



Alma Mater Studiorum-Università di Bologna  
Scuola di Ingegneria

---

# Sistemi software a oggetti

*Corso di Laurea in Ingegneria Informatica*  
Anno accademico 2021/2022

**Prof. ENRICO DENTI**

*Dipartimento di Informatica – Scienza e Ingegneria (DISI)*

# SISTEMI A OGGETTI

- Un *sistema a oggetti* è un *sistema* fatto di (classi e) oggetti che interagiscono fra loro
- Ma cosa vuol dire sistema "*fatto di*" oggetti?
  - Come sono fatti gli oggetti “complessi”?
  - Come interagiscono questi oggetti fra loro?

Progettare un *sistema che risolva un problema* richiede di:

1. ANALIZZARE IL PROBLEMA partendo dai *requisiti*
2. PIANIFICARE IL COLLAUDO e il lavoro
3. PROGETTARE UNA SOLUZIONE (non implementare!)
4. IMPLEMENTARE la *soluzione* scelta
5. COLLAUDARE *l'implementazione* realizzata.



# MODELLI DEL SISTEMA

---

- Per esprimere il risultato di queste fasi si ricorre a *modelli* del sistema, spesso espressi graficamente
  - In fase di **ANALISI** si modella il **Problema** dai *requisiti*  
Il risultato è un modello detto *Architettura Logica*,  
accompagnato dal *Piano del Collaudo*
  - In fase di **PROGETTO** si modella il **Sistema**  
Il risultato è un modello detto *Architettura del Sistema*
- È su questi modelli che si ragiona per valutare le caratteristiche e le proprietà di un sistema, *non sul codice!!*

# MODELLI E VISTE

- **L'Architettura Logica** si articola in viste, che fanno riferimento al problema in quanto tale:
  - **struttura** = macro-blocchi (derivanti dai requisiti)
  - **interazione** = relazioni dinamiche fra i macro-blocchi
  - **comportamento osservabile** dei singoli macro-blocchi
- Anche **l'Architettura del Sistema** si articola in viste, riferite però alla specifica soluzione che si sta ipotizzando:
  - **struttura** = macro-blocchi della specifica soluzione
  - **interazione** = specifica di dettaglio di come avviene
  - **comportamento di dettaglio** dei vari blocchi

# MODELLI E VISTE

- L'Architettura Logica si articola in *viste*, che fanno riferimento al problema

DIAGRAMMA DELLE CLASSI (generale)

- **struttura** = macro-blocchi (derivanti dai requisiti)
- **interazione** = relazioni tra macro-blocchi
- **comportamento** = Diagrammi degli stati

Diagrammi di sequenza

- Anche l'Architettura del Sistema si articola in *viste* riferite però alla specifica

DIAGRAMMA CLASSI DI DETTAGLIO  
(anche 10-100 volte più classi...)

- **struttura** = macro-blocchi della specifica soluzione
- **interazione** = specifica di dettaglio di come avviene
- **comportamento di dettaglio** dei vari blocchi



# MODELLARE UN SISTEMA

- Per modellare un sistema è essenziale **chiarire bene le relazioni fra le entità del dominio**
  - le descrizioni a parole possono essere fuorvianti...
  - .. e spesso ambigue.
- QUALCHE ESEMPIO:
  - una flotta **ha** un ammiraglio
  - una flotta **ha** delle navi
  - un esagono **ha** sei vertici
  - un libro **ha** delle pagine

Il verbo “avere” in queste frasi  
*non ha lo stesso significato!*  
Per modellare bene un sistema o un  
oggetto complesso occorre capire  
bene i diversi significati.

# RELAZIONI FRA OGGETTI

- Ragioniamo su queste frasi:
  - una flotta **ha** un ammiraglio
  - una flotta **ha** delle navi
  - un esagono **ha** sei vertici

- La flotta non è fatta di ammiragli...
- ...però, è in relazione con un (solo) ammiraglio che la comanda.

- La flotta è invece fatta di navi...
- che però **esistono singolarmente anche senza la flotta**
- Però, la flotta è tale solo finché c'è almeno una nave (forse..)

- **I vertici esistono solo finché esiste l'esagono**, perché il concetto stesso di "vertice" ( $\neq$  punto) prevede e sottintende una figura
- Inoltre, perché l'esagono esista, **tutti i vertici sono essenziali**

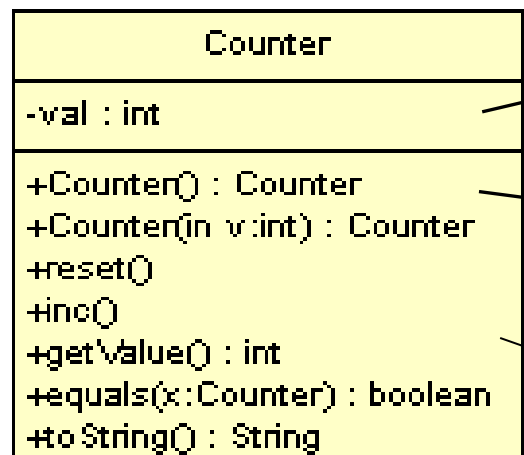
# UML

- *UML (Unified Modelling Language)* è un *linguaggio grafico* per esprimere modelli di un sistema a oggetti
  - ampiamente usato in tutte le fasi: analisi, progetto, implementazione
  - supportato da molti strumenti per analisi, progetto...
  - ... ma anche *reverse engineering* e *generazione semiautomatica di codice* (Java, C++, C#...)
- Definisce molti diagrammi, che catturano diverse *viste*
- Particolare importanza assume per noi il **diagramma della parte strutturale**, detto *Diagramma delle Classi*



# UML: CLASSI

- Una **classe** è rappresentata in UML mediante un disegno come il seguente:



