Dipartimento di Informatica e Sistemistica
Antonio Ruberti

“Sapienza” Università di Roma

Ereditarietà

Corso di Tecniche di Programmazione

Laurea in Ingegneria Informatica

(Canale di Ingegneria delle Reti e dei Sistemi Informatici - Polo di Rieti)

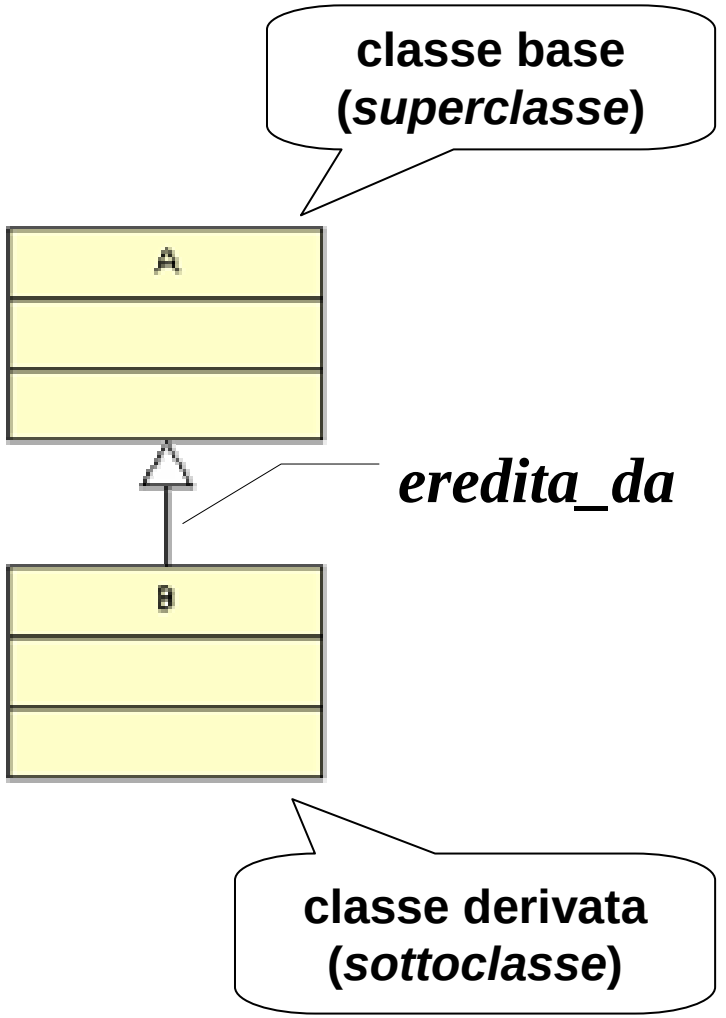
Anno Accademico 2007/2008

Prof. Paolo Romano

Si ringrazia il Prof. Enrico Denti per aver reso
disponibile il proprio materiale didattico sul quale si basano queste slides

EREDITARIETÀ

Una *relazione tra classi:*
si dice che
la nuova classe B
eredita da
la pre-esistente
classe A



EREDITARIETÀ

- **La nuova classe ESTENDE una classe già esistente**
  - **può aggiungere *nuovi dati o metodi***
  - **può accedere ai dati ereditati *purché il livello di protezione lo consenta***
  - **NON può eliminare dati o metodi.**
- **La classe derivata condivide *la struttura e il comportamento*** (*per le parti non ridefinite*) **della classe base**

EREDITARIETÀ

- **Per poter definire una nuova classe *a partire da una già esistente***
- **Bisognerà dire:**
  - **– *quali dati* la nuova classe *ha in più* rispetto alla precedente**
  - **– *quali metodi* la nuova classe *ha in più* rispetto alla precedente**
  - **– *quali metodi* la nuova classe *modifica* rispetto alla precedente.**

ESEMPIO

Dal contatore (solo in avanti) …

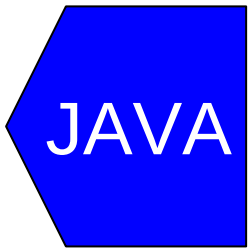
```
public class Counter {
  private int val;
  public Counter() { val = 1; }
  public Counter(int v) { val = v; }
  public void reset() { val = 0; }
  public void inc()   { val++; }
  public int getValue() { return val;}
}
```

Attenzione al-la protezione!

ESEMPIO

… al contatore avanti/indietro (con decremento)

```
public class Counter2 extends Counter {
 public void dec() { val--; }
}
```



Questa nuova classe:

- **eredita da `Counter` il campo `val` (un `int`)**
- **eredita da `Counter` *tutti i metodi***
- **aggiunge a `Counter` il metodo `dec()`**

ESEMPIO

… al contatore avanti/indietro *(con decremento)*

```
public class Counter2 extends Counter {
 public void dec() { val--; }
}
```

JAVA

Ma `val` era privato!!

ERRORE: nessuno può accedere a dati e metodi privati di qualcun altro!

Q

- **`int`)**
- **eredita da `Counter` *tutti i metodi***
- **aggiunge a `Counter` il metodo `dec()`**

EREDITARIETÀ E PROTEZIONE

- **PROBLEMA:**
  il livello di protezione `private` impedisce a chiunque di accedere al dato, *anche a una classe derivata*
  - **va bene per dati "veramente privati"**
  - **ma *è troppo restrittivo* nella maggioranza dei casi**
- **Per sfruttare appieno l'ereditarietà occorre *rilassare un po' il livello di protezione***
  - **senza dover tornare per questo a `public`**
  - **senza dover scegliere per forza la visibilità di package: il concetto di package *non c'entra niente* con l'ereditarietà!**

LA QUALIFICA `protected`

Un dato o un metodo `protected`

- **è come la visibilità di package per chiunque non sia una classe derivata**
- **ma *consente libero accesso* a una classe derivata, indipendentemente dal package (namespace) in cui essa è definita.**

Occorre dunque *cambiare la protezione del campo `val` nella classe `Counter`.*

ESEMPIO

Il contatore “riadattato”...

```
public class Counter {
  protected int val;
  public Counter() { val = 1; }
  public Counter(int v) { val = v; }
  public void reset() { val = 0; }
  public void inc()   { val++; }
  public int getValue() { return val;}
}
```

Nuovo tipo di protezione.

ESEMPIO

… e il contatore con decremento:

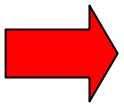
```
public class Counter2 extends Counter {
   public void dec() { val--; }
}
```

Ora funziona !

UNA RIFLESSIONE

La qualifica `protected`:

- **rende accessibile un campo *a tutte le sottoclassi*, presenti *e future***
- **costituisce perciò un *permesso di accesso "indiscriminato"*, valido per *ogni possibile sottoclasse che possa in futuro essere definita*, senza possibilità di distinzione.**



I membri `protected` *sono citati* nella documentazione prodotta da Javadoc (a differenza dei membri qualificati *privati* o con visibilità *package*).

EREDITARIETÀ

Cosa si eredita?

- **tutti i dati della classe base**
  - **anche quelli privati, a cui comunque la classe derivata non potrà accedere direttamente**
- **tutti i metodi...**
  - **anche quelli che la classe derivata non potrà usare direttamente**
- ***... tranne i costruttori*, perché sono specifici di quella particolare classe.**

EREDITARIETÀ E COSTRUTTORI

- **Una classe derivata *non può prescindere dalla classe base*, perché ogni istanza della classe derivata *comprende in sé*, indirettamente, un oggetto della classe base.**
- **Quindi, *ogni costruttore della classe derivata si appoggia a un costruttore della classe base* affinché esso costruisca la "parte di oggetto" relativa alla classe base stessa:**

"ognuno deve costruire ciò che gli compete"

EREDITARIETÀ E COSTRUTTORI

Perché ogni costruttore della classe derivata si deve appoggiare a un costruttore della classe base?

