Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Progettazione incrementale
Ereditarietà

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

UN MONDO IN DIVENIRE

Spesso capita di aver bisogno di un componente *simile* a uno già esistente, ma *non identico*

- abbiamo il contatore che conta in avanti,
  ma *potremmo volerne uno che conti anche all'indietro*
- abbiamo Finestra, ma non va a capo e stampa solo stringhe:
  potremmo volere altri metodi, più evoluti e flessibili

Altre volte, *l'evoluzione dei requisiti* comporta una *modifica* di componenti *già esistenti:*

- necessità di nuovi dati (il colore?) o nuovi comportamenti (metodi)
- necessità di modificare il comportamento di metodi già presenti per adeguarsi alla mutata situazione

COME EVITARE DI RIPROGETTARE
E RIFARE TUTTO DA CAPO?

PROGETTAZIONE INCREMENTALE

È il grande tema della *progettazione incrementale,* che opera *alle differenze* rispetto all'esistente.

Finora, abbiamo solo due possibilità:

- *ricopiare manualmente il codice* della classe esistente e cambiare quel che va cambiato ("copia & incolla")
- *creare un oggetto "composto",* che
  - *incapsuli* il componente esistente
  - *gli deleghi* le operazioni che esso sa già fare
  - *crei sopra di esso le nuove operazioni* che esso non sa fare (sempre che ciò sia possibile..)
- è una forma semplice di ADAPTER
  → un altro *design pattern* molto usato

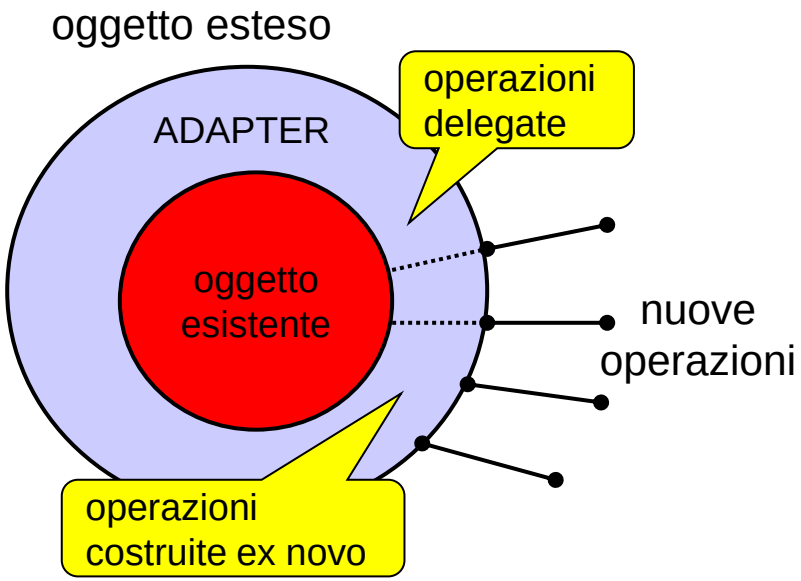
ADAPTER

In generale, un adattatore hardware *adatta* ciò *che si ha* a *ciò che c'è*

- ad esempio, hai la spina italiana ma nel muro c'è la presa britannica o tedesca..

Allo stesso modo, un adattatore software adatta un componente

- *incapsula* l'esistente, per riusarlo
- *gli delega* le operazioni già presenti
- *crea le nuove* sfruttando le funzionalità dell'oggetto incapsulato (se ci riesce..)



DAL CONTATORE "SOLO AVANTI" AL CONTATORE AVANTI / INDIETRO

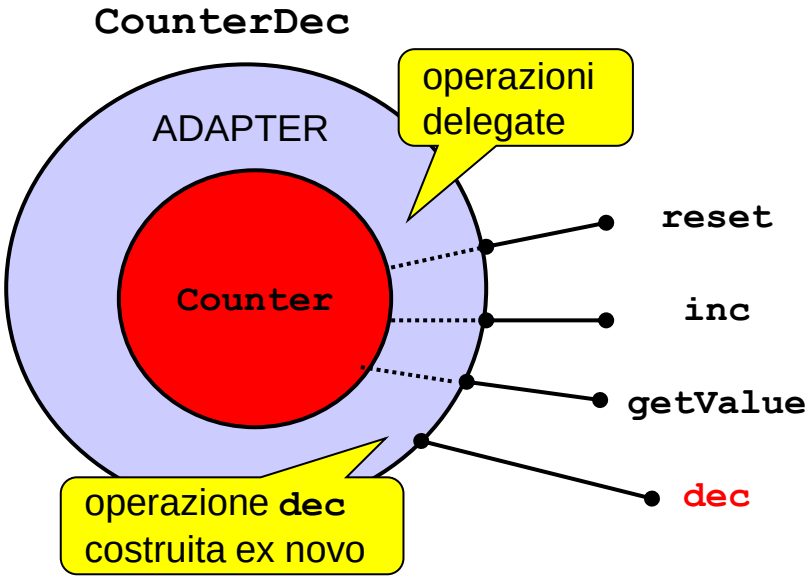
Caso applicativo: il contatore avanti/indietro

- un contatore del tutto simile a **`Counter`** …
- … *ma che offra anche il decremento,* che **`Counter`** non ha

Idea:

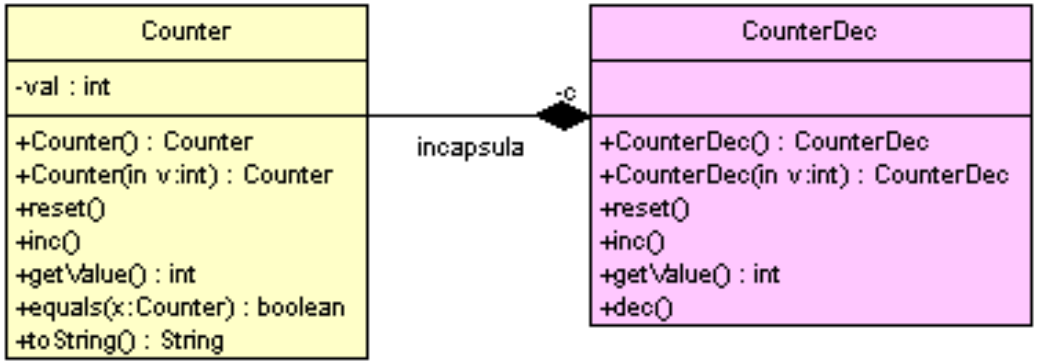
un **`CounterDec`** che adatti **`Counter`**

- incapsula e riusa un **`Counter`**
- le tre operazioni base (**`reset`**, **`inc`**, **`getValue`**), già disponibili, vengono inoltrate all'oggetto interno
- la nuova operazione di decremento (**`dec`**) va invece *realizzata ex novo sfruttando le altre (se si riesce..)*



MODELLO DEL PROGETTO

Con questo pattern, il contatore avanti/indietro è realizzato come "adattamento per composizione" di un Counter



L'oggetto composto, **`CounterDec`**:

- è costruito *costruendo e inglobando* l'oggetto interno
- *pilota* l'oggetto interno incapsulato
- ne riusa le funzionalità, ma orientandole ai suoi nuovi obiettivi
- non è obbligato a mantenere la stessa interfaccia (anche se spesso è così)
- può anche *non offrire* alcune funzioni che erano disponibili in **`Counter`**

IMPLEMENTAZIONE

Java

C#

```
public class CounterDec {
 private Counter c;
 public CounterDec() { c = new Counter(); }
 public CounterDec(int v){ c = new Counter(v); }
 public void reset() { c.reset(); }
 public void inc()   { c.inc(); }
 public int getValue() { return c.getValue(); }
 public void dec() { ... }
}
```

Delega

Come definirla?

COME DEFINIRE IL DECREMENTO?

Non è scontato che le nuova operazione sia fattibile

- dipende se l'entità incapsulata (**`Counter`**) offre funzionalità sufficienti

Si riesce a decrementare il valore?

- non in modo diretto: il campo **`val`** è privato, quindi inaccessibile
- e in modo indiretto..?
  - sì, se ci sono funzioni per (re)impostare il valore del **`Counter`**
  - no, in caso contrario

COME DEFINIRE IL DECREMENTO?

Non è scontato che le nuova operazione sia fattibile

- dipende se l'entità incapsulata (**`Counter`**) offre funzionalità sufficienti

Si riesce a decrementare il valore?