**campi dati:**

“-” se privati, “+” se pubblici,  
“#” se visibilità intermedia

**costruttori e metodi:**

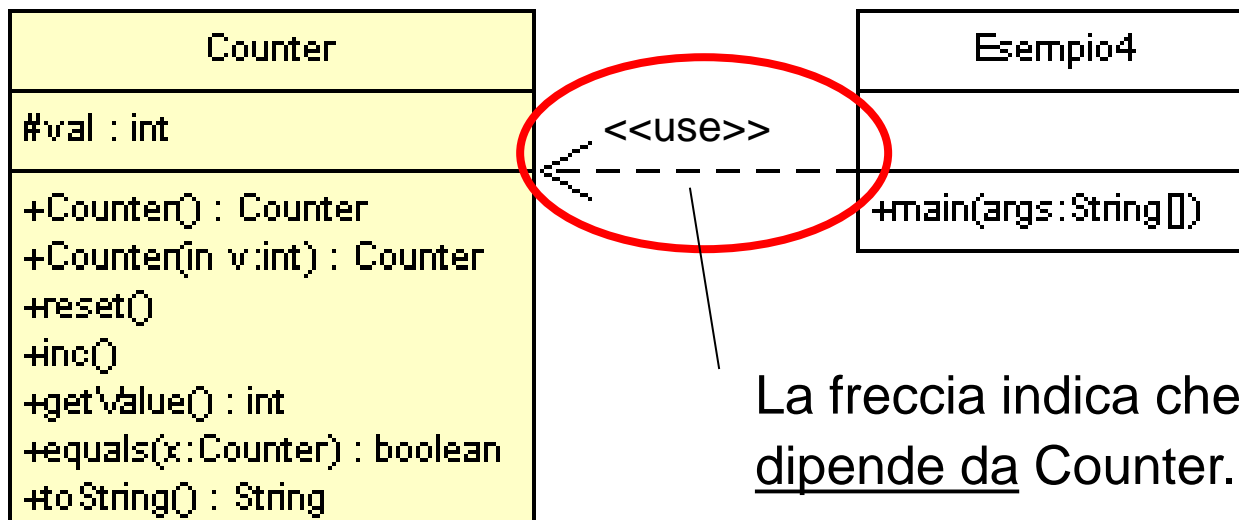
“-” se privati, “+” se pubblici,  
“#” se visibilità intermedia

Eventuali dati o metodi statici  
sono tipicamente sottolineati

- Esistono strumenti per
  - generare lo scheletro del codice dal modello (*forward engineering*)
  - ottenere il diagramma a partire dal codice (*reverse engineering*)

# UML: DIPENDENZA

- Quando una classe *dipende da* un'altra (caso tipico: *la usa*) si dice che vi è una *relazione di dipendenza*, indicata da una freccia tratteggiata.



La freccia indica che Esemplio4 dipende da Counter.

La specifica `<<use>>` (opzionale) dettaglia il tipo di dipendenza



# UML: ASSOCIAZIONI

---

- Quando una entità rappresentata da una classe è in relazione con una o più entità rappresentate da altre classi, si parla di *associazione* fra classi.
- Gli esempi visti prima:
  - una flotta *ha* un ammiraglio
  - una classe *ha* degli studenti
  - un libro *ha* delle pagine
  - un triangolo *ha* tre verticisono appunto *tipi diversi* di associazioni.



# ASSOCIAZIONI INTERESSANTI

---

- **Aggregazione**

- esprime l'idea che un oggetto «complesso» è un aggregato di altri oggetti (le «parti»), *che esistono e vivono indipendentemente dall'oggetto «contenitore»*
  - potrebbero anche essere parte di più «contenitori» allo stesso tempo
  - ESEMPIO: una flotta ha delle navi

- **Composizione**

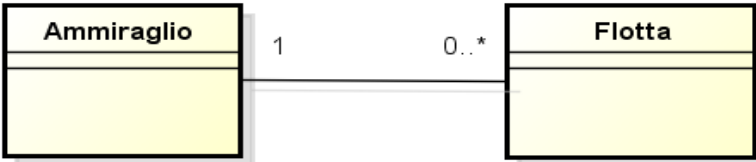



- esprime l'idea che le parti hanno un *tempo di vita intimamente legato a quello dell'oggetto composto*, tipicamente perché *sono tutte essenziali* alla sua esistenza
  - ESEMPIO: un triangolo ha tre vertici



# RIFLETTENDO SU ALCUNI ESEMPI...

- "una flotta **ha** un ammiraglio" → **associazione generica**
  - la flotta non è un aggregato di ammiragli
  - ma non è neppure composta da ammiragli
- "una flotta **ha** delle navi" → **aggregazione**
  - la flotta è fatta di navi, ma *le navi pre-esistono alla flotta e continuano a esistere anche se la flotta viene sciolta*
  - domani (o anche oggi..?) potrebbero far parte di un'altra flotta
- "un triangolo **ha** tre vertici" → **composizione**
  - un triangolo è composto di vertici: ognuno dei tre è *essenziale*
  - un «vertice» *esiste solo mentre esiste il triangolo*: è più di un «punto»
- "un libro **è fatto di** pagine" → **composizione**
  - la pagina non esiste (logicamente) separata dal libro
  - ogni pagina è essenziale perché il libro sia tale (mmhh...)

# RIFLETTENDO SU ALCUNI ESEMPI...

- "una flotta **ha** un ammiraglio" → 
  - la flotta non è un aggregato di ammiragli
  - ma non è neppure composta da ammiragli
- "una flotta **ha** delle navi" → 
  - la flotta è fatta di navi, ma *le navi possono continuare a esistere anche se la flotta non esiste più*
  - domani (o anche oggi..?) potrebbero far parte di un'altra flotta
- "un triangolo **ha** tre vertici" → 
  - un triangolo è composto di vertici: c'è un vertice per ogni triangolo
  - un «vertice» *esiste solo mentre esiste il triangolo*: è più di un «punto»
- "un libro **è fatto di** pagine" → 
  - la pagina non esiste (logicamente) se non c'è il libro
  - ogni pagina è essenziale perché il libro sia tale (mmmm...)