- **solo il costruttore della classe base può sapere come inizializzare i dati ereditati in modo corretto**
- **solo il costruttore della classe base può garantire l'inizializzazione dei dati privati, a cui la classe derivata non potrebbe accedere direttamente**
- **è inutile duplicare nella sottoclasse tutto il codice necessario per inizializzare i dati ereditati, che è già stato scritto.**

Ma cosa vuol dire "appoggiarsi" a un costruttore della classe base?

- **il costruttore della classe derivata *INVOCA AUTOMATICA-MENTE un opportuno costruttore della classe-base***

EREDITARIETÀ E COSTRUTTORI

- ***Ma come può un costruttore della classe derivata invocare un costruttore della classe base?***
  I costruttori non si possono chiamare direttamente!
- **Occorre un modo per dire al costruttore della classe derivata di *rivolgersi al "piano di sopra"***
- **In JAVA, si usa a questo scopo la keyword `super`**
  - **`super` si usa in modo analogo alla keyword `this`**
  - **`this(…)` richiama un altro costruttore della stessa classe**
  - **`super(…)` richiama quello della classe base**

ESEMPIO IN JAVA

Il contatore con decremento:

Costruttore di default *generato automatica-mente* in assenza di altri costruttori

```
public class Coun        extends Counter {
  public void dec() { val--; }
  public Counter2() { super(); }
  public Counter2(int v) { super(v); }
}
```

L'espressione `super(...)` invoca il costruttore della classe base che corrisponde come numero e tipo di parametri alla lista di argomenti fornita.

EREDITARIETÀ E COSTRUTTORI

E se non indichiamo alcuna chiamata a `super(...)`?

- **Il compilatore inserisce automaticamente una chiamata al *costruttore di default* della classe base aggiungendo `super()` o `base()`**
- **In questo caso il costruttore dei default della classe base deve esistere, altrimenti si ha ERRORE.**

EREDITARIETÀ E COSTRUTTORI

RICORDARE: il sistema genera automaticamente il costruttore di default *solo se noi non definiamo alcun costruttore*.

Se c'è *anche solo una definizione di costruttore data da noi*, il sistema assume che noi sappiamo il fatto nostro, e non genera più il costruttore di default automatico.

- **…te**
- **…ella**
- **…se( )**
- **…t della classe base deve esistere, …menti si ha ERRORE.**

`super` : RIASSUNTO

La parola chiave *`super`*

- **nella forma `super(…)`,**
  invoca un costruttore della classe base
- **nella forma `super.val`,**
  consente di accedere al campo `val` della classe base
  (sempre che esso non sia privato)
- **nella forma `super.metodo()` invoca il metodo `metodo()` della classe base**
  (sempre che esso non sia privato)

COSTRUTTORI e PROTEZIONE

- **Di norma, i costruttori sono `public`**
  - **ciò è necessario *affinché chiunque possa istanziare oggetti* di quella classe**
  - **in particolare, è sempre pubblico il costruttore di default generato automaticamente**
- **Possono però esistere classi con tutti i costruttori *non pubblici* (tipicamente, *protected*)**
  - **lo si fa per *impedire di* creare oggetti di quella classe al di fuori di "casi controllati"**
  - **caso tipico: una classe pensata per fungere SOLO da *classe base per altre* da definirsi in futuro (definirà probabilmente solo costruttori *protected*)**

Esempio
Costruttori ed ereditarietà

```
class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}

public class Cartoon extends Drawing {
  public Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
}
```

Cosa Produce in output?

```
class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}

public class Cartoon extends Drawing {
  public Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
}
```

Cosa Produce in output?

Art constructor
Drawing constructor
Cartoon constructor

RIFLESSIONI

- ***Non è stato necessario disporre del codice sorgente* per specializzare il componente**
  - **è bastato il file `.class`**
  - **con idonea documentazione**
- **È stato possibile adattare un componente *di cui non conosciamo il funzionamento interno e che non avremmo saputo costruire***
  - **l'ereditarietà è un mezzo estremamente potente per *riusare e adattare* componenti *anche fatti da altri***
  - **si toccano con mano i vantaggi dell'incapsulamento**

CLASSI E METODI FINALI

- **L'ereditarietà è molto potente, ma a volte occorre poter *impedire* che *una classe* possa in futuro essere estesa**
  - **ad esempio, per proteggersi: le sottoclassi potrebbero accedere a tutti i dati `protected`**
  - **oppure, per impedire il riutilizzo del proprio lavoro**
- **Altre volte, l'esigenza è meno forte: *impedire* che *un singolo metodo* possa in futuro essere sovrascritto**
  - **ad esempio, perché è critico e non si vuole che possa essere modificato in futuro…**
- **Per far fronte a queste situazioni, si può etichettare una classe o un metodo come *finale***

CLASSE FINALE: ESEMPIO

In Java:

```
public final class TheLastCounter
    extends Counter {
    ...
}
```

METODO FINALE: ESEMPIO

In Java:

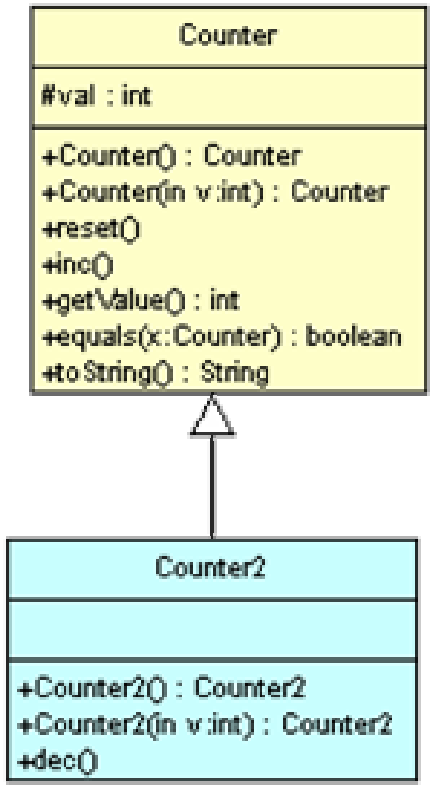
```
public class MyLibrary {
    final int myAlgorithm(..){ ... }
}
```

Ereditarietà: conseguenze

Relazione tipo / sottotipo

EREDITARIETÀ: RIFLESSIONI

- **Se una classe eredita da un'altra, *la classe derivata mantiene l'interfaccia di accesso della classe base***
  - **Naturalmente può *estenderla*, aggiungendo nuovi metodi**
- **Quindi, *ogni `Counter2` è anche un (tipo particolare di) `Counter` !***
- **Ergo, un `Counter2` può essere usato *al posto di un `Counter`* *in modo trasparente al cliente***

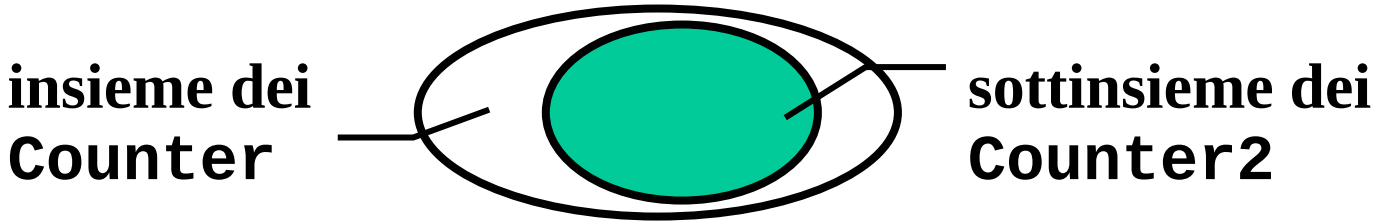


EREDITARIETÀ: RIFLESSIONI

Dire che

- ***ogni `Counter2` è anche un tipo particolare di `Counter`***
- **un oggetto `Counter2` può essere usato *ovunque sia atteso un oggetto `Counter`***

significa dire che l'insieme dei `Counter2` è *un sottoinsieme* dell'insieme dei `Counter`