- non in modo diretto: il campo **`val`** è privato, quindi inaccessibile
- e in modo indiretto..?
  - sì, se ci sono funzioni per (re)impostare il valore del **`Counter`**
  - no, in caso contrario

- **`reset`** (grazie di esistere..)
- costruttori

Realizzazione di **`dec`** – prima possibilità:

- recuperare il valore attuale V del contatore **`getValue`**
- riportare il contatore in uno stato iniziale noto (0) **`reset`**
- riportarlo, tramite incrementi, al valore V' = V-1 **`inc`**

DECREMENTO – PRIMA OPZIONE

```
public void dec() {
  int v = c.getValue();
  c.reset();
  for (int i=0; i<v-1; i++) c.inc();
}
```

Java

C#

Problema: e se **reset()** non fosse fornita, o non ponesse il contatore a zero (ma magari a 15..)?

Limite: il contatore non potrà comunque essere decrementato al di sotto del valore impostato da **reset**

Realizzazione di **dec** – prima possibilità:

- recuperare il valore attuale V del contatore **getValue**
- riportare il contatore in uno stato iniziale noto (0) **reset**
- riportarlo, tramite incrementi, al valore V' = V-1 **inc**

DECREMENTO – SECONDA OPZIONE

```
public void dec() {
  c = new Counter( c.getValue() - 1 );
}
```

Java

C#

Problema: e se non ci fosse un costruttore adatto..?

Realizzazione di **`dec`** – seconda possibilità:

- recuperare il valore attuale V del contatore **`getValue`**
- costruire un nuovo contatore, pre-inizializzato al valore V' = V-1 **`Counter(…)`**

UN PICCOLO MAIN DI TEST

Java

~C#

```
public static void main(String[] args){

   Counter c1 = new Counter();                          // 1
   c1.inc(); c1.inc(); System.out.println(c1);          // 3

   CounterDec c2 = new CounterDec();                    // 1
   c2.inc(); c2.inc(); System.out.println(c2);          // 3
   c2.dec();           System.out.println(c2);          // 2

   Counter c3 = new Counter(10);                        // 10
   c3.inc(); c3.inc(); System.out.println(c3);          // 12

   CounterDec c4 = new CounterDec(10);                  // 10
   c4.inc(); c4.inc(); System.out.println(c4);          // 12
   c4.dec();           System.out.println(c4);          // 11
}
```

UN PRIMO BILANCIO

Pro:

- spesso si può riuscire a riusare un componente esistente per costruirne uno nuovo “alle differenze”

ma

- poiché i campi privati non sono accessibili (e comunque non sarebbe opportuno usarli!), occorre *riscrivere anche i metodi che concettualmente rimangono uguali*
  - abbiamo dovuto riscrivere **`inc`**, **`reset`** e **`getValue`** "inutilmente", solo per esprimere la delega dell'operazione al componente interno
- non è detto che le operazioni già disponibili consentano di *ottenere qualsiasi nuova funzionalità*

OCCORRONO FORME DI RIUSO
PIÙ EFFICACI E *SISTEMATICHE*

L'OBIETTIVO

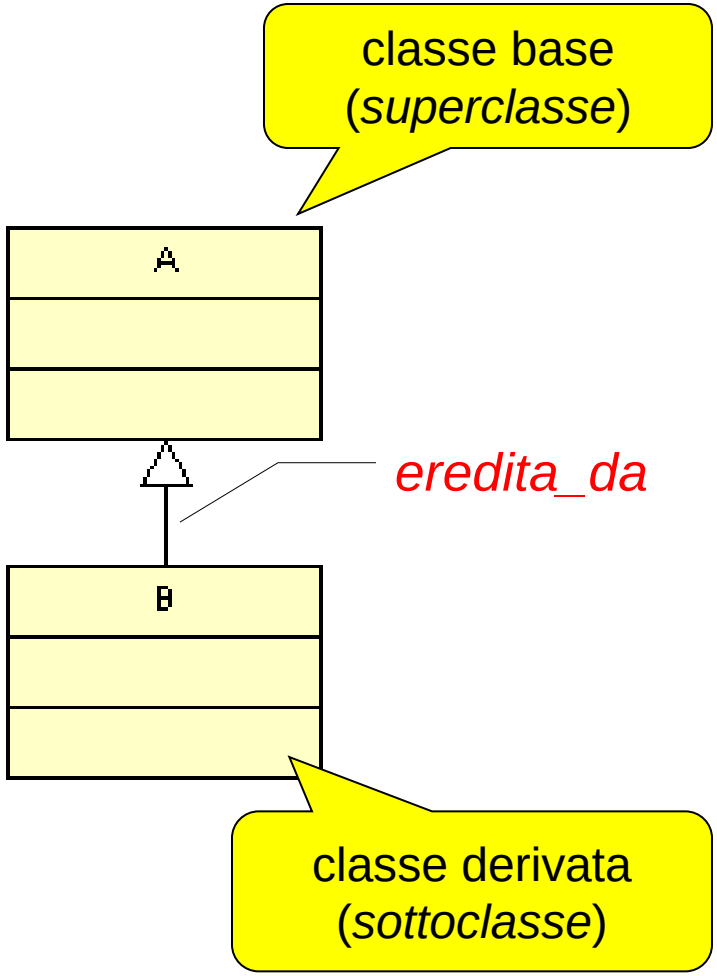
- OBIETTIVO: poter definire una nuova classe *a partire da una già esistente,* senza dover riscrivere "l'ovvio"
- IDEA: definire una nuova classe *estendendone* un'altra, specificando:
  - *quali dati* la nuova classe abbia *in più* dell'altra
  - *quali metodi* la nuova classe abbia *in più* dell'altra
  - *quali metodi* la nuova classe nel caso *modifichi* rispetto all'altra

Questo meccanismo va sotto il nome di *EREDITARIETÀ*

EREDITARIETÀ

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

- Nuova parola chiave:
  - in Java e Scala: **`extends`**
  - in C# e Kotlin: **`:`**



EREDITARIETÀ

- La classe derivata può:
  - *aggiungere* nuovi dati o metodi
  - *accedere* ai dati e ai metodi ereditati (occhio al livello di protezione..)
  - *ridefinire il comportamento di metodi ereditati,* specializzandoli per tenere conto della nuova situazione
- La classe derivata non può:
  - eliminare o togliere alcunché – né dati, né metodi
- La classe derivata condivide *la struttura e il comportamento* (per le parti non ridefinite) della classe base

