Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Ereditarietà: conseguenze. Relazione tipo / sottotipo

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

EREDITARIETÀ: UNA RIFLESSIONE

- Poiché una classe derivata estende la classe-base, senza togliere niente, *la classe derivata offre senz'altro tutte le funzionalità della classe base*
  - eventualmente, può offrirne di più, tramite nuovi metodi.
- Da qui segue il PRINCIPIO DI SOSTITUIBILITÀ:
  al posto di un oggetto della classe-base,
  si può sempre usare un oggetto di una classe derivata
  - ma certo! un'istanza della classe derivata, più ricca della classe base, *non può non saper svolgere* quelle operazioni!
  - ..anche se potrebbe svolgerle in modo "più specifico"..
  - in generale: *non ti puoi lamentare se, allo stesso prezzo, ti diamo un oggetto più ricco di quello che hai chiesto* ☺

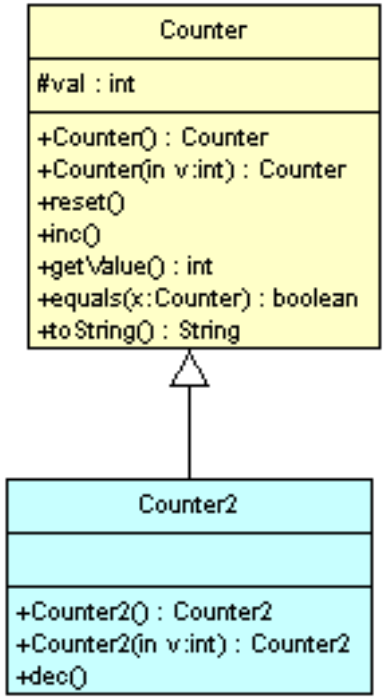
ESEMPIO

ESEMPIO

- a un cliente che si aspetti un **`Counter`** si può dare un **`Counter2`**, perché tutte le funzionalità di un **`Counter`** sono certamente presenti in un **`Counter2`** *in quanto esso estende* ***`Counter`***

- eventualmente, **`Counter2`** potrà svolgere alcune funzionalità in modo "adeguato" alla sua natura di "counter più evoluto"

- non vale il viceversa: a un cliente che si aspetti un **`Counter2`** non si può «rifilare» un banale **`Counter`**, perché non tutte le funzionalità di un **`Counter2`** sono presenti in un **`Counter!`**



EREDITARIETÀ & SUBTYPING

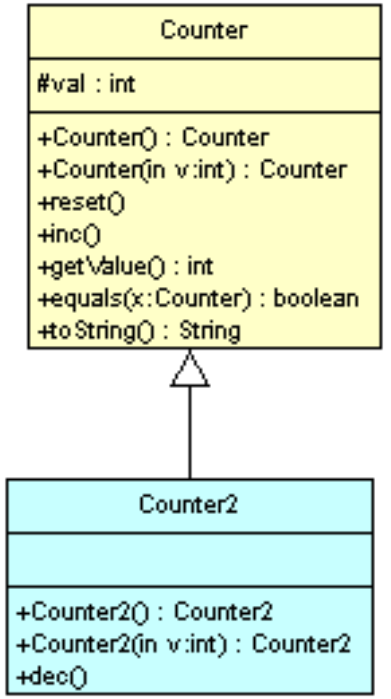
- Conclusione: ogni **`Counter2`** è anche un (particolare) **`Counter`** – ma non viceversa

Ereditarietà IS-A

- Ciò si riverbera sulla relazione fra i due tipi: il tipo **`Counter2`** è *un sottotipo* di **`Counter`**

Subclassing → Subtyping

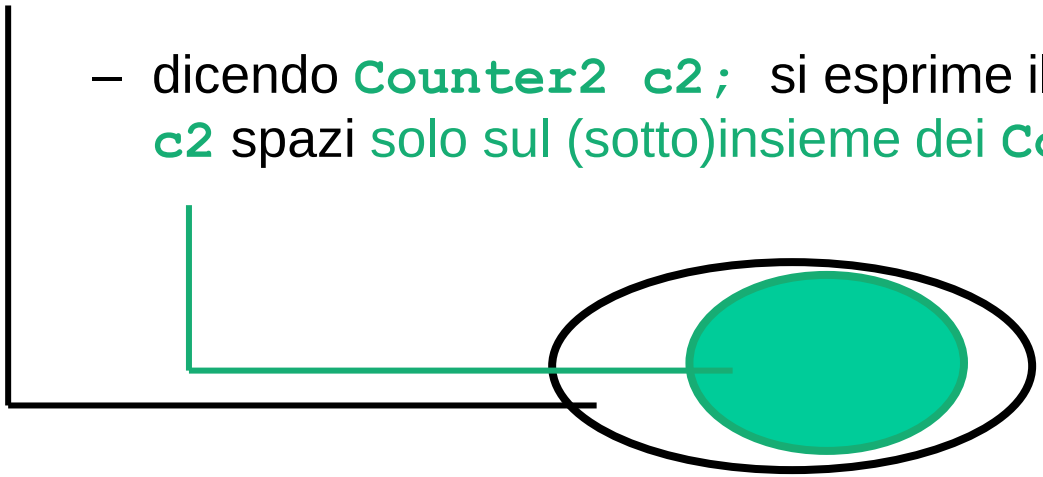
- Questa *compatibilità di tipo* si manifesta in
  - assegnamenti
  - passaggi di parametri da/a funzioni



CONSEGUENZE (1/3)

- Cosa significa dire che "se un cliente si aspetta un **`Counter`** si può dargli un **`Counter2`**" ?

- Un cliente specifica cosa si aspetta quando definisce il ***tipo*** di un riferimento
  - dicendo **`Counter c;`** si esprime il requisito che **`c`** spazi sull'insieme di tutti i **`Counter`**
  - dicendo **`Counter2 c2;`** si esprime il requisito che **`c2`** spazi solo sul (sotto)insieme dei **`Counter2`**



CONSEGUENZE (2/3)

- Quindi:
  - 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`**,
    deve poter in realtà restituire un'istanza di **`Counter2`**
- ma non viceversa:
  - **`Counter`** non è un particolare **`Counter2`**, dunque *non si può "rifilare" un semplice* ***`Counter`*** *a un cliente che esigeva un vero* ***`Counter2`***

CONSEGUENZE (3/3)

In pratica:

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

Java C#

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

Java C#

```
static Counter of(int v){ // fa
  return new Counter2(v);
}
```

Sarebbe sbagliato anche se l'istanza fosse un `Counter2` *perché formalmente si sa solo che* `c` è un `Counter`

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

ESEMPIO

Java
C#

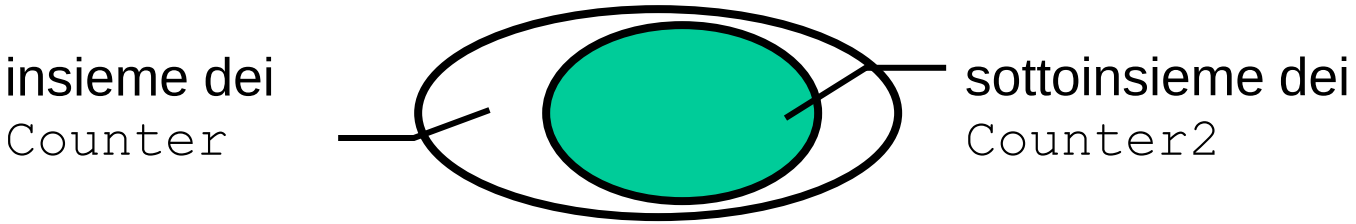
```
public class Esempio6 {
 public static void main(String args[]) {
   Counter  c1 = new Counter(10);
   Counter2 c2 = new Counter2(20);
   c1=c2;              // OK: c2 è anche un Counter
     // c2=c1;         // NO: c1 è solo un Counter
 }
}
```

OK perché **`c2`** è un **`Counter2`**
e *quindi anche implicitamente un* ***`Counter`*** (e `c1` è di tipo **`Counter`**)

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

CONSEGUENZE CONCETTUALI

- Dire che i **`Counter2`** sono un sottoinsieme dell'insieme dei **`Counter`** induce *una classificazione del mondo*
  - se è aderente alla realtà → buon modello, grandi vantaggi
  - se è difforme dalla realtà → disastro, guai a non finire!



- L'ereditarietà IS-A è molto più di un semplice "riuso di codice": riusa *l'astrazione* e con essa *le relazioni fra le cose*

Come si riconosce
una buona classificazione?

CLASSIFICAZIONE DELLA REALTÀ

- Una buona classificazione è quella in cui una sottoclasse *delimita un sottoinsieme* della classe base
- SEMBRA facile da valutare.. ma non sempre è così.
- In effetti, questa caratteristica:
  - è semplice da valutare se gli oggetti sono *valori* che *non subiscono trasformazioni* dopo la costruzione
  - è problematica se, invece, gli oggetti *subiscono trasformazioni* dopo la costruzione perché c'è il rischio che la classe derivata *aggiunga vincoli comportamentali* a quelli della classe base: in tal caso… ..*niente è più come sembra!*

CLASSIFICAZIONE DELLA REALTÀ

ESEMPI

- Ogni **`Studente`** è *anche* una **`Persona`**
  - l'insieme degli Studenti è sottoinsieme dell'insieme delle Persone
  - ogni Studente ha tutte le proprietà di una Persona, più altre
- Ogni **`Quadrato`** è *anche* un **`Rettangolo`**
  - l'insieme dei Quadrati è sottoinsieme dell'insieme dei Rettangoli
  - ogni Quadrato ha tutte le proprietà di un Rettangolo, più altre

CLASSIFICAZIONI DELLA REALTÀ

ESEMPI

- Ogni **`Studente`** è *anche* una **`Persona`**
  - l'insieme degli Studenti è sottoinsieme dell'insieme delle Persone
  - ogni Studente ha tutte le proprietà di una Persona, più altre

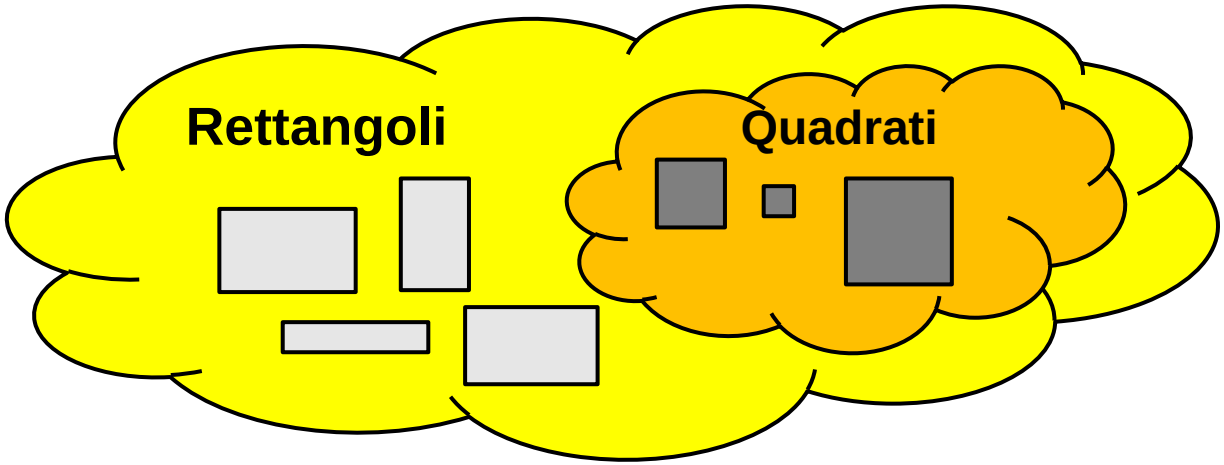
- Ogni **`Quadrato`** è *anche* un **`Rettangolo`**
  - l'insieme dei Quadrati è sottoinsieme dell'insieme dei Rettangoli
  - ogni Quadrato ha tutte le proprietà di un Rettangolo, più altre

… SICURI ?

In realtà, questa relazione insiemistica *è vera solo se si tratta di forme geometriche immodificabili,* che non cambiano dopo la costruzione.