# DAL MODELLO AL CODICE

- La scelta fra **associazione**, **aggregazione**, **composizione** implica **vincoli ben precisi sul codice risultante**
  - al punto che lo scheletro può essere *generato automaticamente*
- In base al significato del tipo di associazione, si stabilisce:
  - se una classe contenga un **referimento** all'altra
  - la **molteplicità** di tale riferimento
    - singola = riferimento singolo
    - multipla = riferimento ad array o altra collection
  - se sia o meno necessaria una **copia degli argomenti** ricevuti
    - aggregazione = contenimento NON esclusivo → copia NON necessaria
    - composizione = contenimento esclusivo → copia opportuna



# DA MODELLO A CODICE: ASSOCIAZIONE GENERICA

- Associazione generica
  - ogni classe contiene un riferimento all'altra
  - se molteplicità  $>1$ , il riferimento sarà a un array (o altra collection)



```
public class Ammiraglio {  
    private Flotta[] flotte ;  
}
```

C#

Java

```
public class Flotta {  
    private Ammiraglio amm;  
}
```

C#

Java

Scala

Kotlin

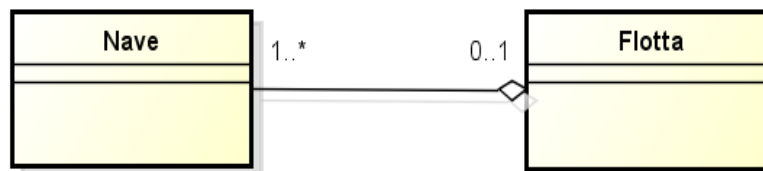
minime differenze



# DA MODELLO A CODICE: AGGREGAZIONE

- Aggregazione

- la classe "aggregante" contiene un riferimento a un array (o collection) dell'altra
- il tempo di vita di una nave è indipendente da quello della flotta (infatti, una nave può stare in più flotte): è un contenimento NON esclusivo  
→ non è necessario fare una copia privata nel costruttore



```
public class Flotta {  
    private Nave[] navi;  
    public Flotta(Nave[] navi) {  
        this.navi = navi;  
    }  
}
```

C#

Java

# DA MODELLO A CODICE: COMPOSIZIONE

- Composizione

- la classe "aggregante" contiene anche qui un riferimento a un array (o collection) dell'altra
- MA il tempo di vita di un vertice è dipendente da quello del triangolo  
→ un contenimento esclusivo → **opportuna (se gestibile) una copia**



```

public class Triangolo {
    private Vertice[] vertici;
    public Triangolo(Vertice[] vertici){
        this.vertici = Arrays.copyOf(vertici, vertici.length);
    }
}
  
```

A meno che l'argomento non sia *immodificabile..!*

Occhio alla copia con strutture di grandi dimensioni...!!

Java

~C#



# DA MODELLO A CODICE: DIPENDENZA (in generale)

- Una classe A **dipende** da una classe B (**A <<uses>> B**) quando A ha bisogno della collaborazione di B per funzionalità che non è in grado di effettuare autonomamente

- CASO 1: un'operazione della classe A **richiede come argomento** una istanza della classe B

```
void fun1(B b) { ... usa b ... }
```

Java

C#

- CASO 2: un'operazione della classe A **restituisce un oggetto** di tipo B

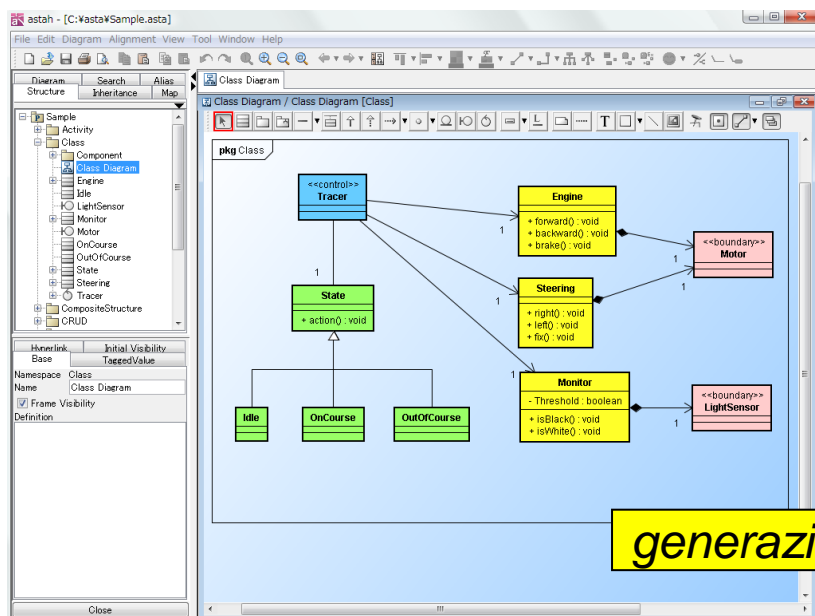
```
B fun2(...) { B b; ... return b; }
```

- CASO 3: un'operazione della classe A **utilizza una specifica istanza** della classe B, senza che però esista un'associazione tra A e B

```
void fun3(...) { B b = new B(...); ... usa b ... }
```

# PERCHÉ IL MODELLO È IMPORTANTE?

- Perché **evidenzia gli elementi strutturali** e le loro relazioni
- Perché **permette di ragionare a un alto livello di astrazione**, lasciando da parte aspetti implementativi e tecnologici
- Perché **è lì che si concentra l'attività creativa**: fatto il modello, appositi *strumenti* possono generare il codice in Java, C#..



generazione Java

```
// Monitor.java
package Class;
public class Monitor {
    private boolean Threshold;
    private Tracer tracer;
    private LightSensor lightSensor;
    public void isBlack() {
        ...
    }
    public void isWhite() {
        ...
    }
}
```

Java

# Un caso di studio: l'orologio

# OROLOGIO: REQUISITI

---

1. Il requisito primo del sistema è "contare il tempo"
2. Per motivi culturali vogliamo che il tempo sia misurato in secondi (e loro multipli: minuti, ore), espressi tramite un *orario (corrente)*
3. Dev'essere possibile **interrogare il sistema** per conoscere in ogni istante l'orario (corrente)





# OROLOGIO: ANALISI DEL DOMINIO

---

- L'orario *evolve nel tempo* secondo precise *regole*, dettate dal nostro *retroterra culturale*:
  - ogni 60 secondi, si avanza di 1 minuto
  - ogni 60 minuti, si avanza di 1 ora
  - ogni 24 ore, si torna a 0 (*e se mai si cambia data..*)
- MA l'orologio non può percepire direttamente il tempo!  
Lo scorrere del tempo appartiene al mondo fisico
- Ogni orologio opera indirettamente, *contando gli "avvisi" di un ente esterno legato al mondo fisico*
  - oscillazioni di un pendolo
  - oscillazioni atomiche
  - circuiti oscillanti basati su quarzi..
  - ...