ESEMPIO

Dal contatore (solo in avanti) …

Java

C#

```
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 livello di protezione..

ESEMPIO

… al contatore avanti/indietro (con decremento)

Java

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

C#

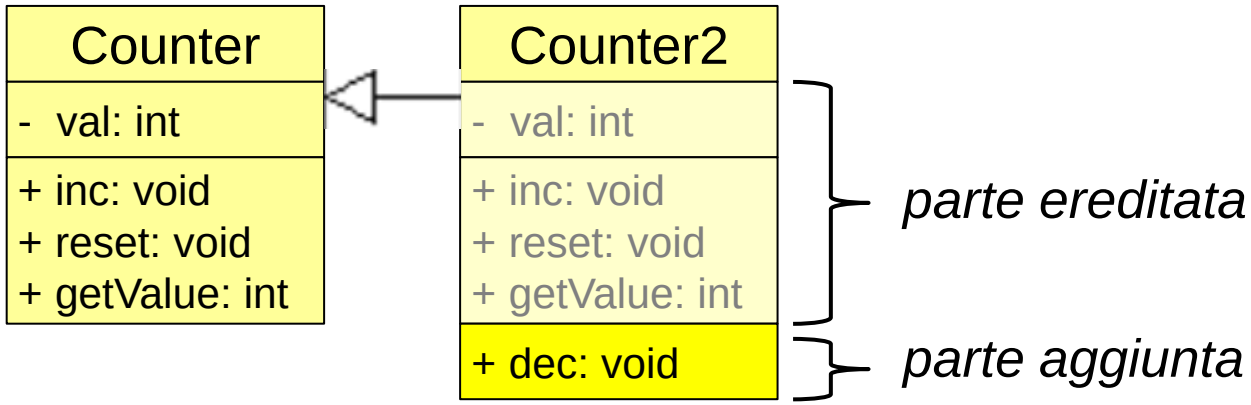
```
public class Counter2 : Counter {
 public void dec() { val--; }
}
```

La nuova classe **`Counter2`** estende **`Counter`**:

- eredita da **`Counter`** il campo **`val`** (un **`int`**)
- eredita da **`Counter`** *tutti i metodi* (**`inc`**, **`reset`**, **`getValue`**)
- introduce in **`Counter2`** il nuovo metodo **`dec()`**

ESEMPIO: STRUTTURA

MODELLO UML



STRUTTURA FISICA

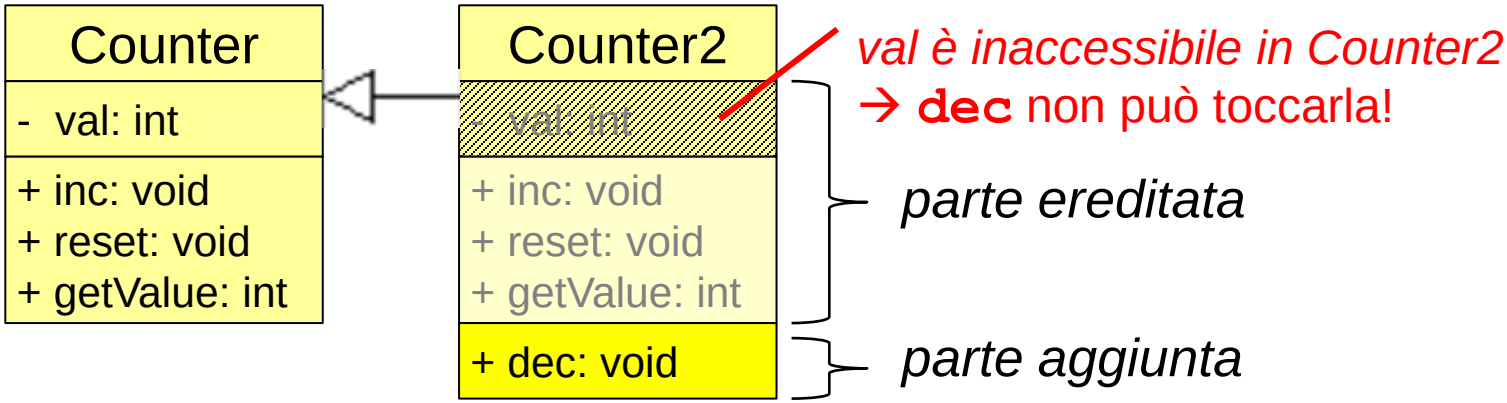
```
Counter c1 = new Counter(); Counter2 c2 = new Counter2();
```

ESEMPIO: PROBLEMA

Problema: **`val`** era privato di **`Counter`**!

`Counter` e **`Counter2`** sono classi *diverse*: nessuno può accedere a dati e metodi *privati* di qualcun altro!

RISULTATO: *non si compila*, perché **`dec`** viola le regole!




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

Java

C#

EREDITARIETÀ E PROTEZIONE

Il punto:

- la qualifica **`private`** impedisce a *chiunque altro* di accedere al dato, *senza distinzioni*
  - va bene per dati “veramente privati”
  - ma *può essere troppo restrittiva* se si ritiene che il componente possa/debba essere esteso in futuro
- Per sfruttare appieno l’ereditarietà occorre *rilassare un po’ il livello di protezione*
  - ci sarebbe la visibilità di package, *ma sarebbe una forzatura* perché obbligherebbe a mettere la classe derivata nello stesso package (e potrebbe essere impossibile)
- Serve una *nuova sfumatura* fra **`private`** e **`public`**

LA QUALIFICA `protected`

Nasce la qualifica **`protected`**

- analoga alla visibilità di package (**`internal`** in C#)
- in più, *consente l'accesso alle eventuali classi derivate,* indipendentemente dal package (namespace) in cui essa è definita.

REFACTORING (revisione del progetto):

- occorre rilassare il livello di protezione del campo **`val`** nella classe **`Counter`**, da **`private`** a **`protected`**

REFACTORING

```
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 livello di protezione: garantisce l'accesso anche a eventuali, future classi derivate.

Java

C#

Ora **Counter2** (immodificato rispetto a prima) si compila e funziona:

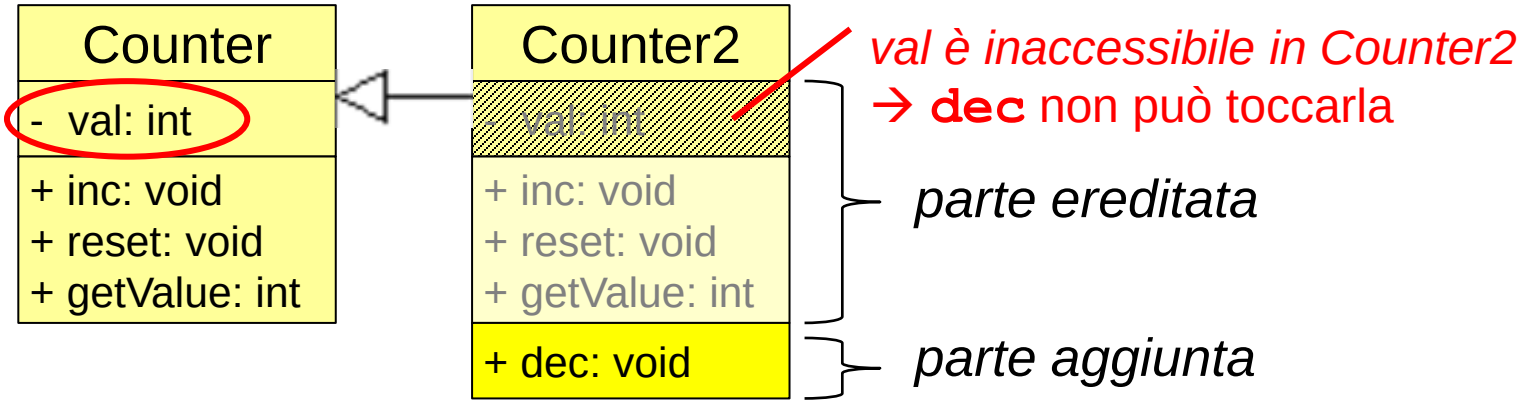
```
public class Counter2 extends Counter {
 public void dec() { val--; }
}
```

Java

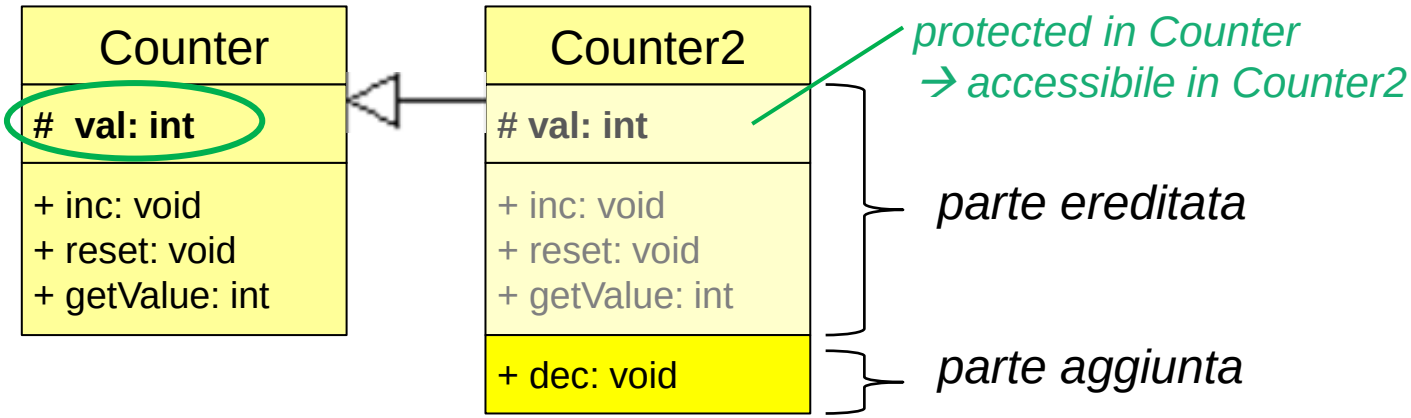
C#

RISULTATO

MODELLO UML originale



MODELLO UML post-refactoring



UN PICCOLO MAIN DI TEST

Java | ~C#

```
public static void main(String[] args){

   Counter c1 = new Counter();                    // 1
   c1.inc(); c1.inc(); System.out.println(c1);    // 3

   Counter2 c2 = new Counter2();                  // 1
   c2.inc(); c2.inc(); System.out.println(c2);    // 3
   c2.dec();           System.out.println(c2);    // 2
}
```

- Funziona come **`CounterDec`**, ma
  - non abbiamo dovuto ridefinire le operazioni che restavano identiche
  - abbiamo dovuto solo introdurre la nuova operazione **`dec`**
  - NB: occhio ai costruttori…

ACCESSO PROTETTO.. DA CHI?