EREDITARIETÀ: CONSEGUENZE

Poter usare un `Counter2` al posto di un `Counter` significa che:

- **se `c` è un riferimento a `Counter`, si deve poterlo usare per referenziare un'istanza di `Counter2`**
- **se una funzione si aspetta come parametro un `Counter`, si deve poterle passare un'istanza di `Counter2`**
- **se una funzione dichiara di restituire un `Counter`, può in realtà restituire un'istanza di `Counter2`**

Ovviamente, non è vero il viceversa:

- **un `Counter` NON È un tipo particolare di `Counter2`**

EREDITARIETÀ: CONSEGUENZE

Poter usare un `Counter2` al posto di un `Counter` significa che:

```
Counter c = new Counter2(11);
Counter2 c2 = new Counter2(); c = c2;
```

```
void f(Counter x){ ... }
...
f(c2); // c2 è un'istanza di Counter2
```

```
Counter getNewCounter(int v){}
      return new Counter2(v);
}
```

```
Counter c = new Counter2(11);
Counter2 c2 = c; // NO, ERRATO!!!
```

ESEMPIO

```
public class Esempio6 {
 public static void main(String args[]) {
  Counter  c1 = new Counter(10);
  Counter2 c2 = new Counter2(20);
  c1=c2;          // OK: c2 è anche un Counter
  // c2=c1;       // NO: c1 è solo un Counter
 }
}
```

OK perché c2 è un Counter2, *quindi anche implicitamente un Counter* (e c1 è un Counter)

NO, perché c2 è un Counter2 *e come tale esige un suo "pari"*, mentre c1 è "solo" un Counter

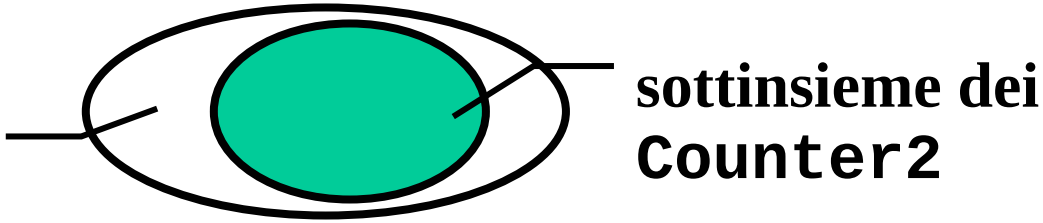
CONSEGUENZE CONCETTUALI

Dire che

l'insieme dei `Counter2` è un sottoinsieme dell'insieme dei `Counter`

induce *una classificazione del mondo*

(aderente alla realtà...?)



Per questo l'ereditarietà è più di un semplice *"riuso di codice": riusa l'astrazione*

CLASSIFICAZIONE DEL "MONDO"

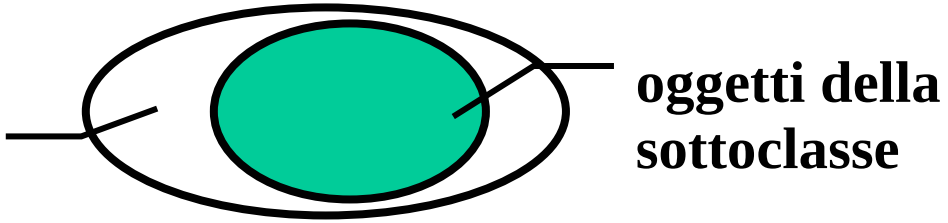
Questa classificazione può essere o meno *aderente alla realtà:*

- **se è aderente alla realtà,**
  - → **rappresenta bene la situazione**
  - → **è un buon modello del mondo**
- **se invece *nega la realtà,***
  - → **non è un buon modello del mondo**
  - → **può produrre assurdità e inconsistenze**

Come si riconosce una *buona* classificazione?

EREDITARIETÀ: CONSEGUENZE

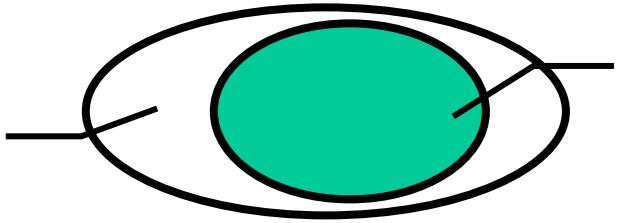
Una sottoclasse deve *delimitare un sottoinsieme* della classe base, altrimenti rischia di modellare la realtà *al contrario.*



Esempi

- **è vero che ogni Studente è *anche* una Persona**
  → Studente può derivare da Persona
- **non è vero che ogni Reale *sia anche* un Intero**
  → Reale non dovrebbe derivare da Intero