# ANALISI DEL PROBLEMA (1/4)

Un **Orologio** è una entità:

- **caratterizzata da un *orario***
  - in sé *immodificabile*, come una stringa
- **dotata di stato** (l'orario corrente) che evolve nel tempo
  - recuperabile tramite un apposito metodo: **getOrario**

CONTRATTO:

l'orologio può assumere che sia trascorsa un'unità di tempo *ogni volta che il mondo fisico chiama il metodo **tic***

Orologio
+ getOrario(): Orario + tic()





# ANALISI DEL PROBLEMA (2/4)

Come si costruisce un **Orologio** ?

- all'atto della costruzione, si dovrà poter specificare *l'orario iniziale* (esattamente come..? con quale granularità..?)

Quali e quanti costruttori?

- **un costruttore da un Orario dato**  
(ma quanto specifico? anche i secondi..?  
anche i millisecondi o nanosecondi.. ?)
- poi.. *un costruttore di default...*?  
(ma che orario dovrebbe impostare ..?)

Orologio
+ Orologio(Orario): Orologio + Orologio(): Orologio + getOrario(): Orario + tic()

?



# ANALISI DEL PROBLEMA (3/4)

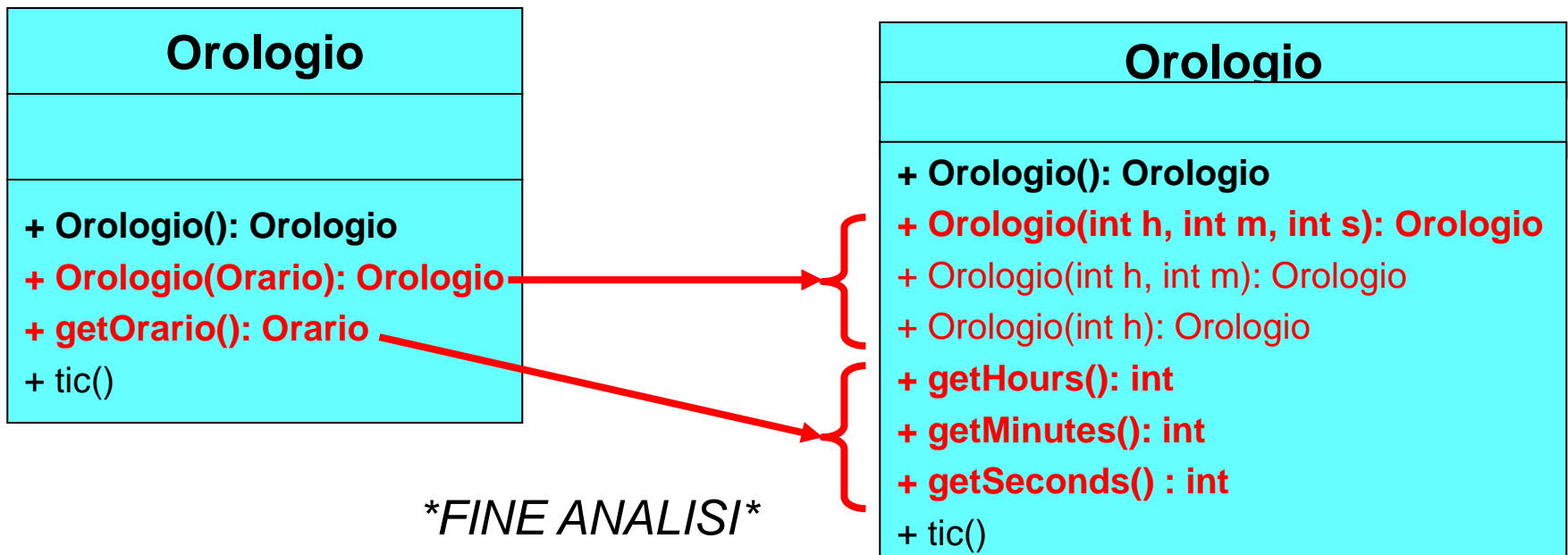
Come rappresentare il concetto di *Orario*?

- poiché il tempo nella nostra cultura è tipicamente espresso in *secondi*, *minuti* e *ore*, di norma visti come interi (altrimenti si ricorre al sottomultiplo), appare ragionevole **rappresentare Orario come terna di valori interi** per adeguarsi ai *requisiti culturali* del committente.
- **grado di libertà**: possiamo o fare una classe *Orario* che li incapsuli, o tenerci i tre interi gestiti singolarmente (pro / contro.. ?)
- Come esercizio, optiamo per la seconda scelta (discutibile)
  - allora, il costruttore con argomento *Orario* si raffina in un costruttore che prende per argomento una *terna di valori interi*
  - possono nascere altri costruttori per sfruttare questo fatto
  - il singolo metodo **getOrario** si raffina scindendosi in una *terna di metodi* – **getHours**, **getMinutes**, **getSeconds**

# ANALISI DEL PROBLEMA (4/4)

Raffinamento a seguito della scelta precedente

- da un solo costruttore con argomento *Orario*  
a tre costruttori con argomenti *interi – ore, minuti, secondi*
- corrispondente scissione del singolo metodo **getOrario**  
nella terna di metodi **getHours, getMinutes, getSeconds**