- Però, occorre ATTENZIONE:
  - 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 definita in futuro*, senza distinzione
  - va utilizzata *con giudizio,* non come alternativa "pigra" a private!
- E Javadoc..?
  - i membri **`protected`** *sono citati di default* nella documentazione prodotta da Javadoc
  - al contrario, i membri *privati* o *package* NON sono citati di default (ma possono esserlo specificando l'opportuna opzione)

CAMPI PROTETTI..
o *METODI PROTETTI ?*

- Una *intelligente via di mezzo* consiste nel definire *campi privati* ma con *metodi accessor protetti*
- In questo modo:
  - il dato in sé è *inaccessibile in modo indiscriminato*
  - MA è possibile *usarlo in modo controllato* nelle sottoclassi
  - RISULTATO: si impedisce di violare la semantica del dato
    - ad esempio, che una temperatura sia moltiplicata per -5..
  - MA garantendo al contempo che una classe derivata possa avere accesso se in futuro dovesse servire
    - ad esempio, per leggere o cambiare la temperatura

EREDITARIETÀ & COSTRUTTORI

- Come sappiamo, la classe derivata eredita *dati e metodi* della classe base
  - inclusi quelli privati, a cui però non può accedere direttamente
- Ci sono però cose che *non* si ereditano: i *costruttori*
  - in effetti, non sono metodi: sono *automatismi*
  - come tali, sono specifici della particolare classe in cui sono definiti
  - non per nulla si chiamano come la classe:
    costruire *un oggetto* o *uno simile* non è la stessa cosa!
- Non è affatto detto che il costruttore della classe base *vada bene anche* per inizializzare una sottoclasse
  - la sottoclasse facilmente ha più dati della classe base!

EREDITARIETÀ E COSTRUTTORI (2)

- In effetti:
  - un'istanza della classe derivata potrebbe avere *nuovi dati,* che un costruttore ereditato non conoscerebbe (e quindi non potrebbe inizializzare correttamente)
    - esempio: contatore colorato – se si ereditasse il costruttore di Counter, chi inizializzerebbe il colore?
  - un costruttore è un automatismo per l'inizializzazione (infatti, si chiama come la classe!): *non è un metodo che offre un servizio*
    - un metodo può essere chiamato quando si vuole e tutte le volte che si vuole: un costruttore no! – viene chiamato solo una volta, all'atto della new, automaticamente
    - un metodo viene chiamato esplicitamente da noi (potremmo anche non chiamarlo mai): un costruttore no! – lo chiama il compilatore all'atto della new, e ciò non può in alcun modo essere evitato

ESEMPIO: UN CONTATORE COLORATO

Java

C#

```
import java.awt.Color;

public class ColoredCounter extends Counter {

 private Color color;

   ... // metodi
}
```

Stavolta la classe derivata introduce un nuovo dato: chi lo inizializza?

Se **`ColoredCounter`** ereditasse i costruttori di **`Counter`**, *che non hanno idea della presenza di* ***`color`***, questi resterebbe non inizializzato: pericoloso e inaccettabile!

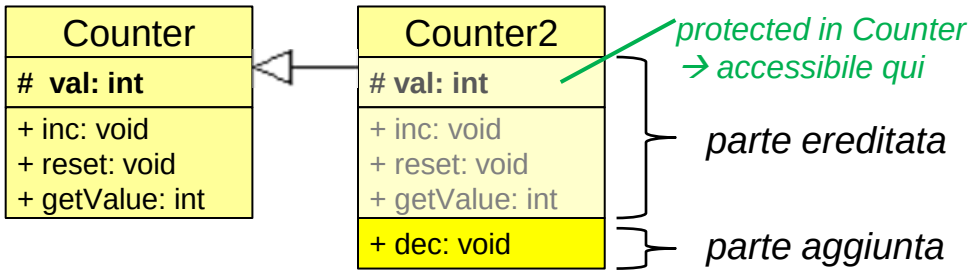
EREDITARIETÀ E COSTRUTTORI (3)

- Dunque, la classe derivata dovrà definire i *suoi* costruttori
  - come ogni classe…
  - naturalmente, anche lei potrà avere un *costruttore di default,* introdotto dal compilatore in assenza di definizioni esplicite
- Tuttavia, poiché alcuni dati sono ereditati, la *sottoclasse non è autonoma* a livello di costruzione
  - la parte ereditata *non è di sua competenza*
  - d'altronde, anche volendo, in presenza di campi privati, non potrebbe neppure farcela, perché non potrebbe accedervi!
- Per questo, ogni costruttore della classe derivata deve «appoggiarsi» a un qualche costruttore della classe base

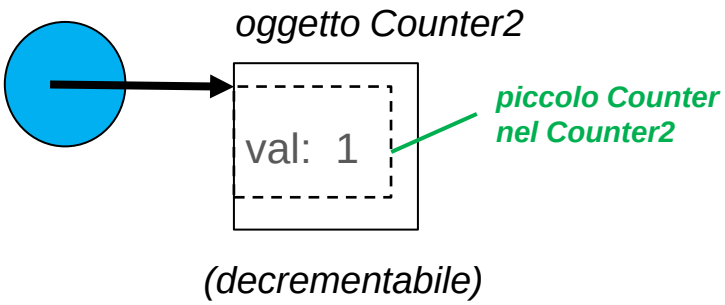
EREDITARIETÀ E COSTRUTTORI (4)

- Concretamente, ogni oggetto della classe derivata comprende al suo interno un oggetto della classe base
- Per questo, essa non è autonoma a livello di costruzione: *qualcuno deve costruirle l'oggetto "interno" ereditato*
  - per costruire un **`Counter2`** bisogna che qualcuno costruisca "il piccolo **`Counter`** che è in lui"!

MODELLO UML



STRUTTURA FISICA



EREDITARIETÀ E COSTRUTTORI (5)

Come ottenere questo?

1. ogni costruttore della classe derivata «*si appoggia*» *necessariamente* a un costruttore della *classe base*
   - che costruisce il “piccolo oggetto base che è in lui”

Cosa vuol dire "si appoggia"?

2. ogni costruttore di classe derivata *specifica un costruttore della classe base* che viene *richiamato automaticamente*
   - se non si specifica niente, viene chiamato quello *di default*
     → se non esiste, *ERRORE DI COMPILAZIONE*

Come si specifica il costruttore da chiamare?

3. tramite la parola chiave **`super`** (**`base`** in C#)

UN PICCOLO MAIN DI TEST

```
public static void main(String[] args){

   Counter c1 = new Counter();                    // 1
   c1.inc(); c1.inc(); System.out.println(c1); // 3

   Counter2 c2 = new Counter2();
   c2.inc(); c2.inc(); System.out.pr
   c2.dec();           System.out.pr
}
```

Java

~C#

In questo caso, la **new** ha funzionato senza colpo ferire, perché è scattato il costruttore di default *generato automaticamente*

- Nel piccolo test di prima, **Counter2** non aveva costruttori
  - il main di test si è compilato *solo grazie al costruttore di default aggiunto automaticamente dal compilatore*
  - è grazie a ciò se la frase **new Counter2()** è stata accettata
  - MA: esattamente, cosa fa tale costruttore..?

UN PICCOLO MAIN DI TEST

```
public static void main(String[] args){

   Counter c1 = new Counter(10);                    // 1
   c1.inc(); c1.inc(); System.out.println(c1);      // 3

   Counter2 c2 = new Counter2(10);
   c2.inc(); c2.inc(); System.out.pr
   c2.dec();           System.out.pr
}
```

Java

~C#

NON COMPILA: non esiste in **Counter2** un costruttore a un argomento!