QUADRATI & RETTANGOLI

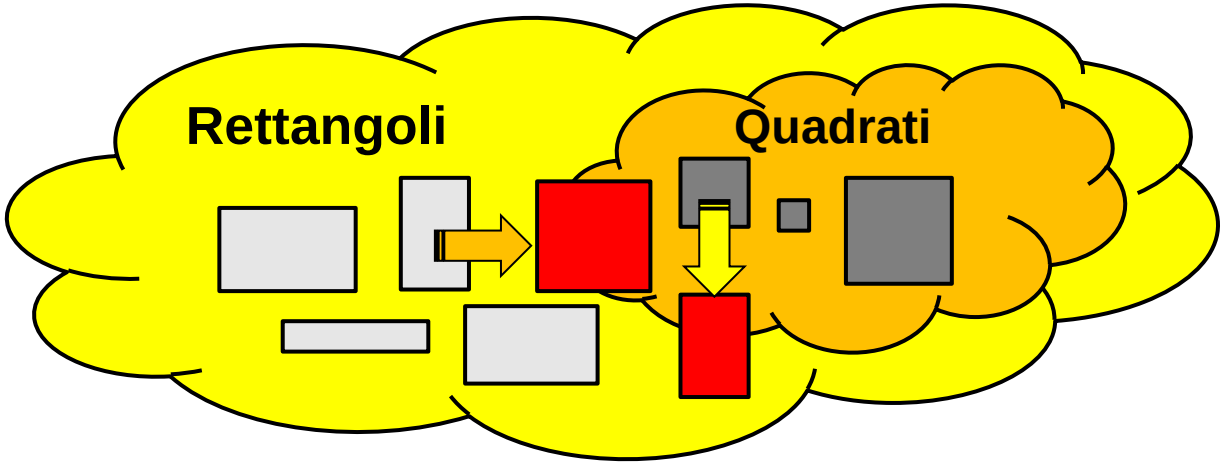
Se sono forme immodificabili, tutto è chiaro e semplice:



- Chi nasce rettangolo, muore rettangolo; chi nasce quadrato, muore quadrato.
- La categoria di ogni oggetto è *certa.*

QUADRATI & RETTANGOLI

Se però i lati dei rettangolo sono modificabili, *allora un quadrato non è più un rettangolo,* perché *in un quadrato i lati non possono cambiare singolarmente!*

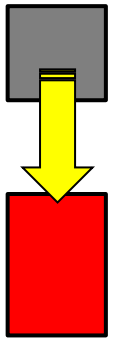


- Non è più vero che chi nasce rettangolo, muore rettangolo e chi nasce quadrato, muore quadrato.
- La categoria di ogni oggetto non è più *certa.*

QUADRATI & RETTANGOLI

Cambiando un lato, un Quadrato si declassa in Rettangolo!

Per garantire che resti quadrato, occorrerebbe aggiungere il *vincolo comportamentale* "lati sempre uguali", che però *non esiste in Rettangolo* → VIOLAZIONE CONTRATTO



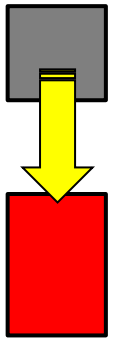
Pensi che questa sia solo astrusa teoria?

- **Non lo è: se non l'affronti, *non sai che codice scrivere!***
- Se metti due metodi **`changeWidth`** / **`changeHeight`** in **`Rettangolo`**, poi *come gestisci la cosa in* ***`Quadrato`****?*
  - se li lasci come sono, ottieni dei quadrati che possono diventare "non più quadrati", cambiando un lato solo dei due
  - se li re-implementi stabilendo che modifica larghezza si ripercuota su altezza (e viceversa), aggiungi un effetto imprevisto…
    ..col bel risultato che chi usa quel Quadrato come Rettangolo, si troverà modificato l'altro lato inaspettatamente → grafica fuori fase

QUADRATI & RETTANGOLI

Cambiando un lato, un Quadrato si declassa in Rettangolo!

Per garantire che resti quadrato, occorrerebbe aggiungere il *vincolo comportamentale* "lati sempre uguali", che però *non esiste in Rettangolo* → VIOLAZIONE CONTRATTO



Pensi che questa sia solo astrusa teoria?

- **Non lo è: se non l'affronti, *non sai che codice scrivere!***
- Se metti due metodi `changeWidth` / `changeHeig` `Rettangolo`, poi *come gestisci la cosa in O*
  - se li lasci come sono, ottieni dei quad "non più quadrati", cambiando
  - se li re-implementi stabilend ezza si ripercuota su altezza (e viceversa), agg imprevisto…
    ..col bel risultato che chi usa q adrato come Rettangolo, si troverà modificato l'altro lato inaspettatamente → grafica fuori fase

COMPLIMENTI VIVISSIMI !

QUADRATI & RETTANGOLI ESEMPIO

```java
package fondt2.ed;

public class Rettangolo {
    private double base, altezza;

    protected void setBase(double base) {
        this.base=base;}
    protected void setAltezza(double altezza) {
        this.altezza=altezza;}
    public double getBase() {
        return base;}
    public double getAltezza() {
        return altezza;}
    public double getDiagonale() {
        return Math.sqrt(base*base+altezza*altezza);}

    public Rettangolo(double base, double altezza){
        setBase(base); setAltezza(altezza);
    }
    protected Rettangolo(){
        // uso riservato a sottoclassi, per quando
        // base e altezza sono inizialmente ignote
    }

    public String toString(){
        return "Rettangolo di base " + getBase() + ", altezza " +
                getAltezza() + " e diagonale " + getDiagonale();
    }

    public void show(){
        System.out.println(this);
    }
}
```

Java

```java
package fondt2.ed;

public class Quadrato extends Rettangolo {

    public double getLato() {return getAltezza();}

    public Quadrato(double lato){
        super(lato, lato);
    }

    @Override
    public String toString(){
        return "Quadrato di lato " + getLato()
            + " e diagonale " + getDiagonale();
    }

    @Override
    protected void setBase(double base) {
        super.setBase(base); super.setAltezza(base);
    }

    @Override
    protected void setAltezza(double altezza) {
        super.setBase(altezza); super.setAltezza(altezza);
    }
}
```

Rimedio peggiore del male..

QUADRATI & RETTANGOLI ESEMPIO

Java

```
public static void main(String[] args) {
    Rettangolo r = new Rettangolo(3,4); r.show();
    Quadrato q = new Quadrato(3); q.show();
    r.setAltezza(6); r.show();
```

```
Rettangolo di base 3.0, altezza 4.0 e diagonale 5.0
Quadrato di lato 3.0 e diagonale 4.242640687119285
Rettangolo di base 3.0, altezza 6.0 e diagonale 6.708203932499369
```

Cambiare solo l'altezza di un rettangolo è logico..

```
public static void main(String[] args) {
    Rettangolo r = new Rettangolo(3,4); r.show();
    Quadrato q = new Quadrato(3); q.show();
    r.setAltezza(6); r.show();
    q.setAltezza(6); q.show();
    System.out.println("base del quadrato: " + q.getBase());
    System.out.println("altezza del quadrato: " + q.getAltezza());
}
```

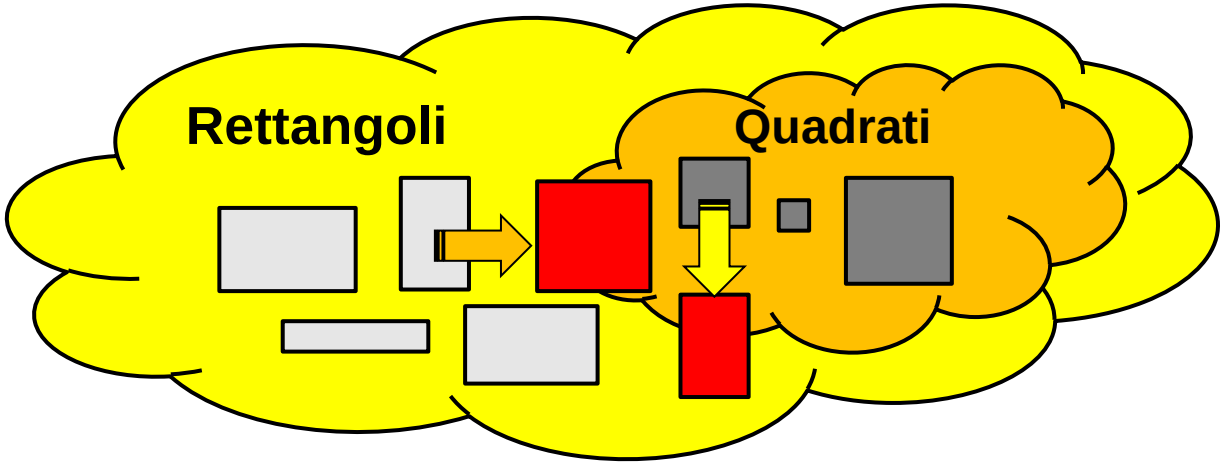
.. ma cambiare quella di un quadrato??

```
Rettangolo di base 3.0, altezza 4.0 e diagonale 5.0
Quadrato di lato 3.0 e diagonale 4.242640687119285
Rettangolo di base 3.0, altezza 6.0 e diagonale 6.708203932499369
Quadrato di lato 6.0 e diagonale 8.48528137423857
base del quadrato: 6.0
altezza del quadrato: 6.0
```

Rimedio peggiore del male: è cambiata surrettiziamente anche la base!!

QUADRATI & RETTANGOLI

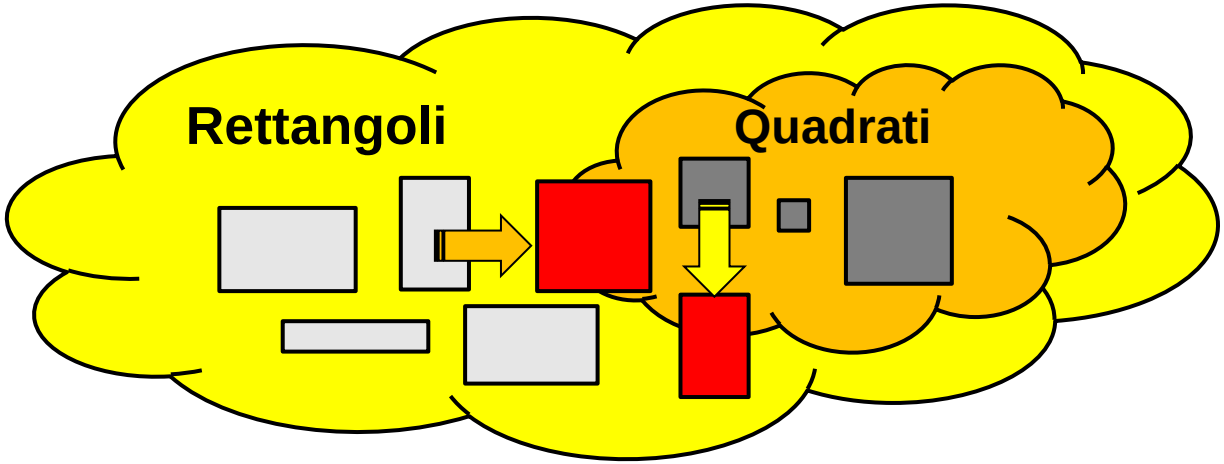
Se però i lati dei rettangolo sono modificabili, *allora un quadrato non è più un rettangolo,* perché *in un quadrato i lati non possono cambiare singolarmente!*



- Non è più vero che chi nasce rettangolo, muore rettangolo e chi nasce quadrato, muore quadrato.
- La categoria di ogni oggetto non è più *certa.*

QUADRATI & RETTANGOLI
BILANCIO

Conclusione: se i lati dei rettangoli sono modificabili, *allora un quadrato non è più un (caso particolare di) rettangolo,* perché *in un quadrato i lati non possono cambiare singolarmente*



- La categoria di ogni oggetto non è più *certa*
- Possibile alternativa: esprimere quadrato e rettangolo come *entità indipendenti*, derivate da una super-classe **`Forme`**