ESEMPIO: Persone e Studenti

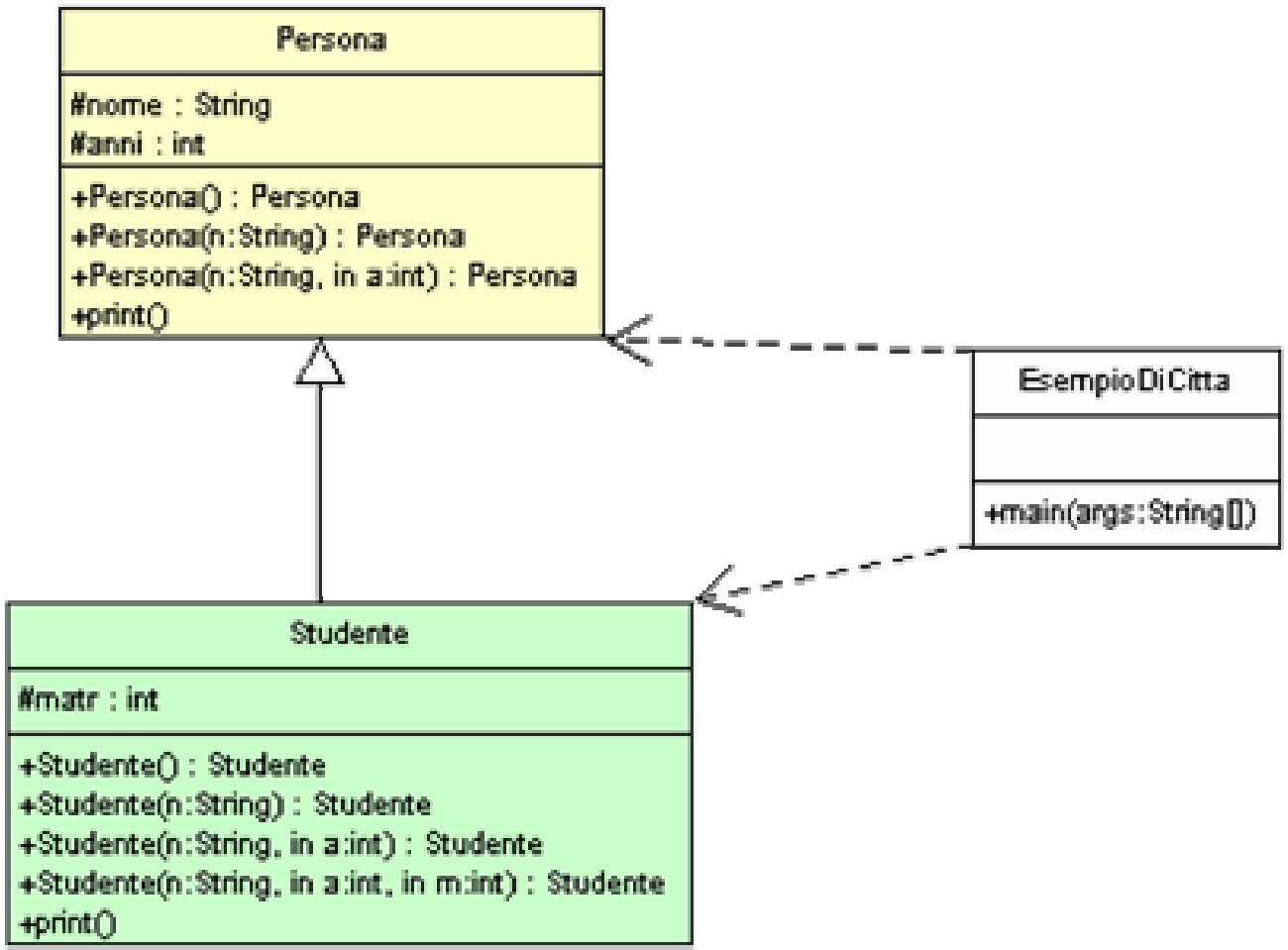


Una classe `Persona`

e una sottoclasse `Studente`

- ***è aderente alla realtà*, perché è vero nel mondo reale che *tutti gli studenti sono persone***
- **compatibilità di tipo: potremo usare uno studente (che è *anche* una persona) ovunque sia richiesta una generica persona**
  ma non viceversa: se serve uno studente, non ci si può accontentare di una generica persona.

Persone e Studenti: MODELLO



LA CLASSE Persona

```
public class Persona {
 protected String nome;
 protected int anni;
 public Persona() {
      nome = "sconosciuto"; anni = 0; }
 public Persona(String n) {
      nome = n; anni = 0; }
 public Persona(String n, int a) {
      nome=n; anni=a; }
 public void print() {
  System.out.print(”Mi chiamo " + nome);
  System.out.println(" e ho " +anni+ "anni");
 }
}
```

LA CLASSE Studente

```
public class Studente extends Persona {
 protected int matr;
 public Studente() {
      super(); matr = 9999; }
 public Studente(String n) {
      super(n); matr = 8888; }
 public Studente(String n, int a) {
      super(n,a); matr=7777; }
 public Studente(String n, int a, int m) {
      super(n,a); matr=m; }
 public void print() {
  super.print();
  System.out.println("Matricola = " + matr);
 }
}
```

Ridefinisce il metodo print di Persona

LA CLASSE `Studente`

```
public class Studente extends Persona {
 public S                (String n, int a, int m) {
      super(n,a); matr=m; }
 public void print() {
  super.print();
  System.out.println("Matricola = " + matr);
 }
}
```

Ridefinisce il metodo `void print()`

- **sovrascrive quello ereditato da `Persona`**
- **è una versione specializzata per `Studente` che però *riusa quello di `Persona`* (`super`), estendendolo per stampare la matricola.**

UN MAIN DI PROVA

```
public class EsempioDiCitta {
 public static void main(String args[]){
  Persona  p = new Persona("John",21);
  Studente s = new Studente("Tom",33);
  p.print();  // stampa nome ed età
  s.print();  // stampa nome, età, matricola
 }
}
```

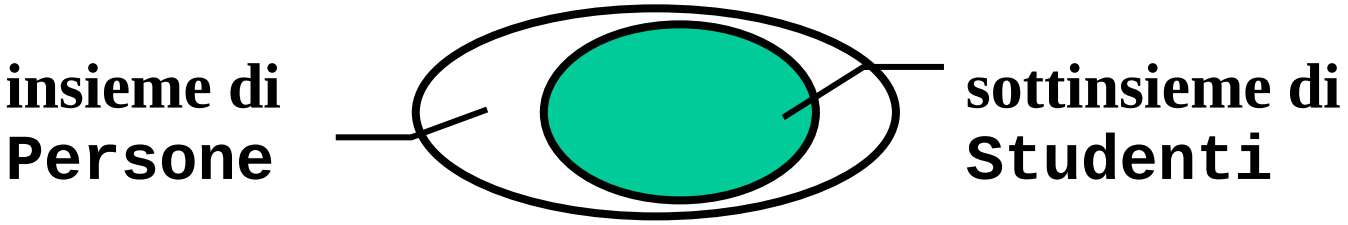
OUTPUT:

```
Mi chiamo John e ho 21 anni
Mi chiamo Tom e ho 33 anni
La mia matricola è 7777
```

print di Persona

print di Studente

UNA NUOVA QUESTIONE



Poiché `Studente` eredita da `Persona`, possiamo usare uno Studente ovunque sia richiesta una Persona.

Ciò equivale infatti a dire che il tipo `Studente` è <u>compatibile</u> col tipo `Persona`, esattamente come **`float`** è compatibile con **`double`.**

Ergo, se `s` è uno `Studente` e `p` una `Persona`, la frase:

```
p = s
```

è LECITA e *non comporta perdita di informazione.*

MA ALLORA…

Se è lecito scrivere:

```
Persona  p = new Persona("John",21);
Studente s = new Studente("Tom",33);
p = s;
```

COSA SUCCEDE ADESSO invocando metodi su `p` ??

```
p.print();  // COSA STAMPA ???
```

Infatti, p è un riferimento a `Persona`,
ma gli è stato assegnato un oggetto `Studente`!

COSA è "GIUSTO" che accada?
QUALE `print` deve scattare?

Ereditarietà: conseguenze

Polimorfismo, problema e opportunità

POLIMORFISMO

- **Un metodo si dice *polimorfo* quando è in grado di *adattare il suo comportamento allo specifico oggetto* su cui deve operare.**
- **In Java la possibilità di usare riferimenti a *una data classe***
  - ad esempio, **`Persona`**

  per puntare a oggetti *di classi più specifiche*
  - ad esempio, **`Studente`**

  apre le porte al polimorfismo
  - il metodo **`print`** può essere polimorfo

POLIMORFISMO

Per avere polimorfismo, occorre che

- **NON si consideri il tipo del riferimento**
  - in tal caso, **`p.print()`** stamperebbe solo nome ed età, perché si chiamerebbe **`print`** della classe **`Persona`**
- **MA si consideri invece il tipo *dell'oggetto effettivamente referenziato***
  - in questo caso, **`p.print()`** stamperebbe nome, età e matricola perché viene invocato il metodo di **`Studente`**

Questo è possibile se il linguaggio adotta la tecnica nota come *LATE BINDING (collegamento ritardato)* per l'invocazione dei metodi

- **default in Java (a richiesta in C++/C# keyword `virtual`)**

UN ESPERIMENTO IN JAVA

`print` è un metodo polimorfo:

poiché `p` in questo momento sta referenziando uno `Studente`, viene chiamato il metodo `print` di `Studente`

→ si stampano nome, età *e matricola*

```
  p.pri                stampa nome ed eta
  s.prin    ;   // stampa nome, età, matricola
  p=s;
  p.print();  // POLIMORFISMO!
 }
}
```

Se però a `p` venisse assegnato un (riferimento a) `Persona`, `p.print` chiamerebbe il metodo `print` di `Persona`.

