEmpty(){ return elem==null; }

 public String toString(){ return isEmpty()? "" :
       elem.toString() +
       (next==null ? "" : ", " + next.toString()); }
}
```

Gli elementi sono degli **`Object`**

STRUTTURE DATI GENERICHE CON TIPO PARAMETRICO

- Lista generica **`edutils2.List`** – nuova versione

Java

```
package edutils2;

public class List<T> {
 private T elem;
 private List<T> next;

 public List(){ elem=null; next=null;}
 public List(T e){ elem=e; next=null; }
 public List(T e, List<T> l){ elem=e; next=l; }

 public T head(){ return elem; }
 public List<T> tail(){ return next; }

 public boolean isEmpty(){ return elem==null; }

 public String toString(){ return isEmpty()? "" :
       elem.toString() +
       (next==null ? "" : ", " + next.toString()); }
}
```

La lista è ora definita esplicitamente come lista «di **T**»

Tipo parametrico **T** al posto di **Object**

NB:nei costruttori non è previsto il tag **<T>**

STRUTTURE DATI GENERICHE CON TIPO PARAMETRICO

- L'uso inteso rimane possibile, ma ora si deve specificare esattamente *che tipo di liste si desideri, caso per caso:*

```
List<Object> l1 = new List<Object>();
List<String> l2 = new List<String>("Pippo",
                  new List<String>("Alberto",
                      new List<String>("Enrico")) );
List<Counter> l3 = new List<Counter>( new Counter(2),
                  new List<Counter>(new Counter2(18)));
List<Frazione> l4 = new List<Frazione>(
                  new Frazione(2,3),
                  new List<Frazione>(new Frazione(5,7)) );
System.out.println(l1);
System.out.println(l2);
System.out.println(l3);
System.out.println(l4);
```

Qui il tipo non importa

Java

Lista di stringhe

Lista di Counter

Lista di frazioni

```
Pippo, Alberto, Enrico
Counter di valore 2, Counter2 di valore 18
2/3, 5/7
```

STRUTTURE DATI GENERICHE CON TIPO PARAMETRICO

- Il mix incontrollato non è ora più possibile, perché le liste così ottenute sono *incompatibili* fra loro

```
l2 = l3; // NO!
l2 = l4; // NO!
l4 = l2; // NO!
```

- Per poter mettere oggetti «qualsiasi» (disomogenei) in una lista ora bisogna etichettarla *esplicitamente* come tale:

Java

```
List<Object> l5 = new List<Object>(
                      new Frazione(2,3),
                      new List<Object>("Pippo",
                        new List<Object>(new Counter2(18))
                      );

System.out.println(l5);
```

```
Pippo, Alberto, Enrico
Counter di valore 2, Counter2 di valore 18
2/3, 5/7
2/3, Pippo, Counter2 di valore 18
```

CONCLUSIONE

- Scrivere funzioni o strutture dati *generiche in tipo* è possibile
- Il modo «old style» di farlo è etichettare gli argomenti come `Object` → *polimorfismo verticale*
  - di fatto abolisce il controllo di tipo
  - i *mis-use case sfuggono a compile time ed esplodono a run time*
- Il modo moderno è adottare il *tipo parametrico (generico)* → *polimorfismo orizzontale*
  - rende il tipo stesso un (ulteriore) parametro
  - rafforza il controllo di tipo
  - intercetta i mis-use case a *compile time*
- Ne riparleremo in futuro, parlando di *varianza*