UN ESEMPIO DIVERSO
PERSONE & STUDENTI

Questo scenario invece non è problematico

- Ogni **`Studente`** è *anche* una **`Persona`**
  - l'insieme degli Studenti è sottoinsieme dell'insieme delle Persone
  - ogni Studente ha tutte le proprietà di una Persona, più altre

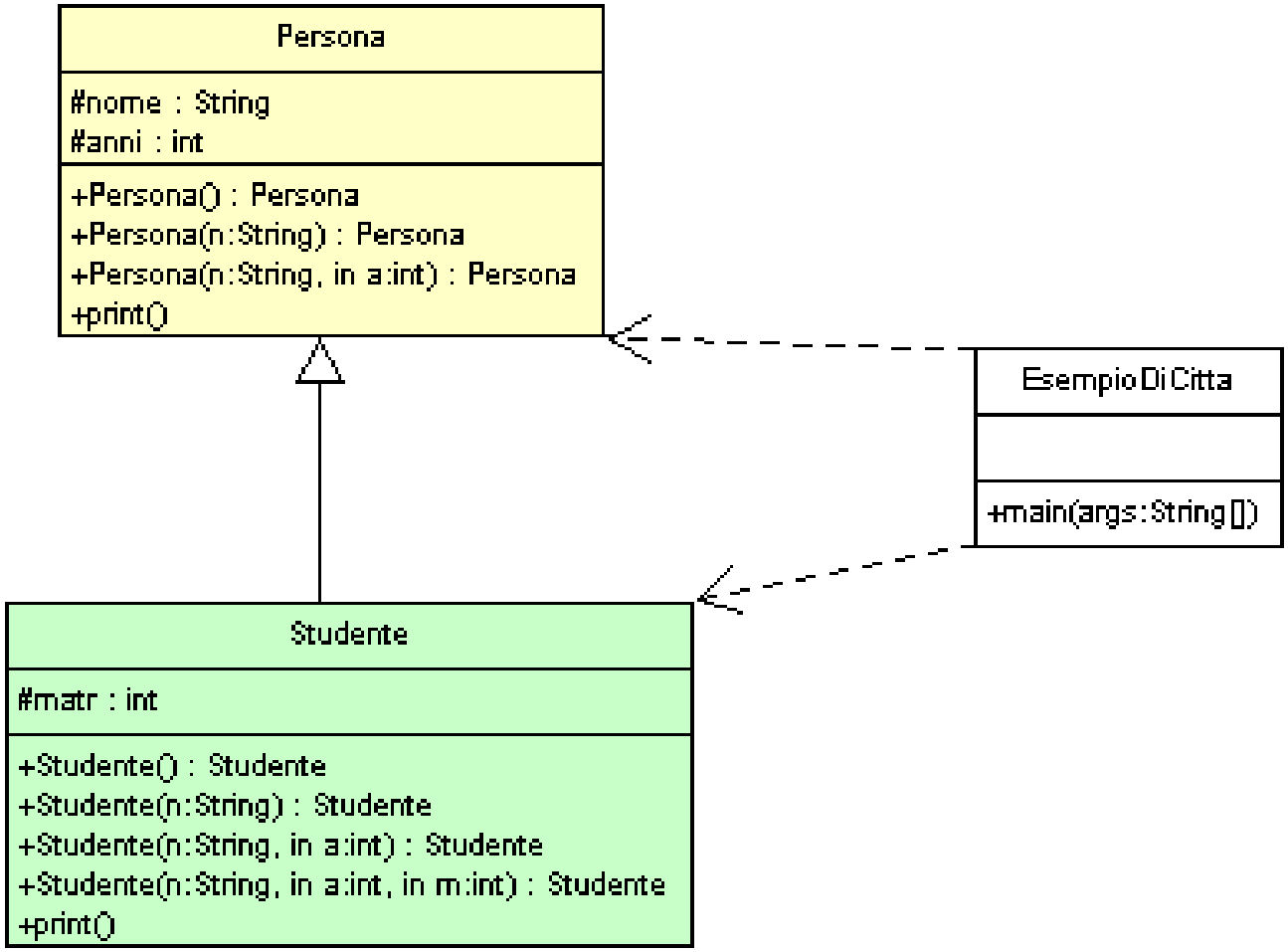
IPOTESI:

- non esistono studenti che non siano persone
- gli studenti *non aggiungono vincoli di comportamento* rispetto alle persone – hanno solo dati in più (matricola, lista esami, voti, etc.)

CONSEGUENZA: SUBTYPING

- il tipo **`Studente`** è sottotipo del tipo **`Persona`**
- compatibilità di tipo: si può usare uno Studente (che è *anche* una Persona) ovunque sia richiesta una Persona – ma non viceversa

Persone e Studenti: MODELLO



LA CLASSE Persona

Java

~C#

```
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");
 }
}
```

..sicuro?

ATTENZIONE !
Hanno davvero senso tutti questi costruttori?

Scala, Kotlin: minime modifiche

LA CLASSE Persona

Scala

```
class Persona(
   val nome:String, var anni:Int=0){
   def print() : Unit = {
     println("Mi chiamo " + nome +
              " e ho " +anni+ " anni");
   }
}
```

Solo costruttori /1 e /2

Kotlin

```
open public class Persona(
   val nome:String, var anni:Int=0){
   public open fun print() : Unit {
     println("Mi chiamo " + nome +
              " e ho " +anni+ " anni");
   }
}
```

Solo costruttori /1 e /2

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=
 public Studente(String
       super(n,a); matr=
 public void print(){
   super.print();
   System.out.println("M
 }
}
```

Java

~C#

C#, Kotlin: `:` anziché **`extends`**

C#: **`base`** al posto di **`super`**

Di nuovo, ATTENZIONE!

Lo fa riusando al proprio interno il metodo **`print`** ereditato da **`Persona`**

- estensione incrementale
- evita duplicazioni di codice
- gestione trasparente dei dati privati (non accessibili direttamente)

Questa **`print`** ridefinisce (sovrascrive) il metodo **`print`** ereditato da **`Persona`** per adattarlo alla classe **`Studente`**

LA CLASSE Studente

Scala

```
Studente(nome:String, anni:Int=0,
    val matr:Int=7777) extends Persona(nome, anni) {
   override def print() : Unit = {
       super.print();
       println("matricola = " + matr);
   }
}
```

Kotlin

```
public class Studente(nome:String, anni:Int=0,
    val matr:Int=7777) : Persona(nome, anni) {
   public override fun print() : Unit {
       super.print();
       println("matricola = " + matr);
   }
}
```

UN MAIN DI PROVA

```
public class EsempioDiCitta {
 public static void main(String args[]){
   Persona  p = new Persona("John",21);
   Studente s = new Studente("Tom",33,874131);
   p.print();  // stampa nome ed età
   s.print();  // stampa nome, età, matricola
 }
}
```

Java

~C#

```
Mi chiamo John e ho 21 anni
Mi chiamo Tom e ho 33 anni
Matricola = 874131
```

`print` di `Persona`

`print` di `Studente`

UN MAIN DI PROVA

Scala

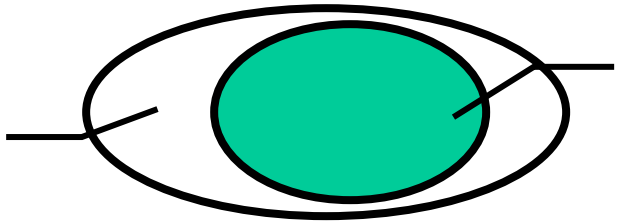
```
object EsempioDiCitta {
 def main(args: Array[String]) : Unit = {
   var p = new Persona("John");
   var s = new Studente("Tom");
   p.print();  // stampa nome ed età
   s.print();  // stampa nome, età, matricola
 }
}
```

Kotlin

```
fun main(args: Array<String>) : Unit {
   var p = Persona("John");
   var s = Studente("Tom");
   p.print();  // stampa nome ed età
   s.print();  // stampa nome, età, matricola
}
```

SITUAZIONE

insieme di **`Persona`**

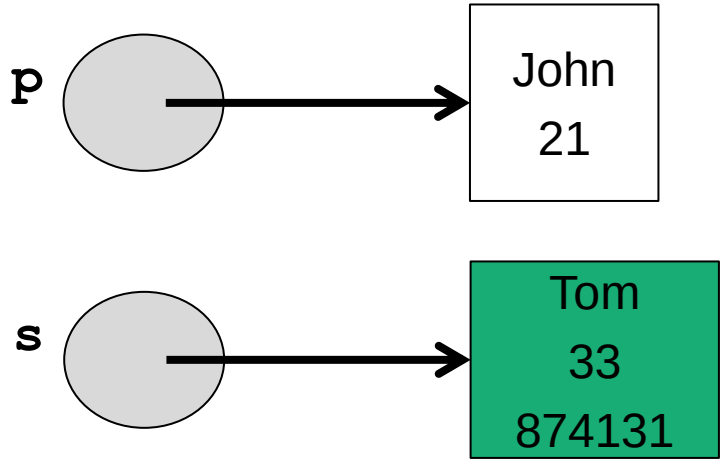


sottoinsieme di **`Studente`**

```
Persona  p = new Persona("John",21);
Studente s = new Studente("Tom",33, 874131);
```

Java

~C#



UNA NUOVA QUESTIONE

- Poiché **`Studente`** eredita da **`Persona`**, è un suo sottotipo: quindi, possiamo usare uno **`Studente`** ovunque sia richiesta una **`Persona`**
  - ciò equivale a dire che il tipo **`Studente`** è *compatibile* col tipo **`Persona`**, come **`float`** con **`double`**, o **`int`** con **`long`**
  - MA NON VICEVERSA: non tutte le persone sono studenti, e d'altronde neanche tutti i **`long`** sono **`int`**
- Ma allora.. una frase come questa:

```
p = s
```

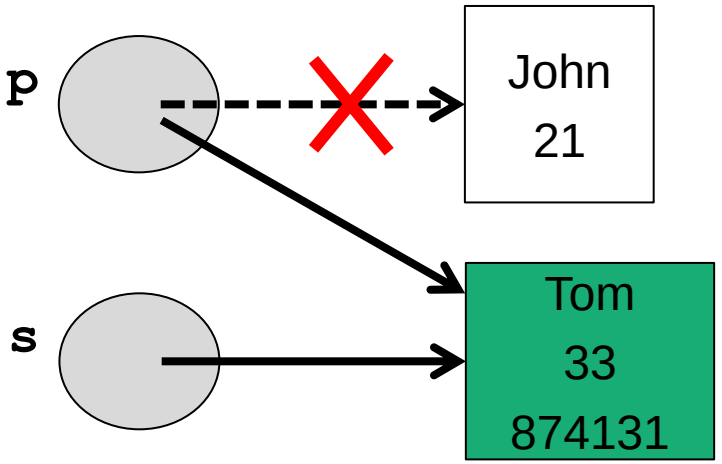
- Si può scrivere? ha senso?
- MA SOPRATTUTTO… se sì, *cosa comporta ?*

UNA NUOVA QUESTIONE

- Poiché:
  - il riferimento **`p`** è un riferimento a **`Persona`**
  - il riferimento **`s`** è un riferimento a **`Studente`**
  - il tipo **`Studente`** è compatibile col tipo **`Persona`**

  la frase **`p = s`** deve essere *lecita*
- Trattandosi di riferimenti, l'operatore **`=`** significa *alias*
  - quindi, la situazione è questa:
  - i due riferimenti **`p`** e **`s`** puntano allo stesso oggetto

Cosa comporta ciò quando si invocano metodi?



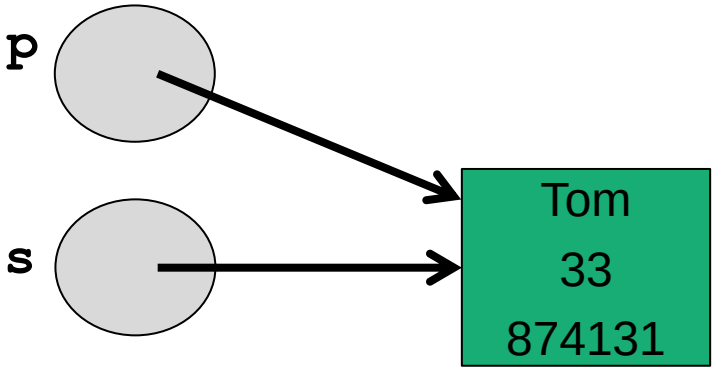
PROBLEMINO

- Se invochiamo il metodo **`print`** su **`p`**, *cosa stampa?*