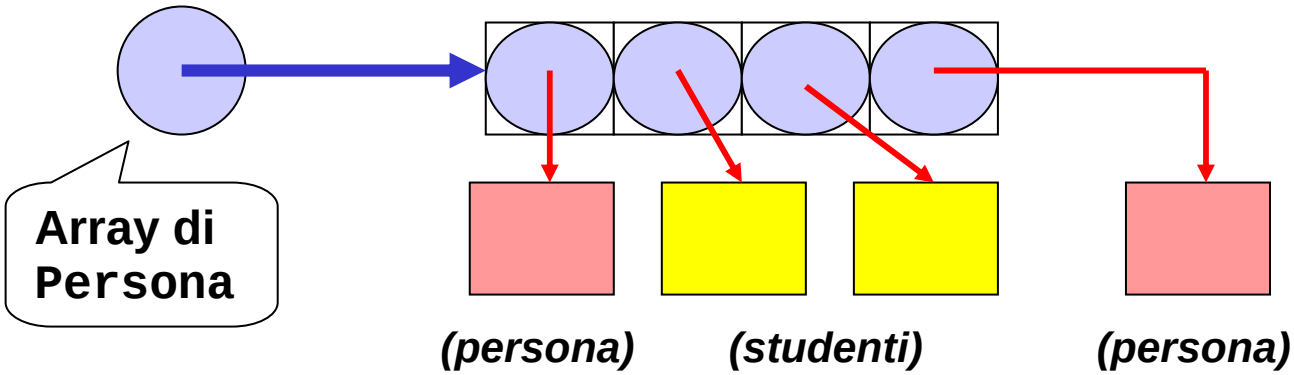
POLIMORFISMO e LATE BINDING

LATE BINDING:

le chiamate ai metodi sono *risolte solo al momento della chiamata,* in base all'*effettivo oggetto referenziato*

- **NON ci si fa "fuorviare" dal *tipo apparente* del riferimento (nel nostro caso, il tipo `Persona` con cui è dichiarato `p`)**
- **SI VA A VEDERE il *tipo effettivo* dell'oggetto su cui è fatta la chiamata (qui, si constata che `p` punta ora a uno `Studente`)**

Quindi, ad esempio, in un array di **`Persona`**, **`v[i].print`** *invocherà ogni volta il metodo appropriato,* cella per cella:

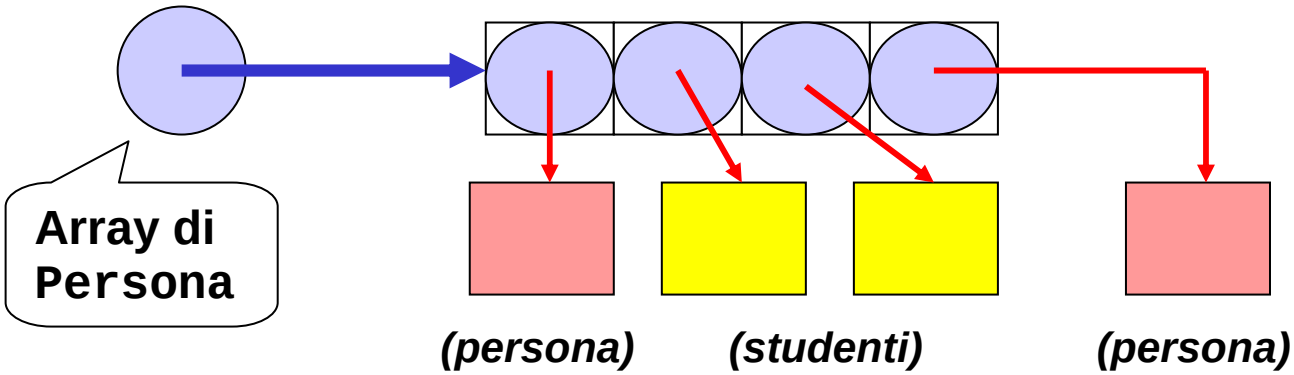


UN ESEMPIO PIÙ CONVINCENTE

```
Persona[] v = {new Persona("John"),new Studente("Jeff"),
               new Studente("Anna"), new Persona("Jane")
              };
for (int i=0; i<v.length; i++) v[i].print();
```

Output:

```
Mi chiamo John e ho 0 anni
Mi chiamo Jeff e ho 0 anni  Matricola = 8888
Mi chiamo Anna e ho 0 anni  Matricola = 8888
Mi chiamo Jane e ho 0 anni
```



LATE BINDING: COME, DOVE, PERCHÉ

COME:

- **il compilatore *non collega a priori* le chiamate con una *specifica funzione* (Binding Statico o *Early Binding*),**
- **si limita a *predisporre il necessario* perché *a run time* si possa chiamare al volo il metodo "giusto" per l'oggetto**

DOVE:

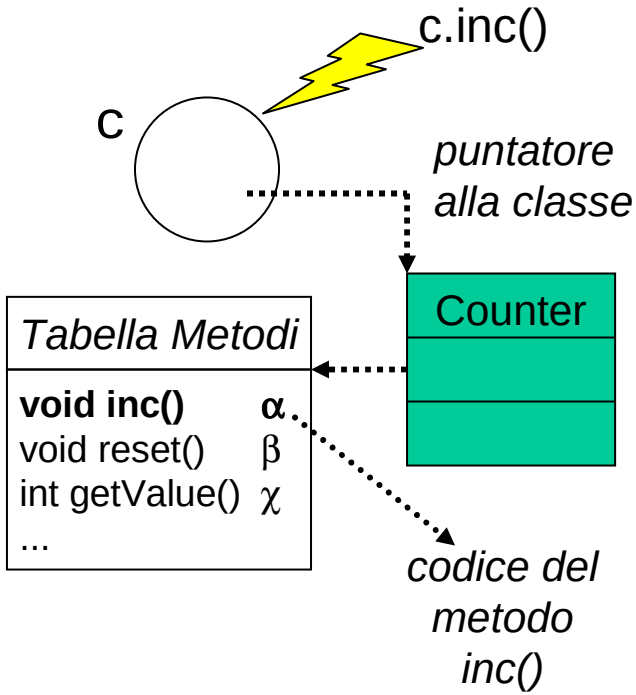
- **il Late Binding è la tecnica standard in Java per (quasi) *tutte le chiamate di metodi***
- **in altri linguaggi (come C++ o C#) va *attivato esplicita-mente* dichiarando i metodi *VIRTUALI***

PERCHÉ adottarlo come tecnica generalizzata?