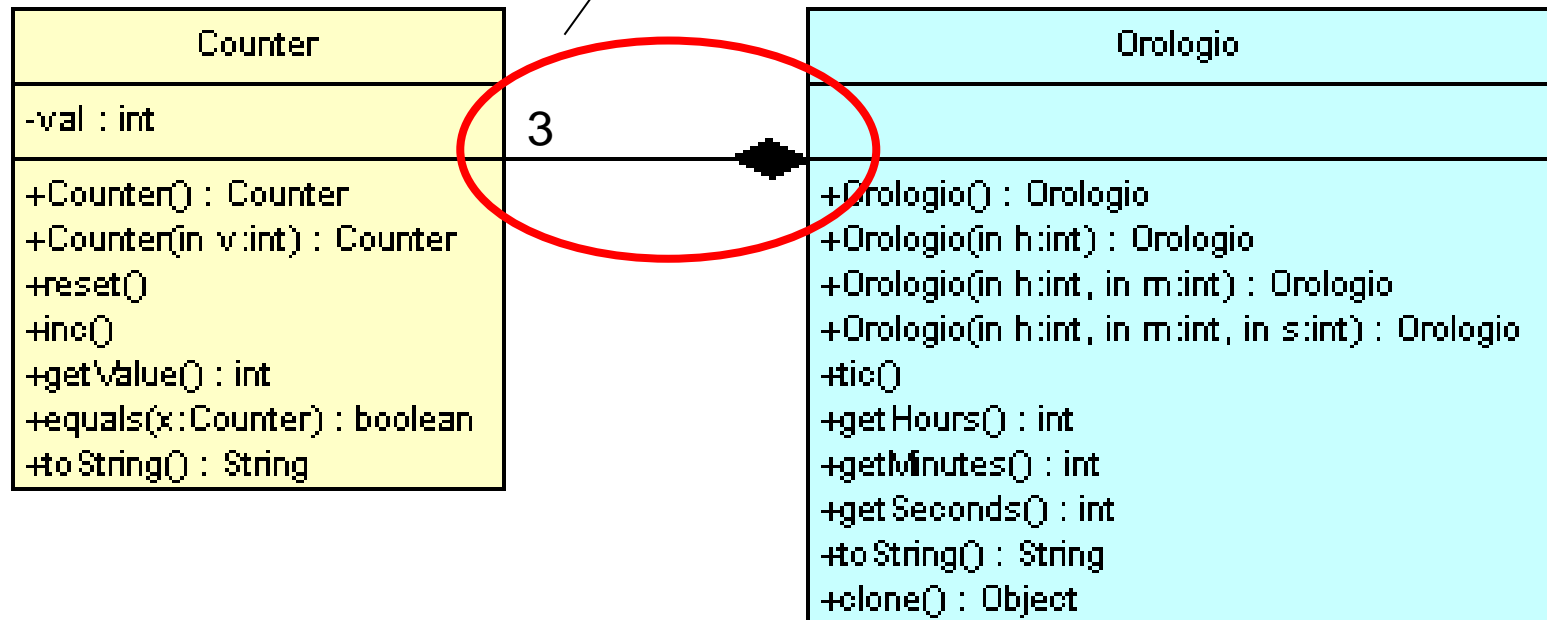


# OROLOGIO: PROGETTO

- Nella fase di analisi appena conclusa si è stabilito (mmhhh...) che l'orario sia rappresentato come *terna di valori interi*.
- Ora, in fase di progetto, fra le varie opzioni possibili, stabiliamo di usare a tal fine un *insieme di tre contatori* da far funzionare in modo *coordinato*.
  - La *terna di valori interi* messa in luce dall'analisi assume quindi la forma concreta di *tre contatori* in fase di progetto.
  - Si aggiunge un'opportuna *ridefinizione di toString* per produrre l'usuale rappresentazione di un orario nella forma **hh:mm:ss**
    - proprio col ":" ....?
    - ... o magari *tenendo conto delle culture locali..?*  
*Visualizzazione su 24 ore o su 12 ore? AM/PM?*

# OROLOGIO: IL MODELLO INTERNO

COMPOSIZIONE: un Orologio è fatto da esattamente tre Counter





# OROLOGIO: REALIZZAZIONE

La composizione si realizza **inserendo nell'oggetto composto riferimenti alle sue parti costitutive**

```
package clock;  
  
public class Orologio {  
    private Counter ore, minuti, secondi;  
    ...  
}
```

Java

C#

Scala

Kotlin

minime differenze

## SIGNIFICATO:

- Un oggetto `Orologio` utilizza / è fatto da *tre* `Counter`
- **Non può comunque accedere ai campi privati dei "suoi" `Counter`, perché essi sono oggetti *incapsulati***
- Può naturalmente usare i loro *metodi pubblici* (ed eventualmente quelli con visibilità di package)



# OROLOGIO: COSTRUZIONE

Poiché le parti interne sono di *esclusiva competenza* dell'oggetto composto, *è il suo costruttore primario* a dover costruire *esplicitamente* le parti interne:

```
package clock;
public class Orologio {
    private Counter ore, minuti, secondi;
    public Orologio(int h, int m, int s) {
        ore = new Counter(h);
        minuti = new Counter(m);
        secondi = new Counter(s);
    }
    ...
}
```