```
p.print();
```

- È una situazione del tutto nuova!
  - Finora, il tipo del riferimento aveva sempre coinciso col tipo dell'oggetto puntato
  - Qui, invece, **`p`** è formalmente un riferimento a **`Persona`**, ma gli è stata assegnata una **`Persona`** molto speciale*: uno* ***`Studente`*** *!*

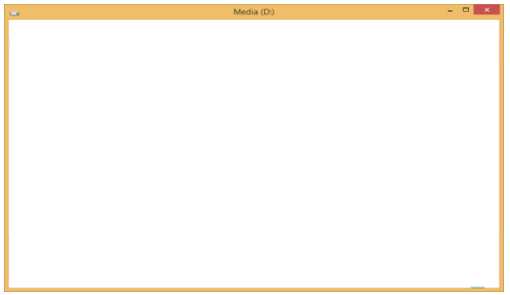
COSA È "GIUSTO"
CHE ACCADA?



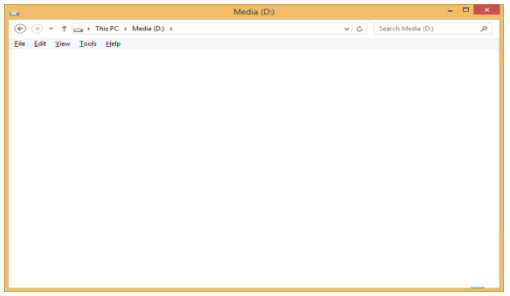
COSA "È GIUSTO" CHE ACCADA ?

Consideriamo uno scenario più interessante:

- classe **`Window`** che rappresenta una finestra grafica base
  - definisce il metodo **`move`** per spostarla in un'altra posizione



- sottoclasse **`WindowWithMenuBar`** che aggiunge la *barra dei menù*
  - ridefinisce il metodo **`move`** affinché sposti anche la barra dei menù



Sia ora **`w`** un riferimento a **`Window`**: come tale, potrebbe puntare *sia a una finestra base*, *sia a una con menù*.

COSA "È GIUSTO" CHE ACCADA ?

Java

~C#

Scenario banale:

```
Window w = new Window();
WindowWithMenuBar wm = new WindowWithMenuBar();
```

sicché:

`w.move()` sposta la finestra base

`wm.move()` sposta la finestra con menù

Scenario più interessante:

```
w = wm;
```

sicché **`w`** è:

formalmente, una finestra base

in realtà, una finestra con menù

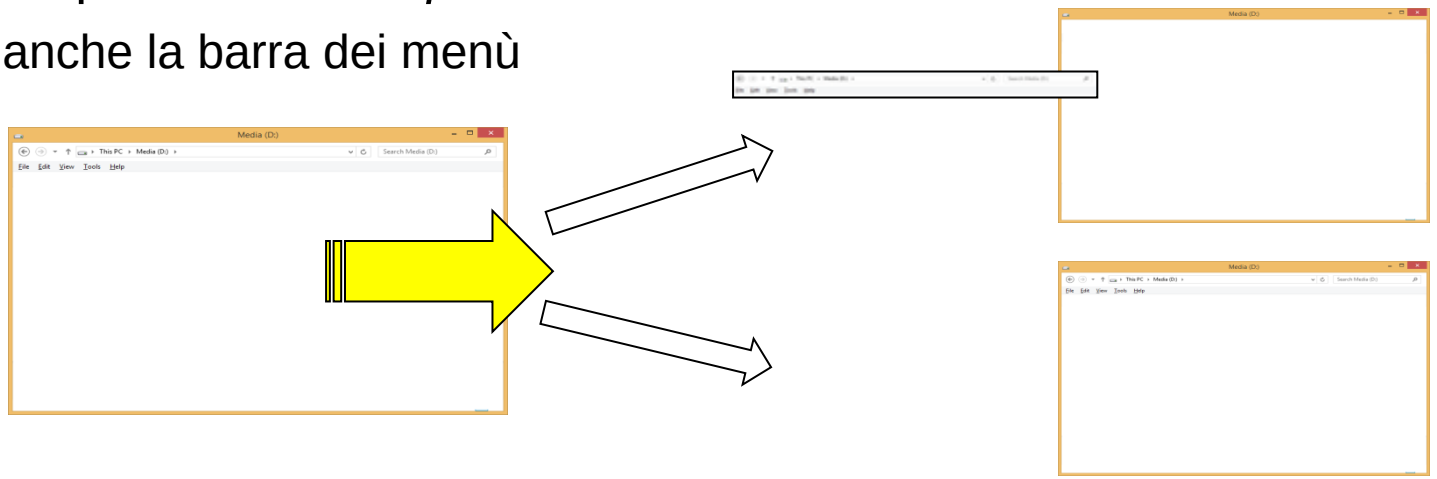
Cosa vogliamo che accada invocando…?

```
w.move();
```

COSA "È GIUSTO" CHE ACCADA ?

Scenario interessante:

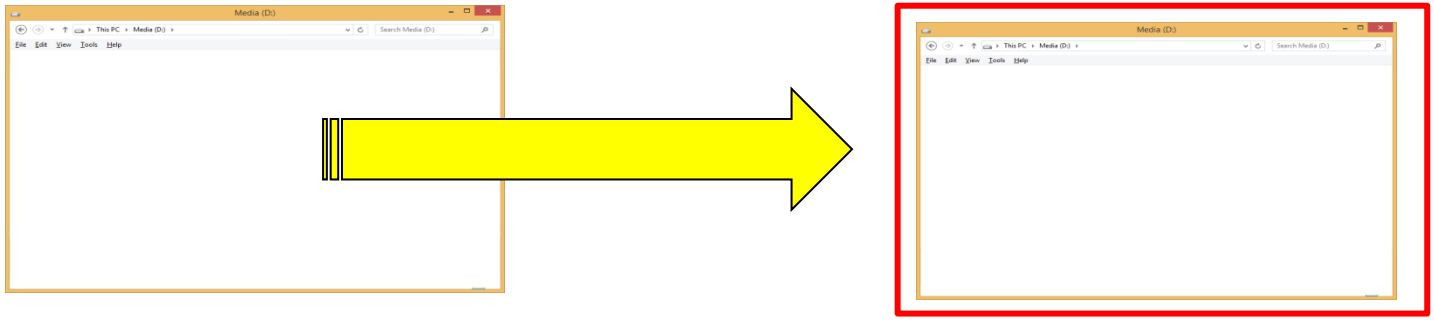
- se prevale la forma (**`w`** è formalmente una **`Window`**), deve spostarsi solo la finestra base
  - la barra dei menù, sconosciuta alla **`move`** di **`Window`**, rimane dov'è
- se prevale la sostanza (**`w`** è in realtà una **`WindowWithMenuBar`**), deve spostarsi *tutto quanto*
  - anche la barra dei menù



COSA "È GIUSTO" CHE ACCADA ?

Scenario interessante:

- se prevale la forma (**`w`** è formalmente una **`Window`**), deve spostarsi solo la finestra base
  - la barra dei menù, sconosciuta alla **`move`** di **`Window`**, rimane dov'è
- se prevale la sostanza (**`w`** è in realtà una **`WindowWithMenuBar`**), deve spostarsi *tutto quanto*
  - anche la barra dei menù



POLIMORFISMO

- Un metodo si dice *polimorfo* quando è in grado di *adattare il suo comportamento allo specifico oggetto* su cui opera.
- In Java e C#, questa possibilità si apre *proprio grazie alla compatibilità di tipo fra oggetti di classi derivate*

COME?

- Usando un riferimento a una classe-base (es. **`Persona`**) per puntare a oggetti *di classi più specifiche* (es. **`Studente`**)
- Ogni metodo (es. **`print`**) definito nella prima e ridefinito nella seconda *si comporta in modo polimorfo*

LA SOSTANZA PREVALE SULLA FORMA

L'IDEA DI FONDO

- NON fermarsi alla superficie delle cose
  ossia, al tipo formale del riferimento
  - in tal caso, **`p.print()`** stamperebbe solo nome ed età, perché scatterebbe la **`print`** della classe **`Persona`**
- MA considerare invece la loro *sostanza*
  ossia, il tipo dell'oggetto *realmente referenziato*
  - **`p.print()`** stampa nome, età e matricola perché scatta la **`print`** di **`Studente`** *in quanto ci si accorge che si tratta di uno Studente*

PREREQUISITO: *rimandare a runtime* la decisione su quale metodo **`print`** debba effettivamente essere chiamato, perché *solo a runtime si può sapere a cosa stia davvero puntando* **`p`** *in quel momento*

Tecnica di *LATE BINDING* (default in Java & Scala, abilitata caso per caso con la keyword **`virtual`** in C# o la keyword **`open`** in Kotlin)

UN ESPERIMENTO IN JAVA

Java

```
public class EsempioDiCitta {
 publi
 Perso
 Stude
  p.print();  // stampa nome ed età
  p=s;
  p.print(); // stampa nome, età, matricola
 }
}
```

Inizialmente, **p** referenzia una **Persona**: quindi, nessun dubbio che venga chiamato il metodo **print** di **Persona**
→ si stampano solo *nome ed età*

In questo momento invece **p** referenzia uno **Studente**, ergo viene chiamato il metodo **print** di **Studente**
→ si stampano nome, età *e matricola*

L'ESPERIMENTO IN C#...

C#

```
public class Persona {
  ...
 public virtual void print() {
   Console.Write (”Mi chiamo " + nome);
   Console.WriteLine(" e ho " +anni+ "anni");
 }
}
```

polimorfo

C#

```
public class Studente : Persona {
  ...
 public override void print() {
   base.print();
   Console.WriteLine("Matricola = " + matr);
 }
}
```

polimorfo

…E IN SCALA / KOTLIN

Scala

```
def main(args: Array[String]) : Unit = {
   var p = new Persona("John");
   var s = new Studente("Tom");
   p.print();  // stampa nome ed età
   s.print();  // stampa nome, età, matricola
   p=s;        // nessuna sorpresa(Studente estende Persona)
   p.print();  // stampa polimorfa
}
```

Kotlin

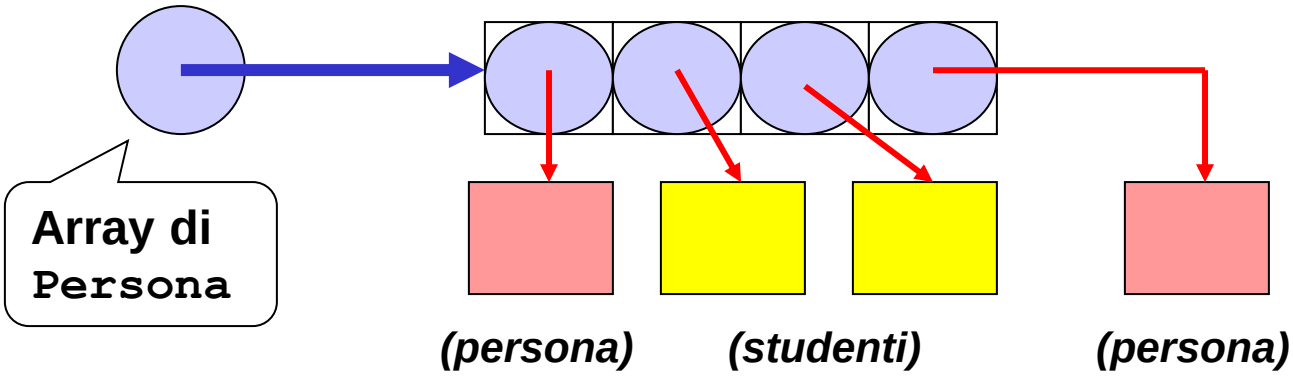
```
fun main(args: Array<String>) : Unit {
  var p = Persona("John");
  var s = Studente("Tom");
   p.print();  // stampa nome ed età
   s.print();  // stampa nome, età, matricola
   p=s;        // nessuna sorpresa(Studente estende Persona)
   p.print();  // stampa polimorfa
}
```

LATE BINDING (collegamento ritardato)

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`** sta puntando a uno **`Studente`**)

Quindi, ad esempio, in un array di **`Persona`**, **`v[i].print`** *invocherà ogni volta il metodo appropriato all'oggetto i-esimo:*



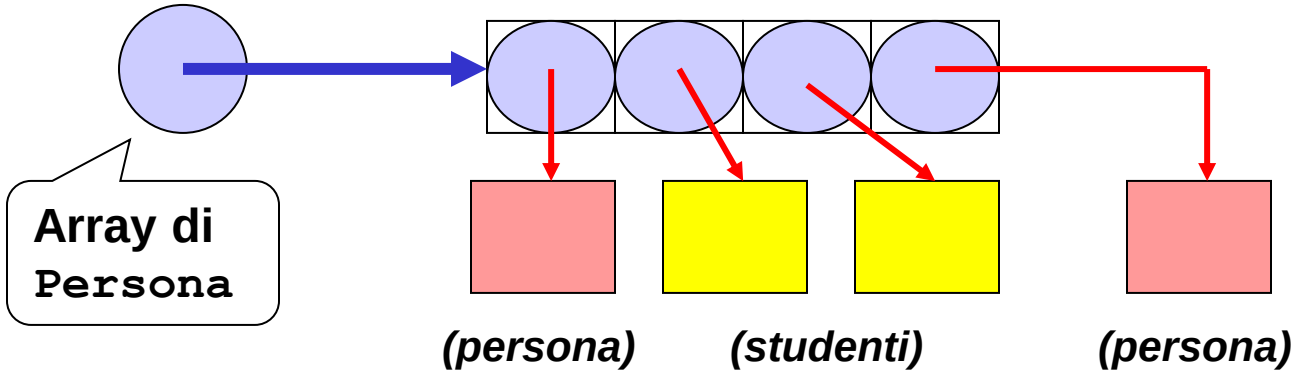
UN ESEMPIO PIÙ CONVINCENTE

Java