- **perché è l'architrave del polimorfismo!**

LATE BINDING: FUNZIONAMENTO

- **Ogni istanza contiene un riferimento alla propria classe**
- **Ogni classe include una tabella *(Virtual Method Table)* che mette in corrispondenza i *nomi dei metodi* da essa definiti con il *codice* di ogni metodo**
- ***Chiamare un metodo* comporta quindi:**
  - **accedere alla tabella VMT della classe cui appartiene l'oggetto**
  - **accedere alla riga della tabella che corrisponde al metodo chiamato e ricavare il riferimento al suo codice**
  - **invocare il metodo identificato.**



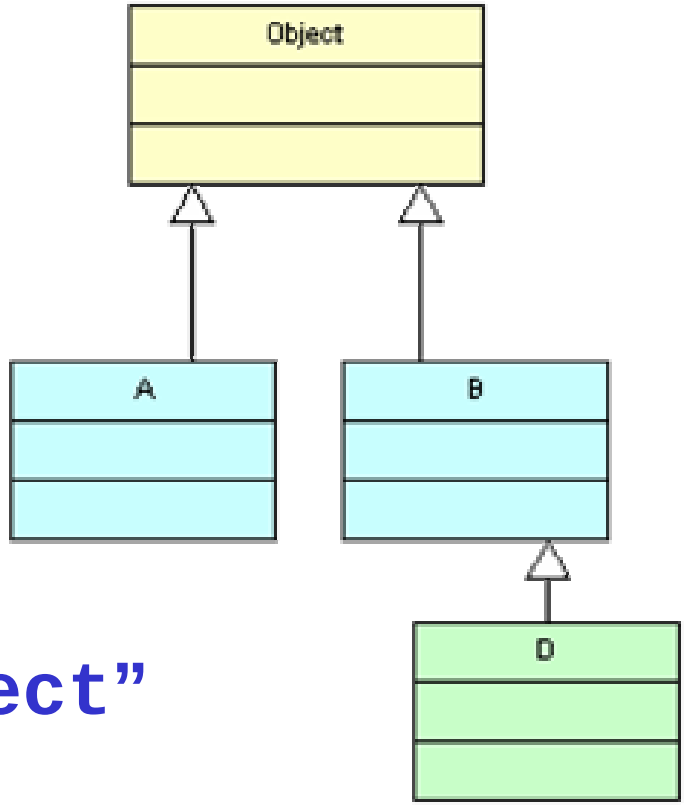
Gerarchie di ereditarietà

GERARCHIE DI EREDITARIETÀ

- **La relazione di ereditarietà determina la nascita di *gerarchie* o *tassonomie* di ereditarietà**

- **In Java, *ogni classe deriva implicitamente dalla classe base `Object`,* che è la *radice della gerarchia***

- **La frase “`class A`” sottintende “`extends Object`”**

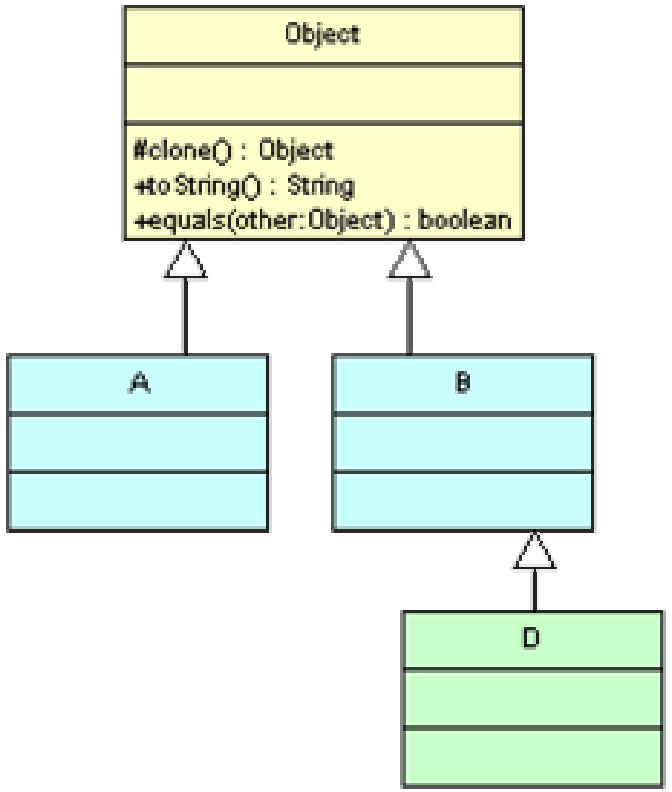


`Object`, LA RADICE DI TUTTO

I metodi definiti dalla classe base *`Object`* sono *ereditati da tutte le altre classi:*

Per questo tali funzionalità sono "predefinite" in tutti gli oggetti!

Le singole classi potranno *ridefinirli* per *personalizzarne il comportamento*.



I PRINCIPALI METODI di `Object`

I metodi definiti dalla classe base *`Object`* sono *ereditati da tutte le altre classi:*

```
public String toString()

public boolean equals(Object obj)

protected Object clone()
...
```

Quella predefinita stampa un identificativo dell'oggetto
Es: `Counter@712c1a3c`

Quella predefinita confronta i riferimenti (funziona come ==)

È la base per clonare oggetti

È protetta perché non tutti gli oggetti devono essere clonabili! Tocca al progettista stabilire se ciò abbia senso, caso per caso.

Un Counter.. da correggere

`Counter`: UN PROBLEMINO

Qualche lezione fa, definimmo così il metodo `equals` di `Counter`:

```
public boolean equals(Counter x){
   return (val==x.val);
}
```

Tale definizione apparve logica, dato che si volevano confrontare due oggetti `Counter`.

Confrontiamola però con *la definizione di `equals` data nella classe `Object`:*

```
public boolean equals(Object x)
```

`Counter`: UN PROBLEMINO (segue)

È evidente che *non è la stessa funzione!*

- **la `equals` di `Counter` *non sta ridefinendo quella di `Object`,* si sta *affiancando a essa***
  → è un caso di *overloading*
- **`Counter` non ha sostituito la `equals` generica (inadatta) con la sua: ne ha aggiunta un'altra!**
  - **ma così *non ha disattivato la prima,* che continua a esistere e potrebbe dare comportamenti "curiosi"**
  - **è pericoloso e quindi *inopportuno***

→ *opportuno correggere `Counter`*

`Counter`: REVISIONE

Da così:

```
public boolean equals(Counter x){
   return (val==x.val);
}
```

a così:

```
public boolean equals(Object x){
   return (val==(Counter)x.val);
}
```

Occorre un CAST, perché il generico `Object` *non ha un* campo `val`!

ma il CAST Java è sicuro e verificato a run time

Consideriamo questo esempio

```
public static boolean idem(Counter[] a, Counter[] 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;
}
```

Si basa sul concetto di "uguale"
TIPICO DELLA CLASSE `Counter`

- La `equals` di `Counter` nella precedente versione *avrebbe creato problemi in presenza di polimorfismo,* pur funzionando bene in questo caso.
- Infatti, ad esempio se l'array `b` fosse stato dichiarato come *array di Object* la vecchia `equals` non sarebbe stata trovata: sarebbe stata chiamata la `equals` ereditata da `Object`, che però confronta i riferimenti!

ORA INVECE…

```
public static boolean idem(Counter[] a, Counter[] 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;
}
```

Si basa sul concetto di “uguale”
TIPICO DELLA CLASSE `Counter`

- La nuova `equals` di `Counter` assicura invece *il perfetto funzionamento del polimorfismo,* in tutte le situazioni.
- Se anche l’array `b` fosse dichiarato come *array di Object* verrebbe trovata e usata la nuova `equals`, ottenendo il corretto comportamento.