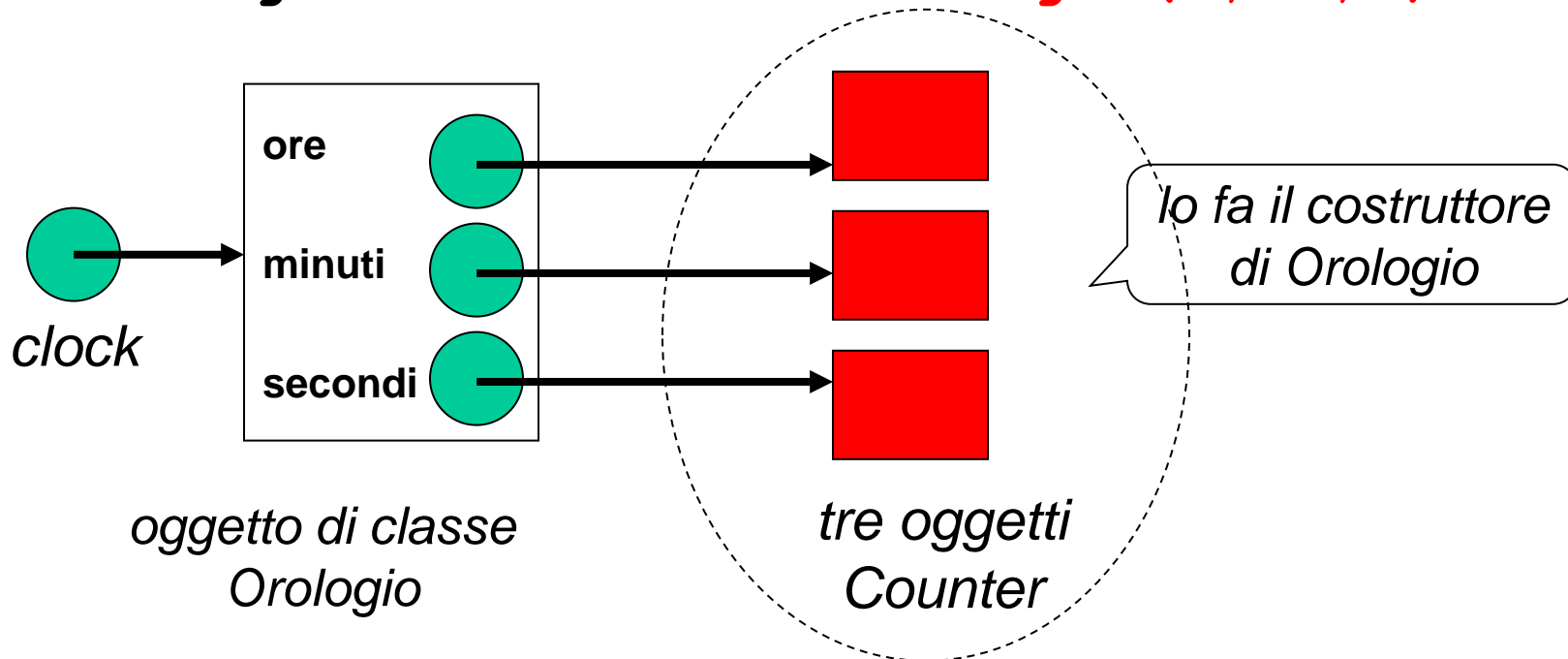
Java

C#

# SIMULAZIONE PASSO-PASSO

1. il cliente costruisce (il guscio del-) l'oggetto composto
2. il costruttore di quest'ultimo *s'incarica di costruire le parti interne*, completando l'opera.

`Orologio clock = new Orologio(7,30,0);`







# ALTRI COSTRUTTORI (1/3)

- Potrebbero essere utili altri *costruttori ausiliari*?
- Per capirlo dobbiamo analizzare i *possibili casi d'uso*
  1. Nel tuo quotidiano davvero regoli un orologio *al secondo*..?  
→ Ha senso un costruttore a *due soli argomenti* (ore, minuti)
  2. Nel tuo quotidiano ti capita di regolare l'orologio "alle 14"?  
→ Può avere senso un costruttore a *un solo argomento* (ore)
  3. Ha senso che un orologio si accenda *senza dover essere esplicitamente regolato*?  
Tutti i dispositivi hardware si accendono inizialmente su un orario "prefissato" → Può aver senso un *costruttore di default*
  4. *Questione di praticità*: potrebbe aver senso, invece di tre interi, un costruttore che accetti *una stringa della forma hh:mm:ss*?

## ALTRI COSTRUTTORI (2/3)

Java

```
public Orologio(int h, int m, int s) {  
    ore = new Counter(h);  
    minuti = new Counter(m);  
    secondi = new Counter(s);  
}
```

Costruttore primario a tre argomenti (caso generale)

```
public Orologio(int h, int m) {  
    this(h, m, 0);  
}
```

Costruttore ausiliario a due argomenti

```
public Orologio(int h) {  
    this(h, 0, 0);  
}
```

Costruttore ausiliario a un solo argomento

```
public Orologio() {  
    this(12, 0, 0);  
}
```

Costruttore di default ??  
Sicuri di volerlo? Ha veramente senso?  
E perché proprio le 12?



# ALTRI COSTRUTTORI (3/3)

```
public Orologio(String hhmmss) {  
    // ipotesi: campi separati da ':'  
    int separaMS = hhmmss.lastIndexOf(':');  
    int separaHM = hhmmss.indexOf(':');  
    String hh = hhmmss.substring(0, separaHM);  
    String mm = hhmmss.substring(separaHM+1, separaMS);  
    String ss = hhmmss.substring(separaMS);  
    int h = Integer.parseInt(hh);  
    int m = Integer.parseInt(mm);  
    int s = Integer.parseInt(ss);  
    // ora è facile proseguire ... ☺  
    ...  
}
```

Java



# UN PROBLEMA DI CONSISTENZA

---

## Riflessione

- Tutti i costruttori presuppongono che il cliente passi loro degli argomenti *sensati*
- *MA.. che succede se il cliente specifica un orario assurdo o inesistente, come le 25:71 ?!?*
  - Nel caso del costruttore da argomento stringa: *che succede se il separatore è diverso da ':' , o ci sono caratteri non numerici ?*

## Sono le prime avvisaglie di un grosso problema

- *la costruzione è un automatismo che non si può fermare*
- non servirebbe a niente aggiungere dei controlli: l'oggetto verrebbe costruito *comunque* e sarebbe *inconsistente!*



# UN PROBLEMA DI CONSISTENZA

## Riflessione

- Tutti i costruttori presuppongono che il cliente passi loro degli argomenti *sensati*
- *MA.. che succede se il cliente specifica un orario assurdo o inesistente, come le 25:71 ?!?*
  - Nel caso del costruttore da argomento stringa: *che succede se il separatore è diverso da ':' , o ci sono caratteri non numerici ?*

Risolveremo introducendo il concetto di **ECCEZIONE**.  
Per ora confidiamo che il cliente non faccia scherzi...



# OROLOGIO: FUNZIONAMENTO

- Solo i metodi di `Orologio` possono accedere ai contatori interni che costituiscono ogni orologio
- Il metodo `tic` fa avanzare l'orario di una *quantità fissa* di tempo *stabilita da contratto* – ad esempio, 1 secondo:

```
public void tic() {  
    secondi.inc();  
    if (secondi.getValue()==60) {  
        secondi.reset(); minuti.inc();  
    }  
    // eccetera.. completare!  
}
```

Java

C#

- Purtroppo, tale contratto *non compare nella signature dei metodi*, né può essere *enforced*: bisogna gestirlo a parte



# OROLOGIO: toString

Il metodo `toString` deve produrre una stringa che rappresenti l'orario *secondo la nostra cultura* – ad es. `hh:mm:ss`

```
public String toString() {  
    // implementazione naif da migliorare  
    return "" + ore.getValue() + ":"  
            + minuti.getValue() + ":"  
            + secondi.getValue();  
}
```

Java

C#

È una versione molto naif:

- Funziona bene per orari come 12:30:22 ☺...
- ...ma decisamente male per orari in cui *minuti* e/o *secondi* siano espressi su una cifra sola (es. 12:5:4 ☹)



# OROLOGIO: UN POSSIBILE `main`

- Collaudare ogni metodo è facile, ma per "far girare" questo componente bisogna poter chiamare `tic` ogni secondo, come da contratto
- **MA noi non percepiamo il tempo:** il tempo appartiene al mondo fisico, quindi occorre *interfacciarsi in qualche modo con l'hardware sottostante*
- A tal fine, la funzione statica di sistema **`Thread.sleep`** *sospende l'esecuzione per il tempo specificato (in ms)*
  - è quindi facile fare un ciclo (senza fine?) che venga eseguito *circa una volta al secondo invocando **`Thread.sleep(1000)`***
  - attenzione, però: se il ciclo è senza fine, sarà impossibile fermare il programma, se non con metodi drastici! (CTRL+C...)