```
Persona[] persone =        {
  new Persona("John"), new Studente("Jeff", 25, 98461),
  new Studente("Anna", 20, 17459), new Persona("Emma",4)};
for (Persona p : persone) p.print();
```

~C#

```
Mi chiamo John e ho 0 anni
Mi chiamo Jeff e ho 25 anni Matricola = 98461
Mi chiamo Anna e ho 20 anni Matricola = 17459
Mi chiamo Emma e ho 4 anni
```



UN ESEMPIO PIÙ CONVINCENTE

Java

```
Persona[] persone =       {
  new Persona("John"), new Studente("Jeff", 25, 98461),
  new Studente("Anna", 20, 17459), new Persona("Emma",4)};
for (Persona p : persone) p.print();
```

~C#

```
val persone = Array( new Persona("John"),new Studente("Tom"),
                     new Studente("Anna"),new Persona("Jane")
                     );
for (p <- persone) p.print();
```

Scala

```
val persone = arrayOf( Persona("John"), Studente("Tom"),
                  Studente("Anna"),Persona("Jane"));
 for (p in persone) p.print();
```

Kotlin

LATE BINDING: TECNICA

COME FUNZIONA

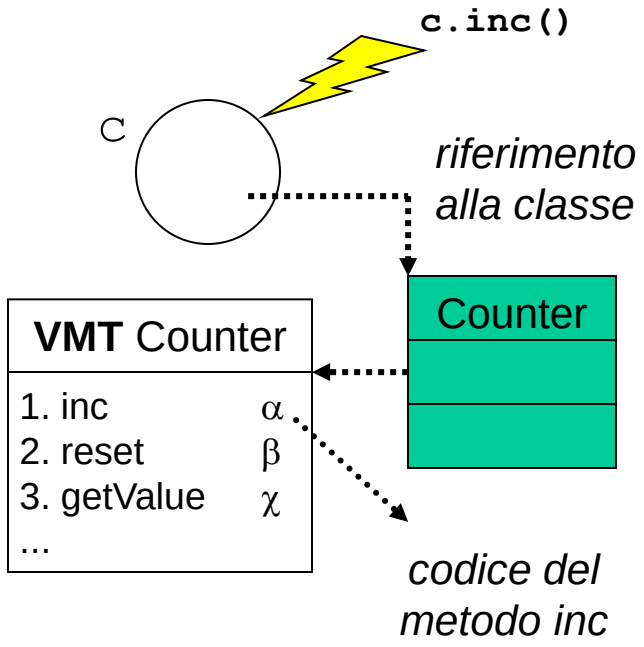
- il compilatore *non collega più a priori* le chiamate con una *specifica funzione* (Binding Statico o *Early Binding*),
- ma *predispone il necessario* perché *a run time* si possa chiamare al volo il metodo “giusto” per l'oggetto

COME FA?

- Facile: a ogni classe è associata una tabella (VMT = Virtual Method Table) in cui per ogni *nome di metodo polimorfo* si specifica *dove trovare il suo codice*
  - per chiamare un metodo basta *consultare la giusta riga della tabella*
  - grande risultato a un costo trascurabile: una chiamata indiretta!

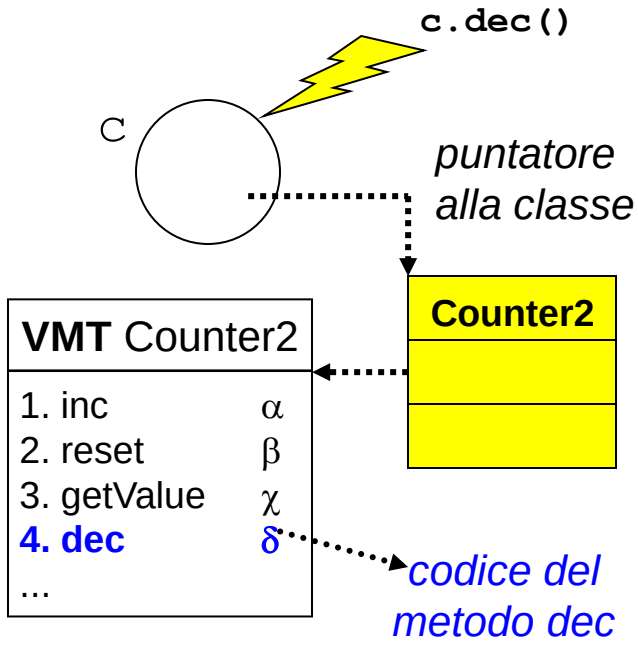
LATE BINDING: FUNZIONAMENTO (1)

- Ogni oggetto contiene un riferimento alla sua classe
- Nella VMT sono elencati i *nomi dei metodi* della classe con il corrispondente *indirizzo del codice*
- Chiamare un metodo è facile:
  - si accede alla tabella VMT della classe cui appartiene l'oggetto
  - si accede alla riga della tabella che corrisponde al metodo chiamato e si ricava il riferimento al suo codice
  - si invoca il metodo identificato
  - costo: UNA chiamata indiretta!



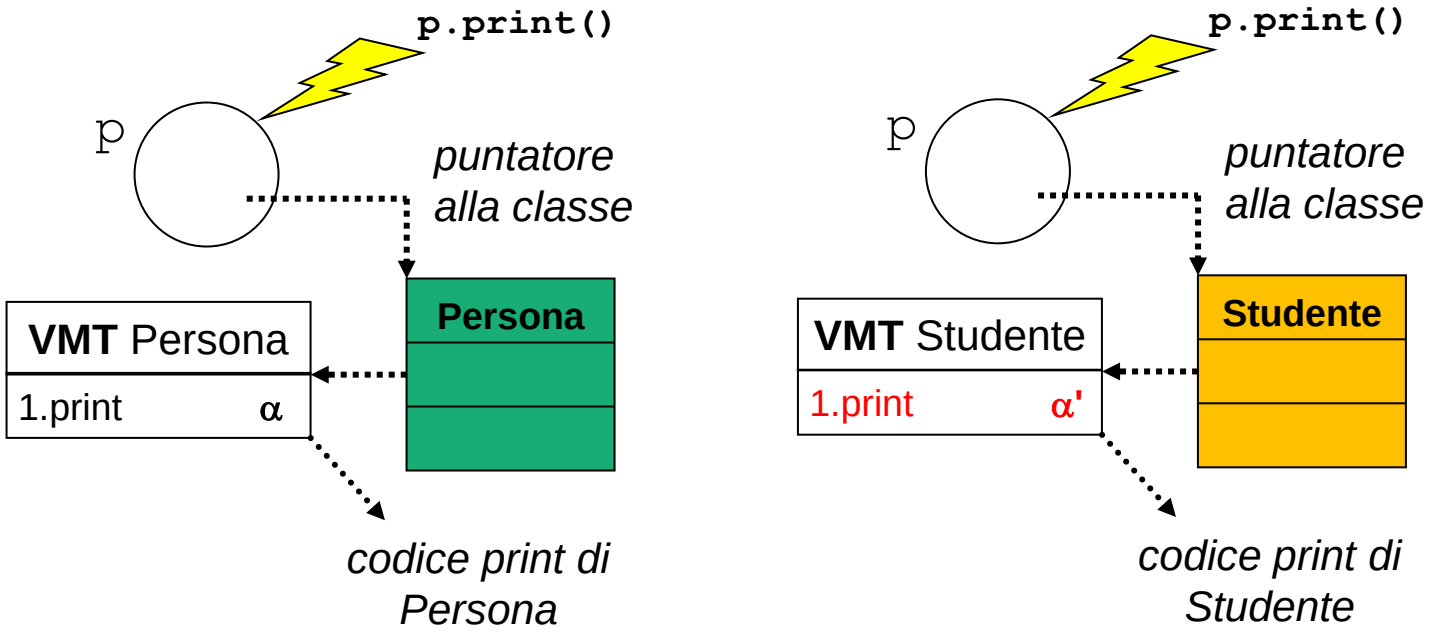
LATE BINDING: FUNZIONAMENTO (2)

- Ogni classe derivata estende la VMT della classe base
  - aggiunge in coda nuove righe per i nuovi metodi (es. **`dec`**)
  - …



LATE BINDING: FUNZIONAMENTO (3)

- Ogni classe derivata estende la VMT della classe base
  - .. e cambia l'indirizzo dei metodi ereditati che ridefinisce, lasciandoli però nella stessa riga
  - così, la chiamata **`p.print()`** si traduce in *"chiama il metodo #1"*



GERARCHIE DI EREDITARIETÀ

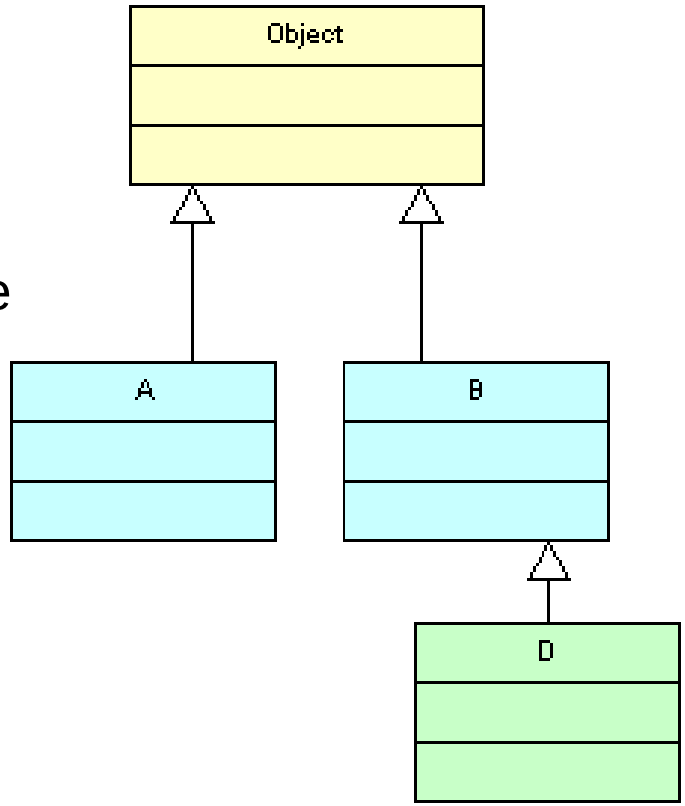
- La relazione di ereditarietà determina la nascita di *gerarchie* (*tassonomie*) di ereditarietà.
- Nei linguaggi OOP, ogni classe deriva implicitamente da una qualche classe base, che è la *radice della gerarchia*