E non è finita…

UNA FUNZIONE “quasi” GENERICA

- La funzione `idem` è definita per gli array di `Counter`, *ma la sua logica di funzionamento è indipendente dal tipo*
- *La scriveremmo identica* anche per confrontare array di `String`... o di `Persona`... !

```
public static boolean idem(Counter[] a, Counter[] b)
public static boolean idem(String[] a, String[] b)
public static boolean idem(Persona[] a, Persona[] b)
...
```

- Cambia solo la dichiarazione del *tipo dell’array*
- *MA ALLORA… viene da chiedersi se non si possa scriverne una VERSIONE GENERICA valida per tutti!*

UNA FUNZIONE GENERICA

- **Per ottenere un comportamento "generico" è possibile *usare come tipo formale `Object`* in quanto tipo *più "generico";* si parla allora di *"polimorfismo verticale"***

Funzione generica con parametri array di Object

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

- **Funziona… MA equivale praticamente ad abolire (quasi) il controllo di tipo!**
- ***Si potrebbero passare due array di tipi diversissimi e il compilatore non se ne potrebbe accorgere…***
- ***… ma il confronto ben difficilmente avrebbe un senso!***

UNA ALTERNATIVA MIGLIORE

- **JAVA 1.5 introduce il concetto di TIPO GENERICO per *scrivere* componenti *parametrici rispetto al tipo***
- **Il tipo diventa un parametro, da specificare poi al momento dell'uso:**

Funzione specifica per i Counter:

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

Funzione generica:

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

- **Questa funzione, scritta una volta sola, può essere usata per confrontare qualsiasi coppia di array *dello stesso tipo, qualunque sia il tipo: darà (giustamente) errore tentando di confrontare array di tipi diversi***

UN CASO DI STUDIO

Numeri Reali e Numeri Complessi

- **Si vogliono progettare due classi che catturino il concetto di *numero reale* (`Real`) e di *numero complesso* (`Complex`)**

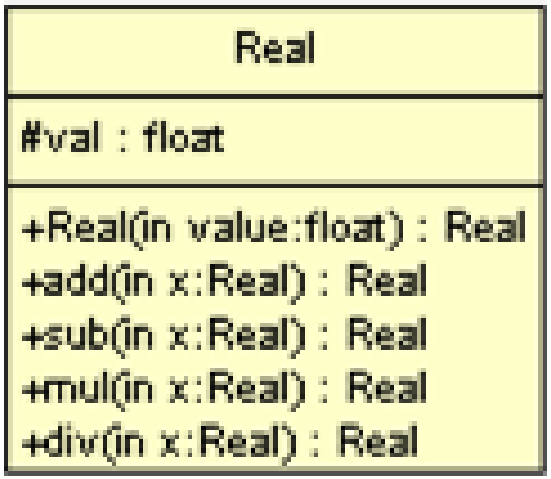
Principi-guida

- **aderenza alla realtà**
- **"efficienza" della rappresentazione**

Imposteremo un progetto per sottoporlo poi a *revisione critica.*

UN PRIMO APPROCCIO

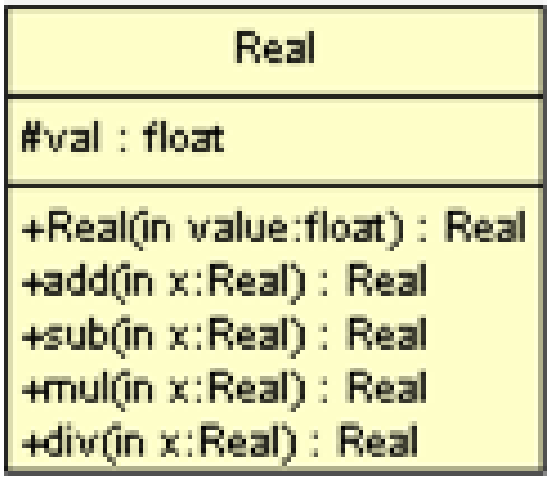
- **Un progettista *poco esperto* potrebbe iniziare subito a rappresentare il concetto di *numero reale*, senza riflettere a fondo.**
- **In un tale scenario, probabilmente verrebbe progettata una classe `Real` come questa:**
  - rappresentazione interna: un float
  - costruzione a partire da un valore float
  - metodi per effettuare le quattro operazioni (sum, sub, mul, div)
  - ed eventualmente altre...



PRIMO APPROCCIO: Real

Possibile realizzazione di questo `Real`:

```
class Real {
 protected float val;
 public Real(float value) { val = value; }
 public Real sum(Real x){
   return new Real(val + x.val); }
 public Real sub(Real x){
   return new Real(val - x.val); }
 public Real mul(Real x){
   return new Real(val * x.val); }
 public Real div(Real x){
   return new Real(val / x.val); }
}
```



PRIMO APPROCCIO: Complex

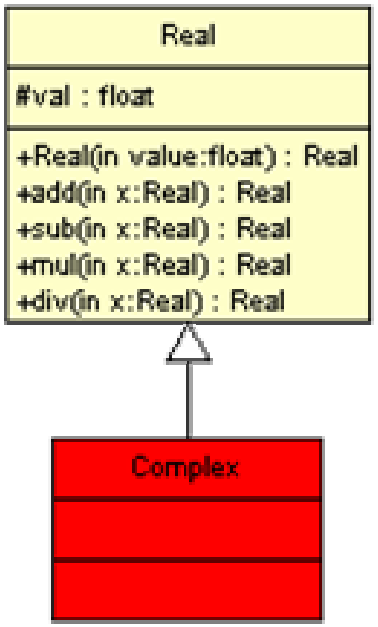
Ora si deve definire la classe `Complex`.

- un progettista *poco esperto*, considerando che un `Complex` è caratterizzato da parte reale e immaginaria, potrebbe decidere di *derivare `Complex` da `Real`.*

- ***MA la realtà è fatta a rovescio!***
  I complessi NON SONO un sottoinsieme dei reali!

- NON modella correttamente la realtà
- È errato, deve essere ripensato da capo!

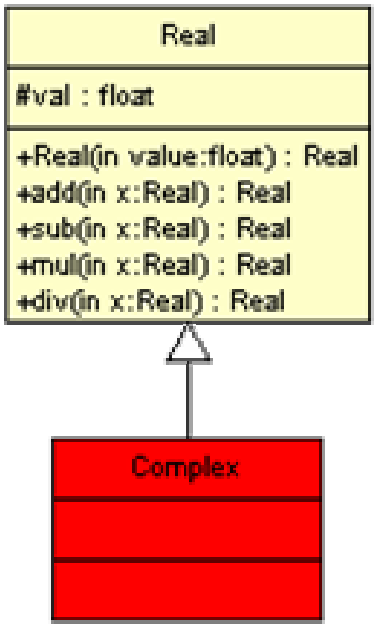
Modello falso della realtà !



PRIMO APPROCCIO: bilancio

Sarebbe un modello assurdo!

- **si potrebbe assegnare un `Complex` a un `Real`, *ma non viceversa***
- **le compatibilità di tipo funzionerebbero tutte *"a rovescio"* rispetto alla realtà matematica usuale.**



Dove sta l'errore di fondo?

- **La parte immaginaria *non è una proprietà extra dei numeri complessi: ce l'hanno anche i reali*, tanto è vero che *sappiamo perfino quanto vale!* (zero)**
- **Nei reali non si indica solo perché è sottintesa, *non perché non ci sia!* L'abitudine tende a fuorviare…!**

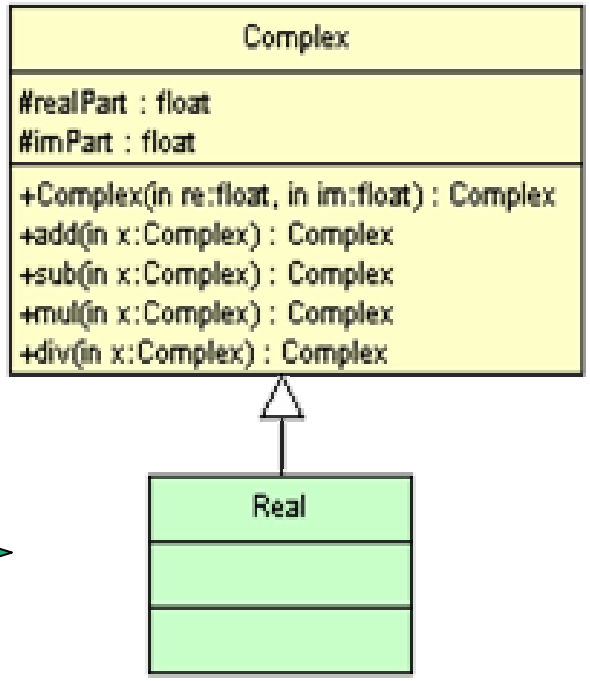
UN NUOVO APPROCCIO

- **Un progettista *esperto* riflette innanzitutto *sulla relazione esistente fra reali e complessi***

$$R \subset C$$

- **Perciò, egli decide *a priori* che il modello corretto è che `Complex` generalizzi `Real`**
  - le considerazioni di efficienza, se mai, vengono *dopo*