# OROLOGIO: UN POSSIBILE `main`

Esempio di principio:

- detto o1 l'orologio,
- questo ciclo stampa ogni secondo l'orario corrente:

```
while (true) {  
    System.out.println(o1);  
    try { Thread.sleep(1000); }  
    catch (InterruptedException e) {}  
    o1.tic();  
}
```

Java

Si intravede il  
costrutto di gestione  
delle *situazioni*  
*eccezionali*  
(*ECCEZIONI*)

In questo caso, si specifica che se il  
programma dovesse essere  
interrotto brutalmente dall'esterno...

..non si dovrebbe fare  
assolutamente niente,  
perché è *normale così*!

# ESPERIMENTO IN JSHELL

- In JShell, non è necessario il try/catch, perché l'ambiente interattivo gestisce da solo le situazioni eccezionali
- Si presta quindi bene per un rapido esperimento
  - piccolo ciclo che stampa un intero ogni secondo
  - ... o ogni mezzo secondo

```
jshell> int i=0; while(true) {  
i ==> 0  
...> i++; System.out.println(i); Thread.sleep(1000); }
```

1  
2  
3  
4  
5  
6

Java

```
jshell> int i=0; while(true) {  
i ==> 0  
...> i++; System.out.println(i); Thread.sleep(500); }
```

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11

# Un esempio più complesso: traghetti & porti



# LA COMPAGNIA TeethFerries

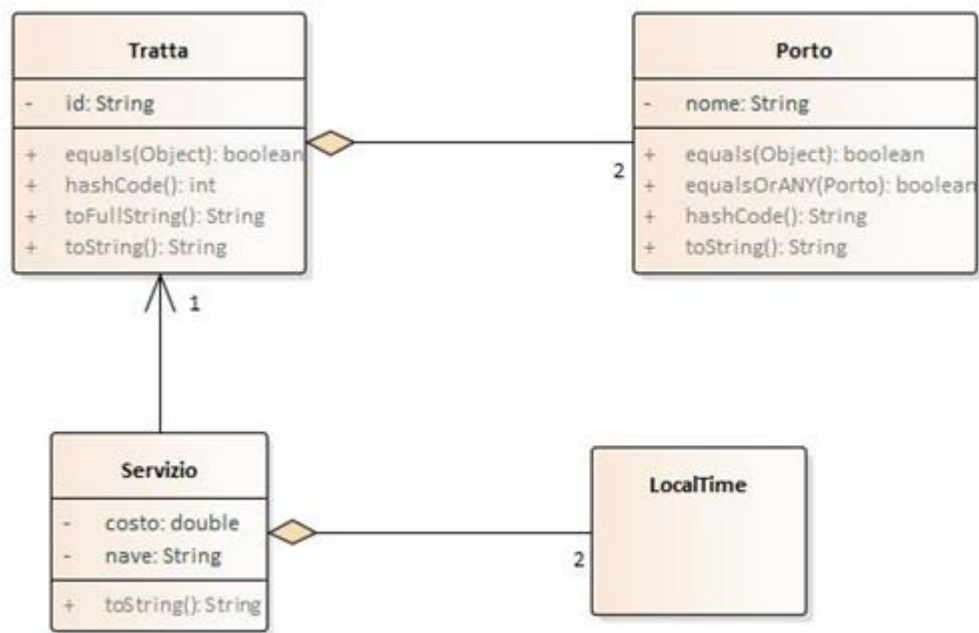
## Una compagnia di traghetti

- Analisi del dominio
  - la compagnia serve varie *tratte* fra diversi *porti*
  - su ogni tratta, la compagnia offre specifici *servizi* svolti da una ben precisa *nave* con precisi *orari* di partenza/arrivo
- Analisi di dettaglio
  - un *porto* è caratterizzato dal suo *nome*
  - una *tratta* è caratterizzata da un *identificativo* e da *due porti* (di partenza e di arrivo)
  - un *servizio* è caratterizzato da una *nave*, un *costo* e *due orari* (di partenza e di arrivo)

*Composizione:* tutte le parti sono indispensabili e hanno tempo di vita legato all'oggetto composto

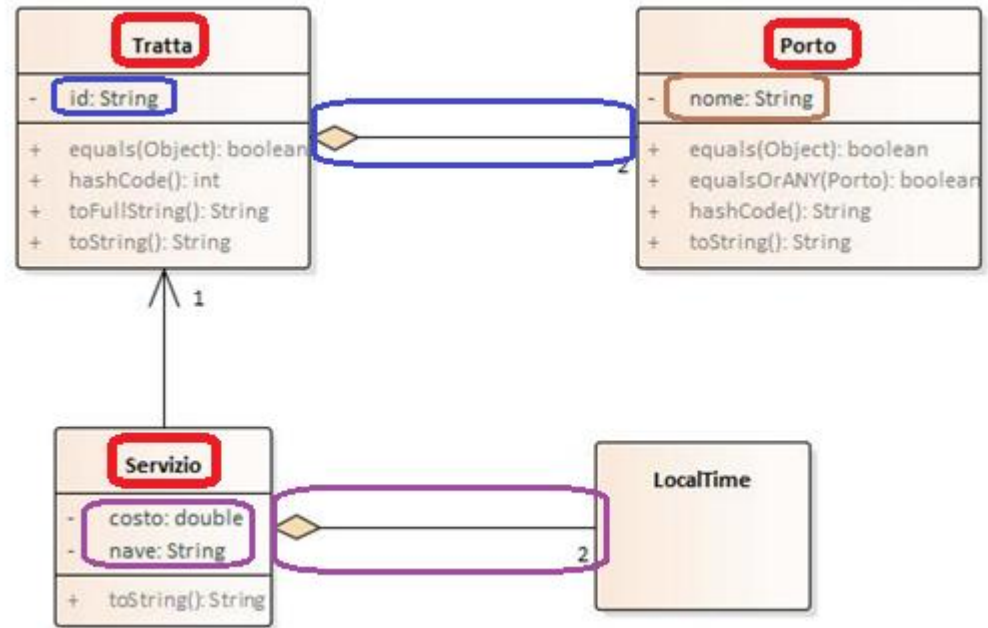
# TeethFerries: MODELLO UML

Il modello si trasporta facilmente in codice: le parti dell'oggetto composto diventano campi privati dell'oggetto "contenitore".