  - *quale sia tale classe* e *come sia fatta la tassonomia* sono elementi che possono *variare* da un linguaggio all'altro
- La radice della gerarchia si chiama
  - in Java: `Object`
  - in C#: `Object`
  - in Scala: `Any`
  - in Kotlin: `Any`

Radice dei soli tipi-riferimento (non dei tipi primitivi)

Radice di tutti i tipi (*everything is an object:* non esistono più tipi primitivi!)

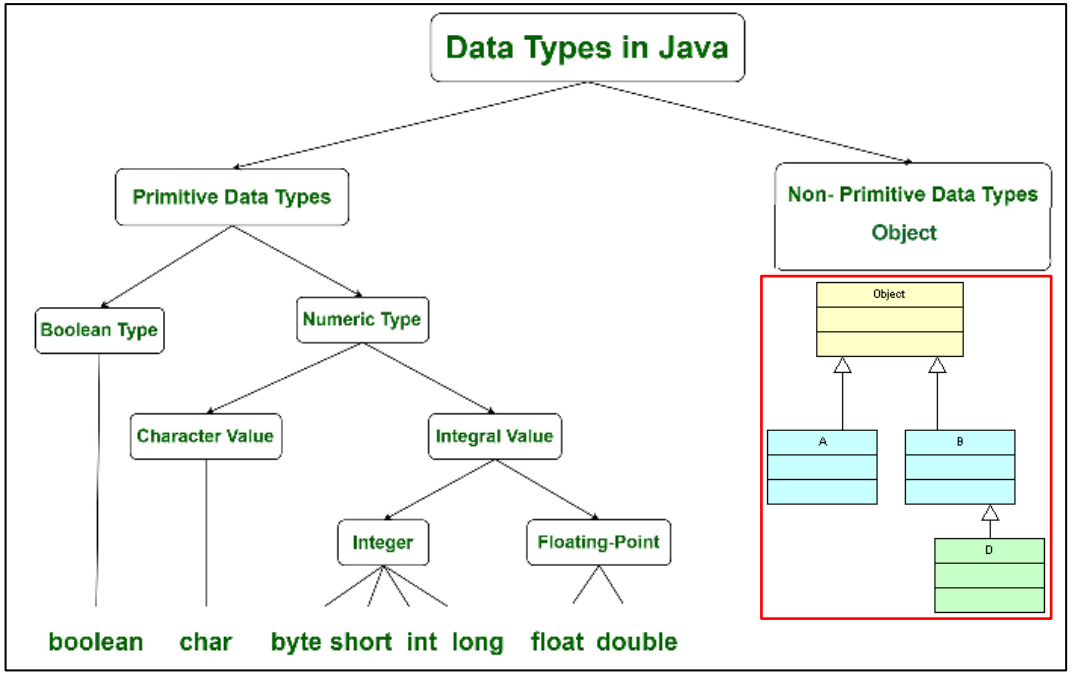
GERARCHIE DI EREDITARIETÀ

- Perciò, la frase "**`class A`**" sottintende sempre «*deriva dalla top class*»
  - in C#, Scala, Kotlin derivano da essa *anche le classi dei tipi-base (gli ex tipi primitivi);* in Java no
  - è nella top class che vengono definite le funzionalità «*predefinite*», come **`toString`**, **`equals`**, **`clone`**
    - grazie all'ereditarietà, esse sono quindi disponibili automaticamente in qualunque tipo di oggetto



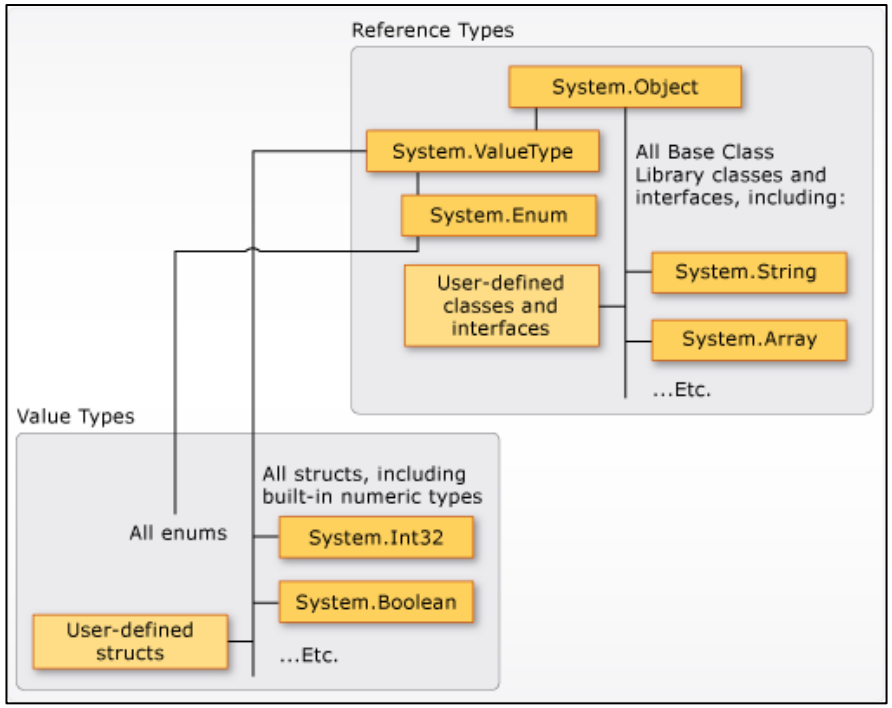
GERARCHIE DI EREDITARIETÀ

- Java
  - i tipi primitivi *non* sono oggetti
  - `Object` è il tipo-base capostipite di tutti i *tipi riferimento*
  - la frase "`class A`" sottintende "`extends Object`"



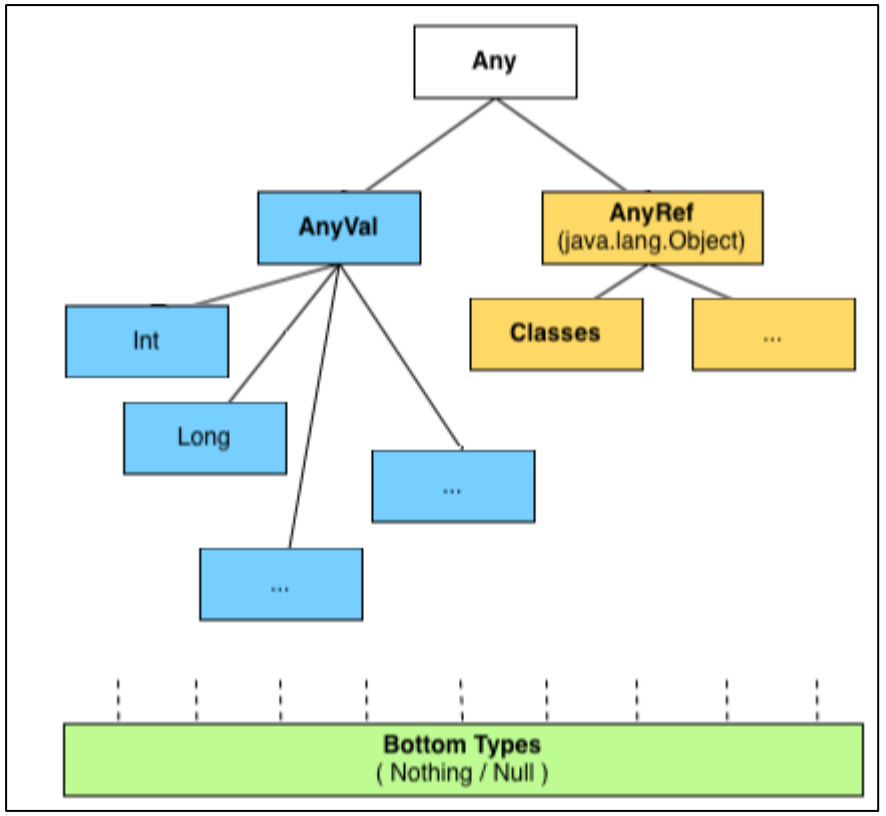
GERARCHIE DI EREDITARIETÀ

- C# (.NET)
  - **`Object`** è il tipo-base generale, specializzato in particolare dalla sottoclasse **`ValueType`**
  - **`ValueType`** è la sottoclasse dei *tipi valore*, corrispondenti ai tipi primitivi di Java *ma promossi a veri tipi di oggetto*
  - ai tipi valori è associato per comodità un alias, che diventa una *keyword riservata*:
    - ad es. **`System.Object`** → **`object`**
    - ad es. **`System.Int32`** → **`int`**
    - …



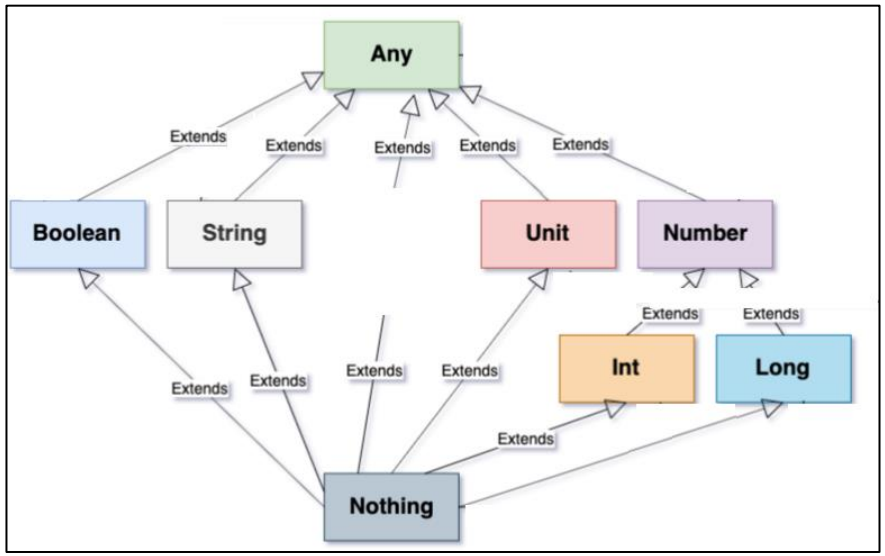
GERARCHIE DI EREDITARIETÀ

- Scala
  - **`Any`** è il tipo-base generale, specializzato dai due sottocasi **`AnyRef`** e **`AnyVal`**
  - **`AnyRef`** è la classe base dei *tipi riferimento*, corrispondente a **`Object`** in Java
  - **`AnyVal`** è la classe base dei *tipi valore*, corrispondenti agli ex tipi primitivi di Java
  - Le due «bottom classes» **`Null`** e **`Nothing`** sono sottotipi di qualunque altra classe
    - servono a «chiudere» il type system nei casi limite



GERARCHIE DI EREDITARIETÀ

- Kotlin
  - **`Any`** è il tipo-base generale, *circa* equivalente a **`Object`** di Java
  - da essa derivano sia i *tipi valore* sia i *tipi riferimento*
  - La «bottom class» **`Nothing`** è sottotipo di ogni altra classe
    - non c'è la classe Null

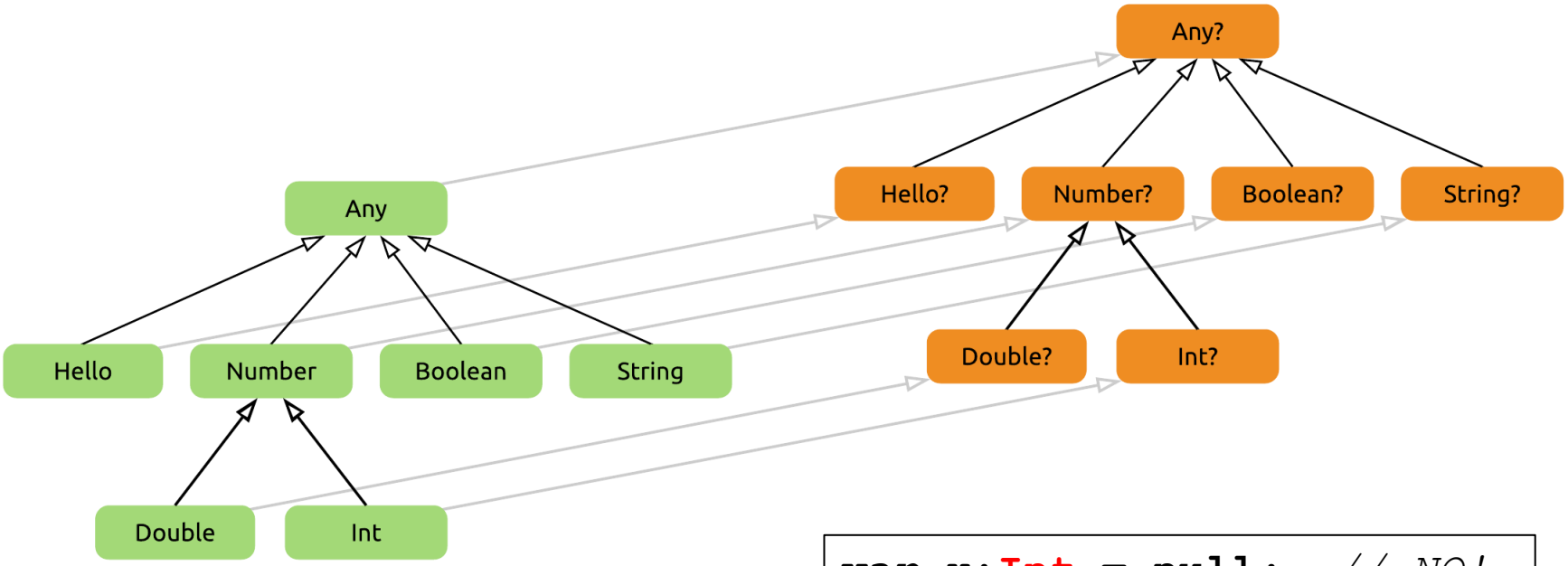


  - peculiarità: per garantire *null safety*, i riferimenti alle classi di norma *non possono essere null («non-nullable classes»)*
  - per rappresentare i casi in cui ciò è necessario, a ogni classe ne viene affiancata un'altra *nullable*, caratterizzata dal *suffisso* **`?`**

GERARCHIE DI EREDITARIETÀ

- Kotlin:

non-nullable classes vs. nullable classes