- Infatti, se avessimo tentato di passare un valore, il programma non si sarebbe compilato!
  - il compilatore avrebbe lamentato la mancanza del costruttore/1 in **Counter2**

DELEGA ALLA CLASSE BASE LA KEYWORD `super` (`base` in C#)

- La parola chiave **`super`** (Java, Scala, Kotlin) / **`base`** (C#) opera *in modo analogo* alla keyword **`this`**
  - nella forma **`super(...)`** [usata solo in Java; in C#, **`base(...)`**] invoca il costruttore opportuno (argomenti) della classe base
  - nella forma **`super.campo`** [in C#, **`base.campo`**] , consente di accedere al campo **`campo`** della classe base *(sempre che esso non sia privato)*
  - nella forma **`super.metodo()`** [in C#, **`base.metodo`**] invoca il metodo **`metodo()`** della classe base *(sempre che esso non sia privato)*

COMPLETAMENTO DEL `Counter2`

Il contatore con decremento completato (Java)

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

Java

Ex costruttore di default
Stavolta va specificato
perché il compilatore
non lo aggiunge più

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

- Questa versione di **`Counter2`** ha i giusti costruttori:
  - introduce esplicitamente il costruttore/1
  - specifica anche il costruttore/0 perché, in presenza di costruttori espliciti, il compilatore *non* aggiunge più nulla di sua iniziativa

UN PICCOLO MAIN DI TEST

```
public static void main(String[] args){
   Counter c1 = new Counter(10);                    // 1
   c1.inc(); c1.inc(); System.out.println(c1);      // 3

   Counter2 c2 = new Counter2(10);
   c2.inc(); c2.inc(); System.out.pri
   c2.dec();           System.out.println(c2);      // 2
}
```

Java

~C#

Ora finalmente compila tutto!

- Il costruttore di **`Counter2`** *rimpalla la costruzione su un apposito costruttore di* ***`Counter`***, che è il «vero proprietario» del campo dati **`val`** da inizializzare
  - in questo caso, il costruttore a un argomento di **`Counter2`** rimpalla la costruzione sul costruttore analogo di **`Counter`**

COMPLETAMENTO DEL `Counter2`

Il contatore con decremento completato (C#)

C#

```
public class Counter2 : Counter {
   public void dec() { val--; }
   public Counter2() : base() {}
   public Counter2(int v) : base(v) {}
}
```

Ex costruttore di default
Stavolta va specificato
perché il compilatore
non lo aggiunge più

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

Counter IN SCALA e KOTLIN

Scala

```
class Counter(protected var myValue : Int = 1){
   def reset() : Unit = { myValue = 0; }
   def inc() : Unit   = { myValue +=1; }
   def getValue() : Int = { return myValue; }
   def equals(x : Counter) : Boolean = {
                        return myValue==x.myValue; }
   override def toString() : String = {
                        return "Counter di valore " + myValue; }
}
```

Kotlin

```
open public class Counter(protected var myValue : Int = 1){
   publi                                                    }
   pub
   publ                                                 lue; }
   publ                                                    {
                                                        alue; }
   publi
                        return "Counter di valore " + myValue; }
}
```

In Kotlin, le classi destinate a essere *estese tramite ereditarietà* devono essere preventivamente etichettate **open**

Altrimenti, l'estensione sarà rigettata

Counter IN SCALA e KOTLIN

Scala

```
class Counter(protected var myValue : Int = 1){
   def reset() : Unit = { myValue = 0; }
   def inc() : Unit   = { myValue +=1; }
   def getValue() : Int = { return myValue; }
   def equals(x : Counter) : Boolean = {
                        return myValue==x.myValue; }
   overr                                          valore " + myValue; }
}
```

Se la proprietà pubblica si chiama **value**, il metodo *non* si può chiamare **getValue** Altrimenti, clash!

MOTIVO: sotto banco, alle proprietà pubbliche il compilatore associa già degli accessor generati automaticamente, che si chiamano proprio **getXX** (li usa la JVM)

Kotlin

```
open pu                                          Value : Int = 1){
   publ                                          ; }
   public fun inc() : Unit   { myValue++; }
   public fun getValue() : Int { return myValue; }
   public fun equals(x : Counter) : Boolean {
                        return myValue==x.myValue; }
   public override fun toString() : String {
                        return "Counter di valore " + myValue; }
}
```

Se la proprietà pubblica si chiama **value**, il metodo *non* si può chiamare **getValue** Altrimenti, clash!

Counter2 IN SCALA e KOTLIN

```
class Counter2(value : Int = 1) extends Counter(value) {
   // tutto va come se si potesse scrivere:
   //  def this() = super()
   //  def this(v : Int) = super(v)
   // ma ciò è illecito: si opera via parametri di classe
   def dec() : Unit = { myValue -= 1; }
}
```

Non si chiama **value** per evitare name clash nell'accessor **getValue**

Scala

```
public class Counter2(value : Int = 1) : Counter(value) {
   // tutto va come se fossero definiti i costruttori
   // seguenti (infatti, provandoci si ha name clash):
   //  public this() : super() {}
   //  public this(v : Int) : super(v) {}
   public fun dec() : Unit { myValue--; }
}
```

Non si chiama **value** per evitare name clash nell'accessor **getValue**

Kotlin

IL CONTATORE COLORATO: COMPLETAMENTO

Il costruttore del contatore colorato deve

- inizializzare direttamente il colore (definito in questa classe)
- rimpallare al «piano di sopra» l'inizializzazione del valore

Java

```
public import java.awt.Color;
public class ColoredCounter extends Counter {
 private Color color;
 public ColoredCounter(int v){
   super(v); this.color = Color.black; }
 public ColoredCounter(int v, Color color){
   super(v); this.color = color; }
 public Color getColor() { return color; }
...
}
```

IL CONTATORE COLORATO: COMPLETAMENTO

Anche **`toString`** ed **`equals`** vanno adattate:

Java

```
public boolean equals(ColoredCounter that) {
  return super.equals(that) && color==that.color; }

public String toString(){
  return super.toString() + " e colore " + color; }
```

- entrambe definiscono il proprio comportamento *«basandosi» sulla* ***`equals`*** */* ***`toString`*** *ereditate da* ***`Counter`***
- la keyword **`super`** invoca un metodo ereditato dalla classe base (è indispensabile per distinguere **`toString`** di **`ColoredCounter`** dalla toString ereditata da **`Counter`**)

UN PICCOLO MAIN DI TEST

Java

~C#

```
public static void main(String[] args){

   Counter c1 = new Counter(10);                    // 1
   c1.inc(); c1.inc(); System.out.println(c1);  // 3

   ColoredCounter cc2 = new ColoredCounter(10); // nero
   System.out.println(cc2);

   ColoredCounter cc3 = new ColoredCounter(14, Color.red);
   cc3.inc(); System.out.println(cc3);

   ColoredCounter cc4 = new ColoredCounter(10, Color.red);
   System.out.println(cc2.equals(cc4));

}
```

Stesso valore,
ma colore diverso

```
Counter di valore 3
Counter di valore 10 e colore java.awt.Color[r=0,g=0,b=0]
Counter di valore 15 e colore java.awt.Color[r=255,g=0,b=0]
false
```

RIASSUNTO

- Tutti i costruttori della classe derivata devono appoggiarsi a un qualche costruttore della classe base, perché
  - 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 può accedere
  - è inutile duplicare nella sottoclasse tutto il codice necessario per inizializzare i dati ereditati, già scritto e *collaudato*
  - si garantisce coerenza nella evoluzione del software: eventuali modifiche nella struttura della classe base (e quindi nei suoi costruttori) *si riverberano automaticamente nelle classi "figlie"*
- PRINCIPIO: *"ognuno deve costruire ciò che gli compete"*

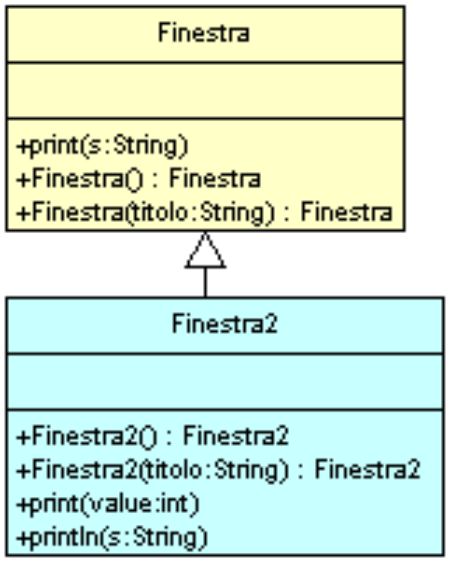
Progettazione incrementale
Estensione del componente **`Finestra`**

IL COMPONENTE `Finestra`: LIMITI

- Riconsideriamo la classe **`Finestra`** (senza factory) Java
  - offre un metodo **`print`** di base, che non va a capo
  - stampa solo stringhe (non interi, né **`float`**, etc)
- Potrebbe farci comodo una sua *versione migliorata*, capace anche di
  - stampare andando a capo (metodo **`println`**)
  - stampare anche **`int`**

OLTRE `Finestra`

- Non abbiamo il codice di **`Finestra`**, quindi *non possiamo modificare il sorgente*
  - d'altronde, non vogliamo neanche farlo!
- Possiamo però sfruttare l'ereditarietà per definire una nuova classe **`Finestra2`** che estenda **`Finestra:`**
  - definendo un *nuovo metodo* ***`println`*** che vada a capo dopo la stampa
  - definendo un *nuovo metodo* ***`print(int)`*** che stampi direttamente un intero, senza doverlo prima convertire in stringa.