Modello corretto della realtà



NUOVO APPROCCIO: caratteristiche

La classe `Complex`

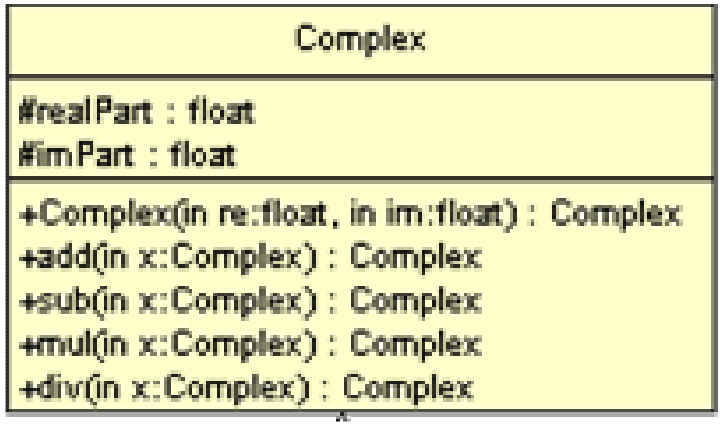
- **IPOTESI: rappresentazione interna basata su due float**
  - *parte reale, parte immaginaria*
- **REQUISITO: le operazioni** (sum, sub, mul, div) **devono poter operare *su qualunque numero complesso***

La classe derivata `Real`

- **ogni istanza ha comunque *due float*, anche se la parte immaginaria è sempre zero *(inefficienza)***
- **le operazioni continuano a operare sui complessi, sebbene possano essere semplificate.**

LA CLASSE `Complex`

Progetto della classe `Complex`



OPERAZIONI

- **somma fra complessi**
- **sottrazione fra complessi**
- **moltiplicazione fra complessi**
- **divisione fra complessi**

… *e inoltre*

- **divisione di un complesso per un fattore reale**
- **coniugato di un numero complesso**
- **modulo (al quadrato) di un complesso**

RELAZIONI UTILI

- **(a+ib) ± (c+id) = (a+c) ± i (b+d)**
- **(a+ib) / x = (a/x + i b/x) *x* ∈ *R***
- **z / w = (z × cgt(w)) / $|w|^2$**

RELAZIONI UTILI

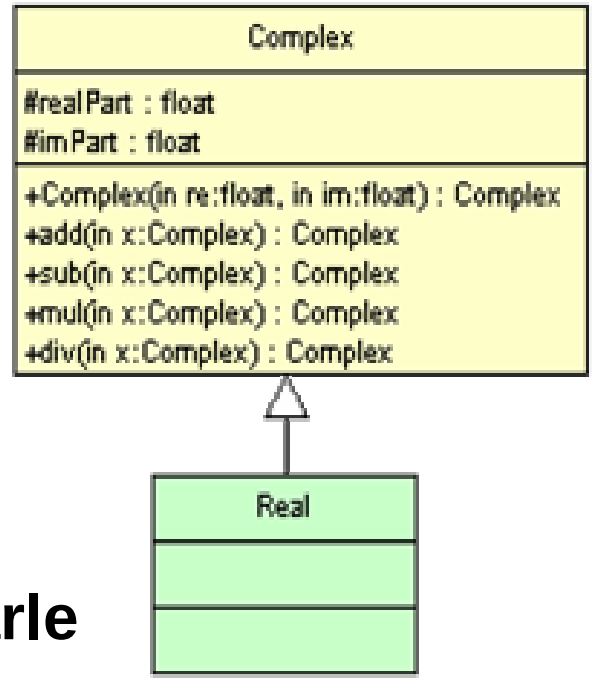
- **(a+ib) × (c+id) = (ac-bd) + i (bc + ad)**
- **cgt(a+ib) = (a-ib)**

Complex: IMPLEMENTAZIONE

```
public class Complex {
 protected float re, im;
 public Complex(float r, float i) { re = r;  im = i; }
 public Complex sum(Complex z){
   return new Complex(re+z.re, im+z.im); }
 public Complex sub(Complex z){
   return new Complex(re-z.re, im-z.im); }
 public Complex mul(Complex z){
   return new Complex(re*z.re-im*z.im, im*z.re+re*z.im); }
 public Complex div(Complex z){
   return mul(cgt(z)).divByFactor(z.squaredModule()); }
 public Complex cgt(Complex z){ return new Complex(re,-im); }
 public Complex divByFactor(float x) {
   return new Complex(re/x, im/x); }
 public float squaredModule(){ return re*re + im*im; }
 public String toString(){ return "" + re + "+i" + im;}
}
```

LA CLASSE `Real`

- **Ogni numero reale ha comunque una parte immaginaria, che però vale zero**
- **Di conseguenza, anche se le operazioni precedenti restano valide, risultano *inutilmente inefficienti***
- **Ergo, può essere utile re-implementarle per guadagnare efficienza**
  - **somma, sottrazione, moltiplicazione e divisione fra reali sono *molto più semplici* delle stesse operazioni fra complessi**
  - **il loro risultato, inoltre, è sempre un reale**



`Real`: IMPLEMENTAZIONE

```
public class Real extends Complex {
 public Real(float x) { super(x,0); }
 public Real sum(Real x){ return new Real(re + x.re); }
 public Real sub(Real x){ return new Real(re - x.re); }
 public Real mul(Real x){ return new Real(re * x.re); }
 public Real div(Real x){ return new Real(re / x.re); }
 public String toString(){ return "" + re; }
}
```

NOTARE:

- **il costruttore di Real si appoggia su quello di Complex, *impostando però sempre la parte immaginaria a zero***
- **le operazioni sono reimplementate *nel caso specifico in cui gli operandi siano entrambi Real*, poiché allora anche il risultato lo è**
- **rimangono comunque disponibili *tutte* le operazioni già esistenti**
- **viene reimplementata `toString()` in modo specifico per i Real**

ESEMPIO D'USO

```
public class Prova {
 public static void main(String args[]){
  Real r1 = new Real(18.5F), r2 = new Real(3.14F);
  Complex c1 = new Complex(-16, 0), c2 = new Complex(3, 2),
           c3 = new Complex(0, -2);
  Real r = r1.sum(r2); Complex c = c1.sum(c2);
  System.out.println("r1 + r2 = " + r); // il reale 21.64
  System.out.println("c1 + c2 = " + c); // il complex -13+2i
  System.out.println("c1 + c2 -i = "+c.sub(new Complex(0,1)));
                                        // il complesso -13+i
  c = c.sum(c3);
  System.out.println("c + c3 = " + c);  // -13+0i
  c = r;
  System.out.println("c = r; c = " + c);
  // POLIMORFISMO: c è reale → toString
  // dei reali → stampa 21.64
 }
}
```

```
r1 + r2 = 21.64
c1 + c2 = -13.0+i2.0
c1 + c2 -i = -
13.0+i1.0
c + c3 = -13.0+i0.0
c = r; c = 21.64
```