```
var x:Int = null;   // NO!
var y:Int? = null;  // OK
```

Java: METODI "PREDEFINITI"

protected Object	clone()	Creates and returns a copy of this object.
boolean	equals(Object obj)	Indicates whether some other object is "equal to" this one.
int	hashCode()	Returns a hash code value for the object.
String	toString()	Returns a string representation of the object.

Java

- **`toString`**: l'implementazione predefinita stampa un identificativo univoco dell'oggetto: ad esempio, **`Counter@712c1a3c`**
- **`equals(Object obj)`**: l'implementazione predefinita confronta i riferimenti, in modo del tutto identico all'operatore **`==`**
- **`hashCode()`**: restituisce un intero rappresentativo dell'oggetto; è imperativo che sia coerente con **`equals`**
- **`clone()`** è la base per clonare oggetti: poiché sta al progettista stabilire se una data classe possa essere *clonabile*, è protetta (non pubblica)
  - per abilitare questa funzionalità, occorre ridefinirla come **`public`** nella classe derivata, invocando questa tramite **`super`**
    (e implementando l'interfaccia **`Cloneable`** – vedremo…)

METODI "PREDEFINITI" NEGLI ALTRI LINGUAGGI

C#

Equals(Object)	Determines whether the specified object is equal to the current object.
Equals(Object, Object)	Determines whether the specified object instances are considered equal.
GetHashCode()	Serves as the default hash function.
GetType()	Gets the Type of the current instance.
MemberwiseClone()	Creates a shallow copy of the current Object.
ReferenceEquals(Object, Object)	Determines whether the specified Object instances are the same instance.
ToString()	Returns a string that represents the current object.

METODI "PREDEFINITI" NEGLI ALTRI LINGUAGGI

Scala

```
final def !=(arg0: Any): Boolean
          Test two objects for inequality.

final def ##: Int
          Equivalent to x.hashCode except for boxed numeric types and null.

final def ==(arg0: Any): Boolean
          Test two objects for equality.

final def asInstanceOf[T0]: T0
          Cast the receiver object to be of type T0.

      def equals(arg0: Any): Boolean
          Compares the receiver object (this) with the argument object (that) for equivalence.

      def hashCode(): Int
          Calculate a hash code value for the object.

final def isInstanceOf[T0]: Boolean
          Test whether the dynamic type of the receiver object is T0.

      def toString(): String
          Returns a string representation of th
```

Kotlin

```
public open class Any {

    public open operator fun equals(other: Any?): Boolean
    public open fun hashCode(): Int
    public open fun toString(): String
}
```

Un caso di studio in Java: numeri reali & numeri complessi

UN CASO DI STUDIO in Java

- Immaginiamo che si voglia progettare una classe per rappresentare il concetto di *numero reale.*

Perché? Non bastano i tipi
`float` e **`double`**.. ?

- NO, non bastano, perché:
  - in Java, **`float`** e **`double`** sono tipi primitivi, *non classi*
    → non dispongono di metodi
  - anche negli altri linguaggi, in cui lo sono, la loro *collocazione nella gerarchia dei tipi* potrebbe *non riflettere le nostre esigenze*
    → impossibilità di sfruttare appieno ereditarietà e polimorfismo
  - in Java, i valori float e double non sono oggetti, quindi non si possono inserire in Liste, Mappe.. Collezioni di oggetti

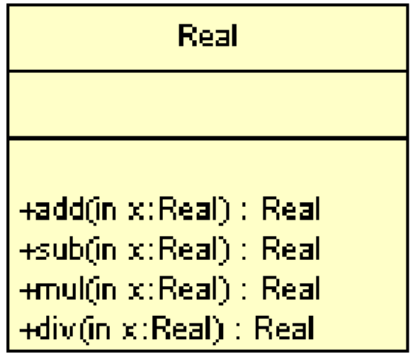
DALL'ANALISI AL PROGETTO

ANALISI DEL DOMINIO DEL PROBLEMA

- I numeri sono *valori,* non variabili!
  - i numeri non si trasformano: 2 sarà sempre 2, 3.14 sempre 3.14
  - i numeri si combinano fra loro tramite *operazioni* che *producono un nuovo numero* (il *risultato*) in base alla loro semantica

PROGETTO

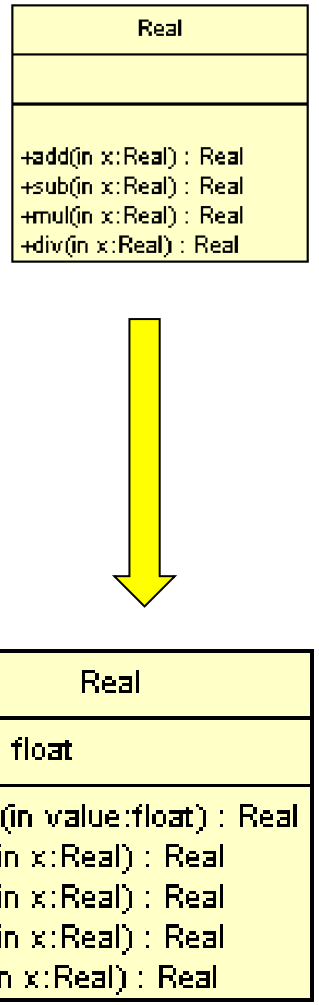
- La classe **`Real`** cattura l'idea di numero reale *immodificabile*
  - il costruttore inizializza il valore
  - un metodo per ogni operazione
    → **`sum`**, **`sub`**, **`mul`**, **`div`**
  - nessun metodo altera l'oggetto corrente (**`this`**): si generano sempre nuovi valori



DAL PROGETTO ALL'IMPLEMENTAZIONE

SCELTE CONCRETE

- Come rappresentare internamente il valore?
  - con un **`float`**
  - con un **`double`**
  - con una coppia di frazioni (analisi matematica!)
  - …
- Stabiliamo di usare un **`float`**
  - quindi anche il costruttore prenderà in ingresso un **`float`**
  - può forse essere utile un secondo costruttore ausiliario che accetti un valore **`double`**..?



DALL'IMPLEMENTAZIONE AL CODICE

Java

```
class Real {
 protected float val;
 public Real(float value) { val = value; }
 public Real sum(Real that){
   return new Real(this.val + that.val); }
 public Real sub(Real that){
   return new Real(this.val - that.val); }
 public Real mul(Real that){
   return new Real(this.val * that.val); }
 public Real div(Real that){
   return new Real(this.val / that.val); }
 public String toString(){ return "" + val; }
 …
}
```

MODIFICA DEI REQUISITI

- Supponiamo ora che il contesto applicativo cambi e sia necessario disporre anche dei *numeri complessi*

UN PRIMO APPROCCIO (corretto...?)

- I numeri complessi hanno una parte reale, come i reali, *ma anche una parte immaginaria* → *estendono i reali*
- Secondo questo approccio, la classe **`Complex`** potrebbe quindi essere una sottoclasse di **`Real`**
  - eredita la parte reale e i metodi corrispondenti
  - aggiunge come nuovo dato la parte immaginaria
  - definisce i nuovi metodi

SARÀ L'APPROCCIO GIUSTO?

COMPLEX DA REAL …?

TECNICAMENTE, non ci sono problemi:

Java