LA CLASSE `Finestra2`

Java

```
public class Finestra2 extends Finestra {
 public Finestra2() { super(); }
 public Finestra2(String titolo) { super(titolo); }
 public void println(String txt){ print(txt + "\n"); }
 public void print(int x){ print("" + x); }
}
```

NOTE

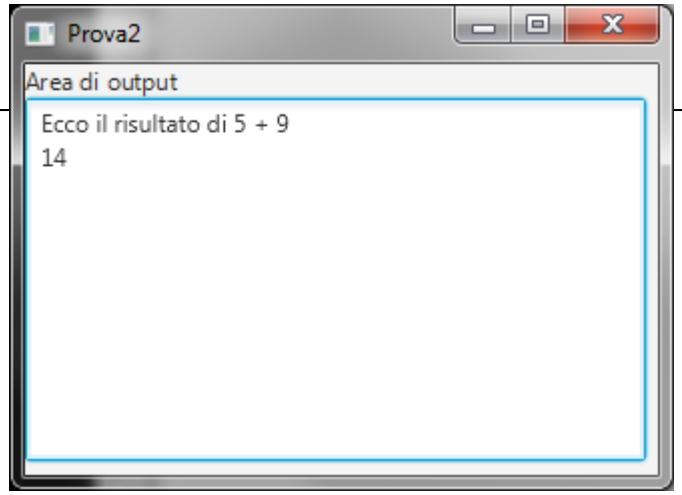
- attenzione ai giusti nomi di package (e relativi **`import`**)
- il costruttore di **`Finestra2`** può invocare qualsiasi costruttore di **`Finestra`** ritenga opportuno e/o cambiare la frase da visualizzare
- i due nuovi metodi di **`Finestra2`** si appoggiano al metodo **`print`** già esistente in **`Finestra`**

UN ESEMPIO D'USO

Aggiungere le opportune **`import`**

Java

```
public class Prova2 {
 public static void main(String args[]){
   Finestra2 f = new Finestra2("Prova2");
   f.println("Ecco il risultato di 5 + 9");
   f.print(5+9); // ora il parametro è un int
 }
}
```



UN PRIMO BILANCIO

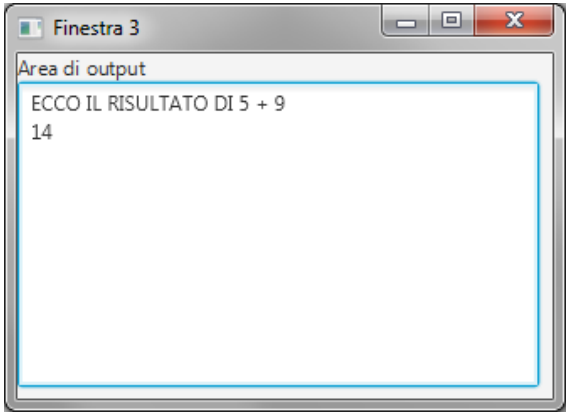
- Non è necessario disporre del codice sorgente per poter specializzare il componente
  - basta il file `.class/.jar` con adeguata documentazione
  - è possibile specializzare *in un nostro package* un componente originariamente definito *in un altro package*
- È possibile riusare un componente *di cui non si conosce il funzionamento interno* e che *non si sarebbe stati in grado di costruire da soli*
  - l'ereditarietà è un mezzo estremamente potente per riusare e adattare componenti fatti da altri
  - si toccano con mano i vantaggi dell'incapsulamento

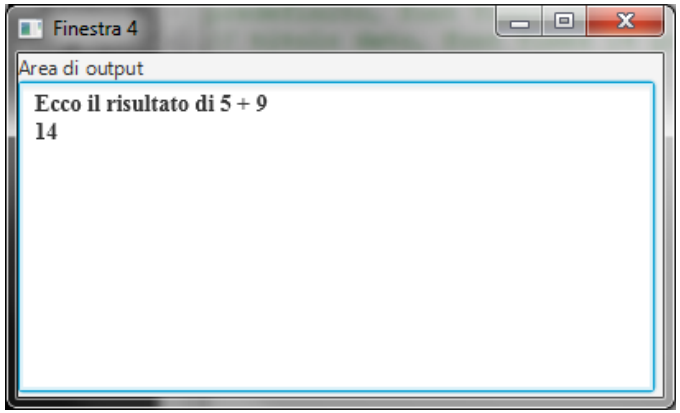
L'APPETITO VIEN MANGIANDO…

Oltre **`Finestra2`**

- **`Finestra3`**
  - estende **`Finestra2`** ridefinendo **`print`** in modo che *stampi tutto in maiuscolo*
- **`Finestra4`**
  - estende **`Finestra2`** aggiungendo un costruttore che permetta di *specificare il font da usare*

Quanto in là ci si può spingere?
(soprattutto: è saggio farlo..?)





Progettazione incrementale: Estensione o Ridefinizione?

SOLO ESTENSIONE…?

- Finora, in entrambi gli esempi visti, la classe derivata ha solo aggiunto nuovi metodi
  - in **`Counter2`**, abbiamo aggiunto `dec()`
  - in **`Finestra2`**, `print(int)` e `println(String)`
- Perciò, il comportamento della classe derivata è risultato essere una *estensione conservativa* della classe base
  - in **`Counter2`**,
    `inc`, `reset` e `getValue` continuano a funzionare come in `Counter`
  - in **`Finestra2`**
    `print(String)` funziona esattamente come in `Finestra`

…O ANCHE MODIFICA?

- L'ereditarietà permette anche di *modificare il funzionamento* dei metodi ereditati, per adattarli alla nuova situazione
  - OCCHIO: non significa che si possa fare qualunque cosa!
- È responsabilità del progettista assicurare che il nuovo comportamento sia sempre *un'estensione conservativa* di quello della classe base, *senza assurdi sovvertimenti*
  - cosa accadrebbe se **`Counter2`** alterasse **`inc`** e lo ridefinisse, ad esempio incrementando di 7…o addirittura *facendo tutt'altro ?*
  - cosa accadrebbe se **`Finestra2`** ridefinisse **`print(String)`** in modo [del tutto] diverso ??
- RICORDA: non tutto il contratto è espresso dalla signature, m questo non significa poterlo ignorare!