```
public class Complex extends Real {
 protected float im;
 public Complex(float r, float i){ super(r); im = i; }
 public Complex sum(Complex that){
   return new Complex(this.val+that.val, this.im+that.im); }
 // analogamente per le altre operazioni
 public String toString(){
   return super.toString() + "+i" + im; }
 …
}
```

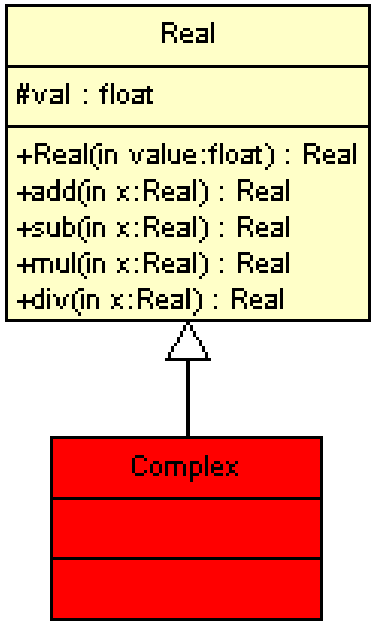
COMPLEX DA REAL …? ! ??

..PECCATO che non stia in piedi: è tutto al contrario!

- si può assegnare un complesso a un reale
  - orrore: non ha senso matematicamente!
  - e nel farlo, la parte immaginaria va persa!! Argh!
- ma non un reale a un complesso
  - che invece avrebbe perfettamente senso
  - e non farebbe perdere informazione!

COSA È SUCCESSO?
COME È POTUTO ACCADERE??

DOVE ABBIAMO SBAGLIATO?



LA GENESI DEL PROBLEMA

- In fase di analisi, siamo partiti dalla frase

  "I numeri complessi hanno una parte reale, come i reali, ma anche una parte immaginaria"

- che era corretta, ma poi ne abbiamo dedotto che quindi

  " estendono i reali"

- che invece è completamente falso e arbitrario!
  - avere una proprietà in più non significa "estendere" un insieme!
  - ANZI: gli oggetti che hanno una proprietà in più sono *di meno* di quelli che hanno meno proprietà!
  - Infatti, matematicamente i complessi non sono un sotto-insieme dei reali: è esattamente l'opposto!
  - Il linguaggio naturale può ingannare: abbiamo frainteso perché abbiamo agito *senza riflettere sulle relazioni del dominio*

REVISIONE DELL'ANALISI

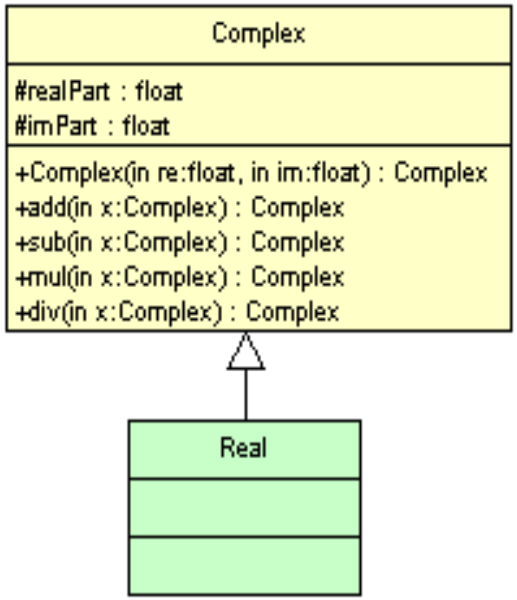
- Introdurre una proprietà in più non indica necessariamente che ci si restringa a un sottoinsieme: può indicare anche la necessità di *ampliare il dominio*, andando verso un *sopra-insieme più grande*
- Nel caso in questione:
  - la parte immaginaria ce l'hanno anche i reali, tanto è vero che *sappiamo perfino quanto vale* (zero)
  - quello che ci ha fuorviato è *che nei reali non si indica,* ma è così solo perché è sottintesa, non perché non ci sia!
- Un conto è non scrivere una proprietà *perché un oggetto non la possiede*, tutt'altro farlo perché *si sa a priori quanto vale*: sono due situazioni diametralmente opposte!

ANALISI DEL DOMINIO ESTESO

- Un'analisi corretta deve partire dal dominio del problema, *così come codificato dagli esperti del settore*.
- In matematica, è assodato che debba essere $R \subset C$

  → qualunque modello software corretto *deve partire da questa relazione*
- In OOP, la relazione "sottoinsieme di" è espressa dall'ereditarietà "IS-A"

  → **`Complex`** *deve generalizzare* **`Real`**
  - non viceversa
  - ogni altra considerazione (efficienza, etc.) viene dopo



ANALISI GENERALE

- La classe (base) **`Complex`**
  - ogni complesso è caratterizzato da *parte reale* e *parte immaginaria (o modulo e argomento)*
  - le operazioni dovranno operare *su qualunque numero complesso* (senza alterare l'oggetto corrente)
- La classe (derivata) **`Real`**
  - ogni reale è anch'esso caratterizzato da *parte reale* e *parte immaginaria*, ma quest'ultima vale sempre 0
  - le operazioni ereditate rimangono applicabili e concettualmente funzionanti, ma – volendo – possono essere semplificate per sfruttare il fatto che la parte immaginaria è 0.

ANALISI DI DETTAGLIO

Estraiamo la conoscenza necessaria dai libri del settore

RELAZIONI DA CONSIDERARE

$(a+ib) \pm (c+id) = (a+c) \pm i\,(b+d)$
$(a+ib) \times (c+id) = (ac-bd) + i\,(bc + ad)$
$(a+ib) / x = (a/x + i\,b/x) \quad \text{se } x \in R$
$(a+ib) / (c+id) = ((a+ib) \times (c-id)) / |(c+id)|^2$
$cgt(a+ib) = (a-ib)$

Operazioni da supportare

- somma, sottrazione, moltiplicazione, divisione fra complessi
- divisione di un complesso per un fattore reale
- coniugato di un numero complesso
- modulo (eventualmente al quadrato) di un numero complesso

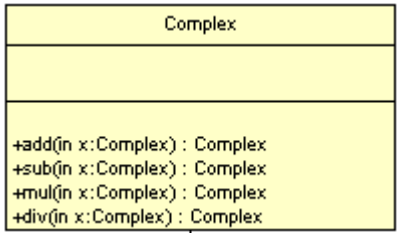
DALL'ANALISI AL PROGETTO

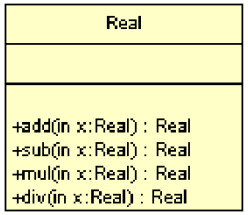
IPOTESI

- come prima, i numeri sono *valori,* non variabili
- le operazioni fra complessi (o fra un complesso e un reale) producono un nuovo numero complesso
- può essere utile (non indispensabile) specializzare le operazioni fra reali, affinché producano un reale

PROGETTO

- In entrambe le classi
  - il costruttore inizializza il valore
  - un metodo per ogni operazione
  - nessun metodo altera l'oggetto corrente (**`this`**): si generano sempre nuovi valori

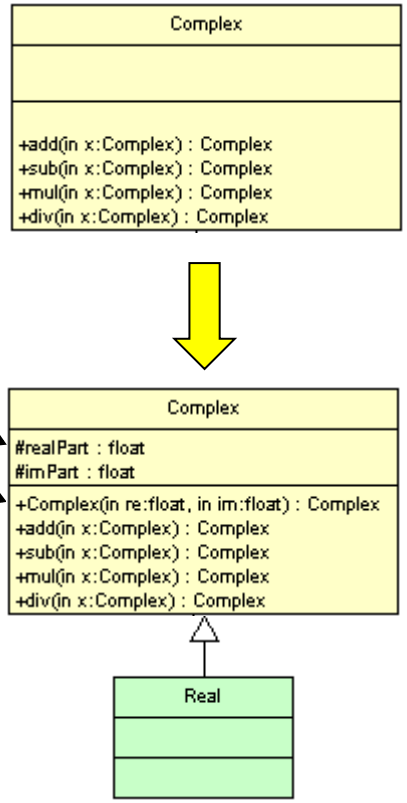




DAL PROGETTO ALL'IMPLEMENTAZIONE

SCELTE CONCRETE: **`Complex`**

- Come rappresentare internamente il valore?
  - un **`float`** per la parte reale,
    un **`float`** per la parte immaginaria
  - quindi anche il costruttore prende
    in ingresso due **`float`**, da interpretare
    come parte reale e parte immaginaria



SCELTE CONCRETE: **`Real`**

- la rappresentazione interna è ereditata
  - la parte immaginaria c'è, ma è sempre 0
  - il costruttore prende in ingresso un solo **`float`**,
    da interpretare come parte reale

Complex: IMPLEMENTAZIONE

Java

```
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;}
}
```

Per brevità, usiamo **z** anziché il classico **that**

`Real`: IMPLEMENTAZIONE

- Tutte le operazioni sono ereditate, e poiché $R \subset C$ funzionano sicuramente anche per i reali.
- Tuttavia, *due operandi reali sarebbero trattati come due complessi qualsiasi,* producendo come risultato un complesso (con parte immaginaria 0) → corretto ma imperfetto
- È opportuno definire nuove operazioni specifiche per i reali
- È utile anche ridefinire **`toString`**, perché non stampi la parte immaginaria per i reali.

Java

```
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; }
}
```

UN MONDO DI NUMERI

Java

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

L'ESEMPIO IN Scala (1/2)

Scala

```
class Complex (val re:Float, val im: Float) {
 def sum(z : Complex) : Complex = { return new Complex(re+z.re, im+z.im); }
 def sub(z : Complex) : Complex = { return new Complex(re-z.re, im-z.im); }
 def mul(z : Complex) : Complex = {
                    return new Complex(re*z.re-im*z.im, im*z.re+re*z.im); }
 def div(z : Complex) : Complex = {
                    return mul(z.cgt()).divByFactor(z.squaredModule()); }
 def cgt() : Complex = { return new Complex(re,-im); }
 def divByFactor(x: Float) : Complex  = { return new Complex(re/x, im/x); }
 def squaredModule() : Float = { return re*re + im*im; }
 override def toString() : String = { return "" + re + "+i" + im;}
}
```

```
class Real(re:Float) extends Complex(re,0F) {
   def sum(x : Real) : Real = { return new Real(re + x.re); }
   def sub(x : Real) : Real = { return new Real(re - x.re); }
   def mul(x : Real) : Real = { return new Real(re * x.re); }
   def div(x : Real) : Real = { return new Real(re / x.re); }
   override def toString() : String = { return "" + re;}
}
```

L'ESEMPIO IN Scala (2/2)

Scala

```
def main(args: Array[String]) : Unit = {
   var r1 = new Real(18.5F);       var r2 = new Real(3.14F);
   var c1 = new Complex(-16F,0F); var c2 = new Complex(3F,2F);
   var c3 = new Complex(0F,-2F);
   var r : Real = r1.sum(r2);       var c : Complex = c1.sum(c2);
   println("r1 + r2 = " + r);       // il reale 21.64
   println("c1 + c2 = " + c);       // il complex -13+2i
   println("c1 + c2 -i = " + c.sub(new Complex(0F,1F))); // il complesso -13+i
   c = c.sum(c3);
   println("c + c3 = " + c);        // -13+0i
   c = r;
   println("c = r; c = " + c);      // qui c è reale
   c = r1.sum(r2);                  // il risultato è un reale, ma c è un Complex
   // r = c.sum(r1);                          // NO
   r = (c.asInstanceOf[Real]).sum(r1);       // SI (cast necessario)
   println("c = r1.sum(r2); r = ((Real)c).sum(r1); r = " + r); // qui c è reale
 }
}
```

L'ESEMPIO IN Kotlin (1/2)

Kotlin

```
open public class Complex (val re:Float, val im: Float) {
 public fun sum(z : Complex) : Complex { return Complex(re+z.re, im+z.im); }
 public fun sub(z : Complex) : Complex { return Complex(re-z.re, im-z.im); }
 public fun mul(z : Complex) : Complex {
                              return Complex(re*z.re-im*z.im, im*z.re+re*z.im); }
 public fun div(z : Complex) : Complex {
                              return mul(z.cgt()).divByFactor(z.squaredModule());}
 public fun cgt() : Complex { return Complex(re,-im); }
 public fun divByFactor(x: Float) : Complex  { return Complex(re/x, im/x); }
 public fun squaredModule() : Float  { return re*re + im*im; }
 public override fun toString() : String { return "" + re + "+i" + im;}
}
```

```
public class Real(re:Float) : Complex(re,0F) {
 public fun sum(x : Real) : Real { return Real(re + x.re); }
 public fun sub(x : Real) : Real { return Real(re - x.re); }
 public fun mul(x : Real) : Real { return Real(re * x.re); }
 public fun div(x : Real) : Real { return Real(re / x.re); }
 public override fun toString() : String { return "" + re;}
}
```

L'ESEMPIO IN Kotlin (2/2)

Kotlin

```
public fun main(args: Array<String>){
  var r1 = Real(18.5F);       var r2 = Real(3.14F);
  var c1 = Complex(-16F,0F); var c2 = Complex(3F,2F); var c3 = Complex(0F,-2F);
  var r : Real = r1.sum(r2);
  var c : Complex = c1.sum(c2);
  println("r1 + r2 = " + r);       // il reale 21.64
  println("c1 + c2 = " + c);       // il complex -13+2i
  println("c1 + c2 -i = " + c.sub(Complex(0F,1F))); // il complesso -13+i
  c = c.sum(c3); println("c + c3 = " + c);          // -13+0i
  c = r;
  println("c = r; c = " + c);      // qui c è reale
  c = r1.sum(r2);                  // il risultato è un reale
  // r = c.sum(r1);                // NO
  r = (c as Real).sum(r1);         // SI (cast non necessario, lo è già)
  println("c = r1.sum(r2); r = (c as Real).sum(r1); r = " + r); // qui c è reale
  r = c.sum(r1);                   // SI
  println("c = r1.sum(r2); r = c.sum(r1); r = " + r); // qui c è reale
}
```

BILANCIO

- Occorre sempre partire da una sana *analisi del dominio*
  - discussione col committente & estrazione informazione da libri e "basi di conoscenza"
  - OBIETTIVO: enucleare le *relazioni insiemistiche* fra le entità
- Occhio alle ambiguità del linguaggio naturale
  In particolare, al significato di "*estendere*"
  - l'estensione *nel senso dell'ereditarietà* delimita un sottoinsieme perché le nuove proprietà che considera *si applicano solo a un sottoinsieme di oggetti più specifici*
  - è cosa del tutto diversa dal considerare nuove proprietà *applicabili a tutti gli oggetti del dominio*, che semplicemente *non erano state prese in considerazione prima*