IL `Counter` IMPAZZITO

Un figlio impazzito di **`Counter`**:

```
public class CrazyCounter2 extends Counter {
   public void dec() { val--; }
   public void inc() { val = val /10; }
   public CrazyCounter2() { super(); }
   public CrazyCounter2 v) { super(v);
}
```

Java

~C#

ASSURDO: ridefinisce il metodo **`inc`** *cambiandone completamente la semantica!*

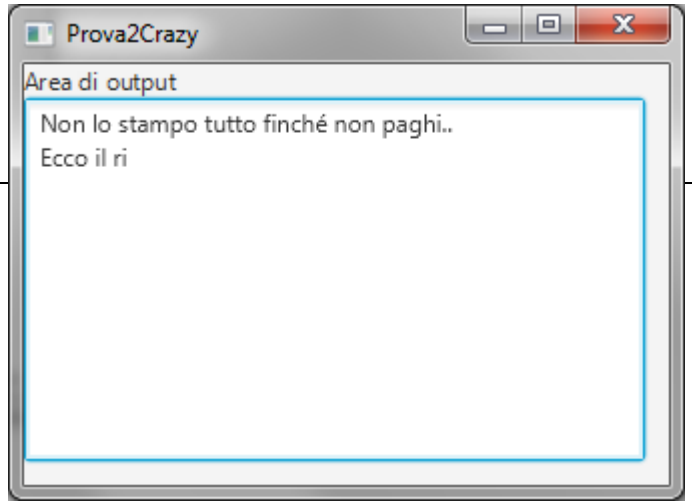
Non è un'estensione conservativa: dà luogo a un assurdo logico e comportamentale

LA Finestra IMPAZZITA

Anche questa figlia di **Finestra** non scherza:

Java

```
public class Finestra2Crazy extends Finestra2 {
  ...
  @Override
  public void print(String txt){
    super.print(
       "Non lo stampo tutto finché non paghi..\n" +
       txt.substring(0,10) );
  }
}
```



THE LESSON LEARNED

- Come tutti i meccanismi, anche l'ereditarietà *non può garantire in sé che il progettista non scriva follie*
- L'interfaccia di una classe *riassume solo una parte* del contratto d'uso
  - specifica *i nomi* delle operazioni e *come invocarle*..
  - .. ma NON specifica i *vincoli semantici* sottintesi nel comportamento
    - che **`inc`** rappresenti una qualche forma di "incremento"
    - che **`reset`** rappresenti una qualche forma di "ripristino"…
- I modelli di *software engineering* evoluti oggi mirano a *catturare* ed *esprimere* anche questi aspetti

METODI e CLASSI FINALI

- Per garantire che un metodo mantenga la sua semantica *anche in futuro,* si può *bloccare l'ereditarietà* impedendo che quel metodo possa mai più essere ridefinito
- A tal fine:
  - in Java e Scala lo si etichetta **`final`**
  - in C#, lo si etichetta **`sealed`**
  - in Kotlin non lo si etichetta perché questo è il default: per permettere l'override, il metodo va etichetttato esplicitamente **`open`**
- Si può portare questo approccio agli estremi etichettando ***`final`*** / ***`sealed`*** *un'intera classe, impedendone così completamente l'estensione futura* da parte di chiunque
  - in Kotlin questo è il default, altrimenti occorre l'etichetta **`open`**

METODI E CLASSI FINALI: ESEMPI

Perché il metodo **`inc`** non possa mai più essere ridefinito:

Java ~C# ~Scala

```
public class Counter {
   protected int val;
   ...
   public final void inc() { val++; }
}
```

C#: **`sealed`**

Perché l'intera classe non possa mai più essere estesa:

Java ~C# ~Scala

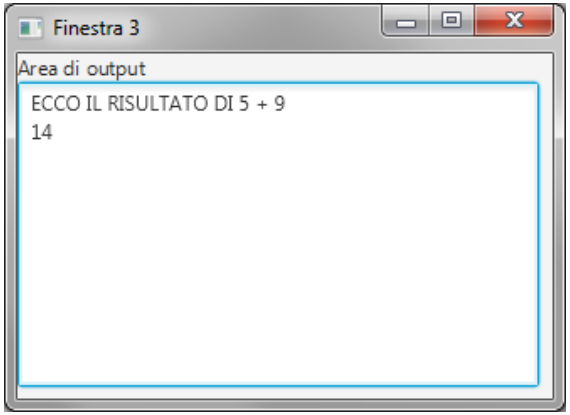
```
public final class LastCounter extends Counter {
        ...
}
```

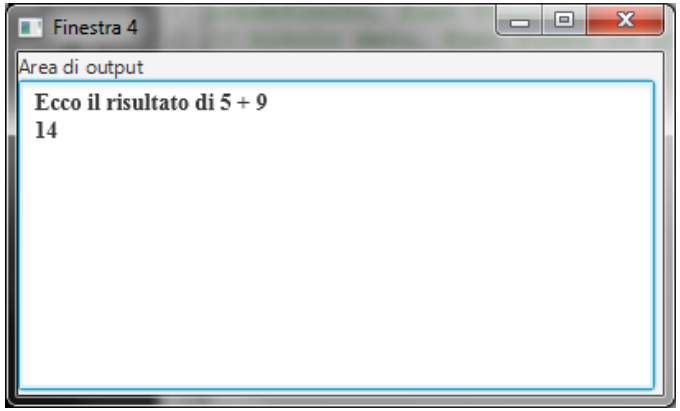
C#: **`sealed`**

TORNANDO ALLE FINESTRE..

Oltre **`Finestra2`**

- **`Finestra3`**
  - estende **`Finestra2`** ridefinendo **`print`** in modo che *stampi tutto in maiuscolo*
- **`Finestra4`**
  - estende **`Finestra2`** aggiungendo un costruttore che permetta di *specificare il font da usare*
  - …ma come fa, se **`Finestra2`** non dà accesso ai suoi elementi interni..?





NUOVE FINESTRE

Java

```
public class Finestra3 extends Finestra2 {
   …
 @Override
 public void print(String txt){
   super.print(txt.toUpperCase());
   // OCCHIO: senza super sarebbe ricorsione infinita!
 }
}
```

BUONA ridefinizione: usa solo *metodi pubblici* della classe base, non sfrutta "trucchi" né si basa su ipotesi non garantite.

Java

```
public class Finestra4 extends Finestra2 {
   …
 public Finestra4(String titolo, Font font) {
   super(titolo);
   ta.setFont(font);
 }
   …
}
```

PESSIMA ridefinizione: sfrutta una conoscenza implicita dei meccanismi interni della classe base → *codice fragile e mal progettato, che si rompe alla prima modifica* di Finestra2.

`Finestra2` CON FACTORY

- La versione originale di **`Finestra`** consentiva la creazione diretta di oggetti, ma non garantiva il rispetto del vincolo tecnologico che ce ne fosse una sola.
- La *versione affinata* di **`Finestra`** usava una *factory*: di conseguenza, i costruttori *non erano pubblici e la costruzione avveniva per via indiretta*

```
Finestra f = Finestra.of("Prova1");
```

Si riesce a estendere questa?

Java

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
static Finestra	of(java.lang.String titolo)	Fabbrica delle finestre: crea e restituisce una Finestra con il titolo dato, ma solo se essa non è già esistente!
void	print(java.lang.String txt)	Stampa la stringa data nell'area di output della finestra.

`Finestra2` CON FACTORY

- Se i costruttori di **`Finestra`** sono *privati,* non c'è niente da fare: il componente non ammette estensioni
- Se invece sono *protetti*, l'estensione è possibile
  - i costruttori di **`Finestra2`** possono agganciarsi ai precedenti
  - **`Finestra2`** definirà poi *il suo nuovo metodo factory* (eventualmente, anche con un nome diverso dal precedente)

USANDO LA VERSIONE CON FACTORY di **`Finestra`**:

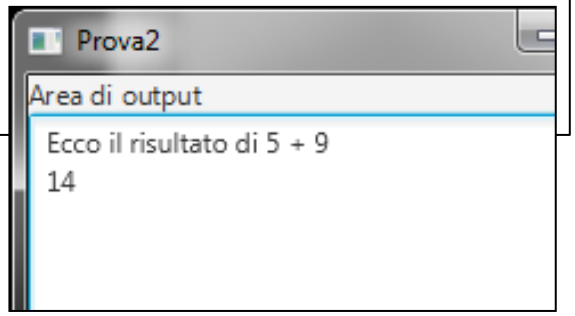
Java

```
public class Finestra2 extends Finestra {
 protected Finestra2(String titolo) { super(titolo); }
 public static Finestra2 of(String titolo) {
   return new Finestra2(titolo);
 }
}
```

L'ESEMPIO CON FACTORY

Java

```
public class Prova2 {
 public static void main(String args[]){
   Finestra2 f = Finestra2.of("Prova2");
   f.println("Ecco il risultato di 5 + 9");
   f.print(5+9); // ora il parametro è un int
 }
}
```



In realtà, *non è questo il modo migliore per fare grafica*

- è un approccio «old style» (infatti, è Swing..)
- in JavaFX si fa in modo che *l'intera app sia grafica*, non che un'applicazione classica "apra una finestra"

UN APPROCCIO ALTERNATIVO

- Per questo, Zio Enrico vi offre una alternativa migliore, coerente con JavaFX: **`GraphicApplication`**
  - in tale approccio l'applicazione non crea alcuna finestra grafica, *perché è lei stessa già un'applicazione grafica!*
  - NB: per questa app, è necessario installare e configurare JavaFX
- La vostra classe dovrà *estendere* ***`GraphicApplication`***
  - non c'è più il classico main (statico, void, etc.)
  - il vostro algoritmo dovrà essere incapsulato in un *nuovo metodo* ereditato da **`GraphicApplication`**, che chiameremo.. **`main`** ! ☺
  - Ma attenzione: *non ha niente a che spartire* col «classico main"!
  - *non governiamo più noi il flusso di controllo:* lo fa l'applicazione
  - non si ereditano solo «un po' di comode funzioni": si eredita *un'impostazione, una idea, una filosofia d'uso*

UN APPROCCIO ALTERNATIVO

Tre metodi fondamentali: **`main`**, **`setTitolo`**, **`print`**

- OCCHIO: nonostante il nome, **`main`** non è il classico main statico (non a caso, è diversa anche la signature)

ed.utils

Class GraphicApplication

Java + JavaFX

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
void	**`main`**() Contiene la logica dell'applicazione.
void	**`print`**(java.lang.String txt) Mostra la stringa data nell'area di output della finestra.
void	**`setTitolo`**(java.lang.String titolo) Imposta il titolo della finestra.

Incorpora l'algoritmo desiderato

L' ESEMPIO RIFORMULATO

È già essa stessa un'applicazione grafica

```
public class ProvaGA extends GraphicApplication {
  @Override
  public void main(){
    setTitolo("MyApp");
    print("Ecco il risultato di 5 + 9");
    print(""+(5+9));
  }
}
```

Java + JavaFX

Incorpora l'algoritmo

NON è il main statico a cui siamo abituati, che *non c'è più.* Ora *non governiamo più noi* il flusso di controllo.

I metodi non sono più invocati su una finestra creata da noi ma (implicitamente) su **`this`**, perché *la finestra "è" l'applicazione.*

Più avanti nel corso approfondiremo questo argomento.

Un esempio conclusivo: Persone e Studenti

PERSONE & STUDENTI (1/2)

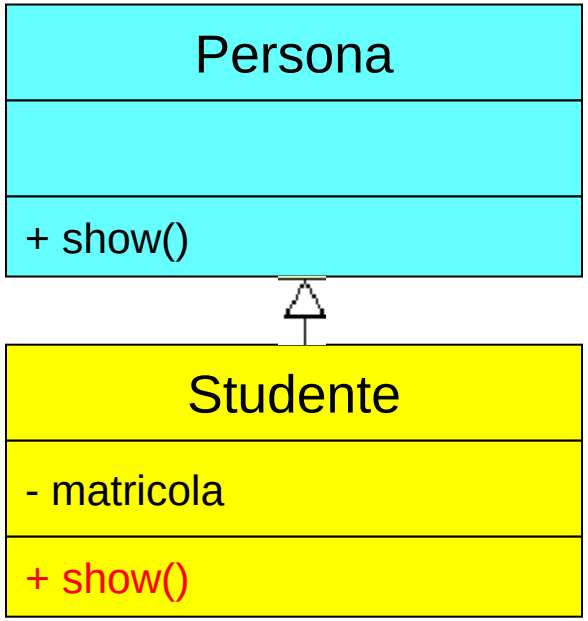
Supponiamo di aver modellato l'entità *Persona*

- dotata, fra le altre cose, in un metodo **`show()`**

e di voler ora modellare l'entità *Studente*

IPOTESI:
gli studenti sono persone con la particolarità di avere anche un *numero di matricola*

CONSEGUENZA:
occorre ridefinire il metodo **`show`** di Studente in modo che visualizzi *anche* tale dato, *oltre a tutti gli altri che già erano mostrati* in Persona



PERSONE & STUDENTI (1/2)

Supponiamo di aver modellato l'entità *Persona*

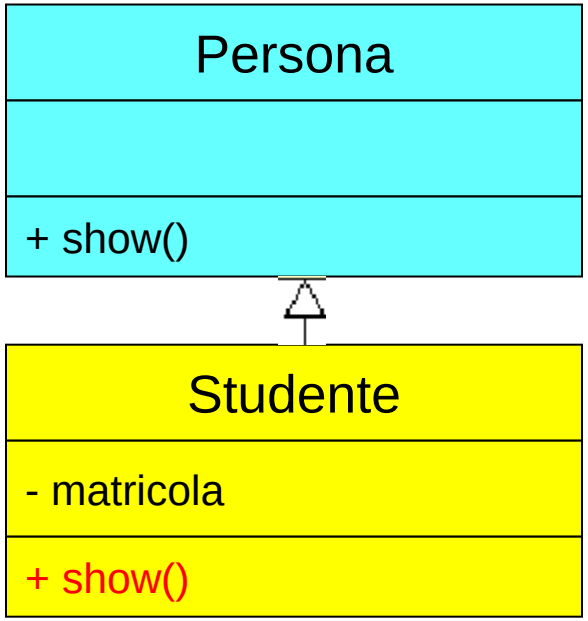
- dotata, fra le altre cose, in un metodo **`show()`**

e di voler ora modellare l'entità *Studente*

IPOTESI:
gli studenti sono persone con la particolarità di avere anche un *numero di matricola*

CONSEGUENZA:
occorr
in mo
tutti g

Ridefinizione conservativa: **`show()`** continua a fare ciò che faceva prima *e in più mostra anche la matricola*



IMPLEMENTAZIONE in Java e C#

Java ~C#

```
public class Persona {
   protected String nome;
   public void show() {
        System.out.println(nome);
   }
}
```

Può essere utile l'annotazione `@Override` per sottolineare che si tratta di una *ridefinizione intenzionale*, voluta.

Java ~C#

```
public class Studente extends Perso
   private long matricola;
   public void show() { // RIDEFINITO
        System.out.println(nome + " " + matricola);
   }
}
```

Ridefinizione conservativa: `show()` fa ciò che faceva prima *e in più visualizza anche la matricola*

IMPLEMENTAZIONE in Java e C#

Così, però, la **`show`** di **`Studente`** *sostituisce totalmente* la precedente, ereditata da **`Persona`**

- facile e naturale, perché la **`show`** originale faceva una cosa sola
- ma… *se la **`show`** originale fosse stata complessa e preziosa?*

Sovrascrivere totalmente è spesso uno spreco

- spesso è preferibile *recuperare il comportamento precedente e riusarlo*

Java

~C#

```
public class Studente extends Persona {
   private long matricola
   public void show() {
        super.show();
        System.out.println(matricola);
   }
}
```

Recupera e riusa il comportamento precedente, *aggiungendo le peculiarità di Studente.*

IMPLEMENTAZIONE in Scala

Implementiamo qui una versione leggermente più ampia di Persona, con anche l'età (pessima idea…)

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

Scala

Non servono costruttori né accessor: sono introdotti automaticamente dal compilatore sulla base dei parametri di classe

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

Keywords **extends** e **super** usate come in Java

Scala

IMPLEMENTAZIONE in Kotlin

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

Kotlin

Come in Scala, anche qui costruttori e accessor sono introdotti automaticamente dal compilatore

In Kotlin, le classi destinate a essere *estese tramite ereditarietà* devono essere preventivamente etichettate **open** *e lo stesso vale per ogni metodo da ridefinire*

Altrimenti, l'estensione sarà rigettata

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

Keywords **:** e **super** usate come in C#

Kotlin

UN PICCOLO MAIN

Scala

```
object PersoneStudenti {
 def main(args: Array[String]) : Unit = {
    var p = new Persona("John");
    var s = new Studente("Tom");
    p.show();  // stampa i dati base
    s.show();  // stampa anche l
 }
}
```

Java

~C#

```
public class PersoneStudenti {
 public static void main(String args[]){
    Persona  p = new Persona("John");
    Studente s = new Studente("Tom");
    p.show();  // stampa i dati base
    s.show();  // stampa anche la matricola
 }
}
```

Kotlin

```
fun main(args: Array<String>) : Unit {
    var p = Persona("John");
    var s = Studente("Tom");
    p.show();  // stampa i dati base
    s.show();  // stampa anche la matricola
}
```