# TeethFerries: MODELLO UML

Il modello si trasporta facilmente in codice: le parti dell'oggetto composto diventano campi privati dell'oggetto "contenitore".





# TeethFerries: Porto

```
package teethferries.model;

public class Porto {
    private String nome;
    public Porto(String nome) {
        this.nome = nome;
    }
    public String getNome() {
        return nome;
    }
    public String toString() {
        return nome;
    }
    ...
}
```

Java

# TeethFerries: Tratta

Java

Una Tratta è composta da due Porti, partenza e arrivo

```
package teethferries.model;

public class Tratta {
    private String id;
    private Porto partenza, arrivo;

    public Tratta(String id, Porto partenza, Porto arrivo) {
        this.id = id; this.partenza = partenza; this.arrivo = arrivo;
    }

    public String getId() {
        return id;
    }

    public Porto getPortoPartenza() {
        return partenza;
    }

    public Porto getPortoArrivo() {
        return arrivo;
    }

    public String toString() {
        return "da " + getPortoPartenza() + " a " + getPortoArrivo();
    }
}
```





# TeethFerries: Servizio

Java

```
package teethferries.model;
```

```
public class Servizio {
```

```
    private String nave;
```

```
    private Tratta tratta;
```

```
    private LocalDateTime orarioPartenza, orarioArrivo;
```

```
    private double costo;
```

```
    public Servizio(String nave, Tratta tratta,
```

```
        LocalDateTime orarioPartenza, LocalDateTime orarioArrivo, double costo) {
```

```
        this.nave = nave; this.tratta = tratta; this.costo = costo;
```

```
        this.orarioArrivo = orarioArrivo; this.orarioPartenza = orarioPartenza;
```

```
    }
```

```
    public String getNave() { return nave; }
```

```
    public Tratta getTratta() { return tratta; }
```

```
    public LocalDateTime getOrarioPartenza() { return orarioPartenza; }
```

```
    ...
```

```
}
```

Un Servizio è composto da una Tratta e da due LocalDateTime, gli orari di partenza e arrivo

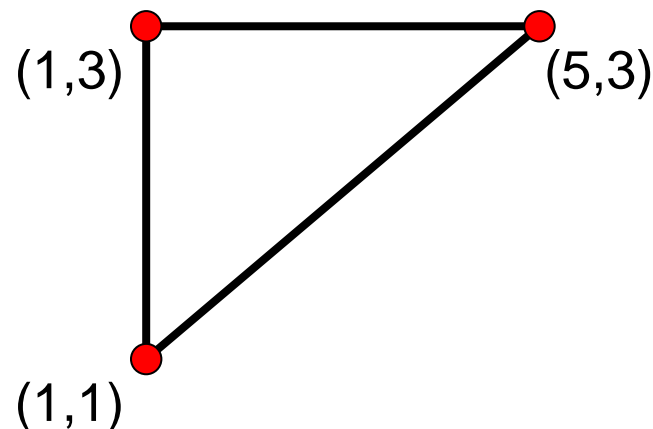
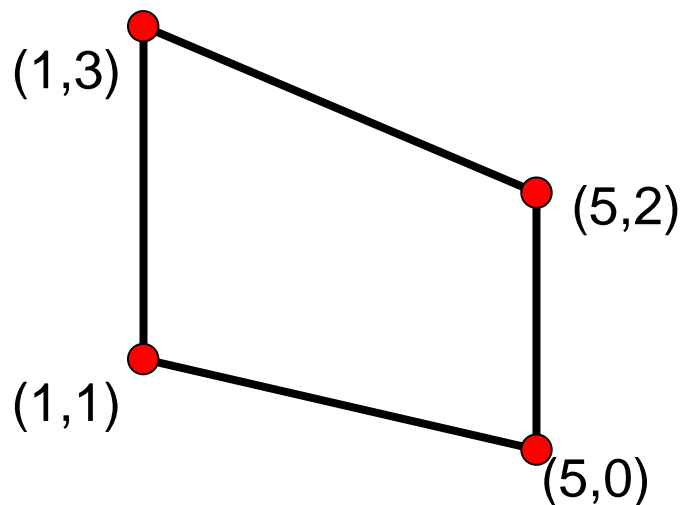


# Esercitazione: Punti & Poligoni

# ANALISI DEL DOMINIO (1/3)

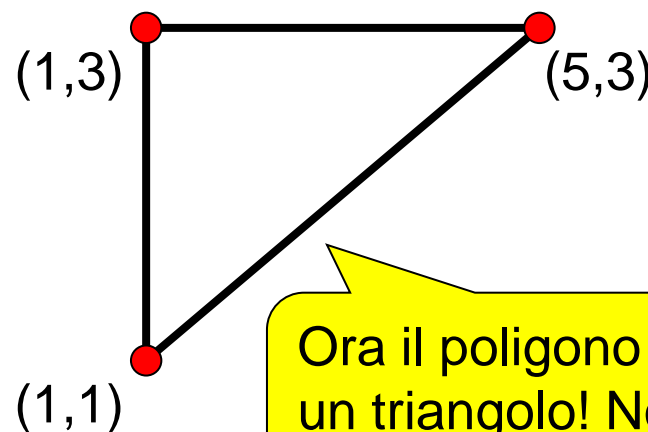
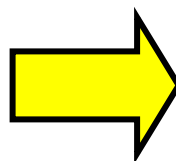
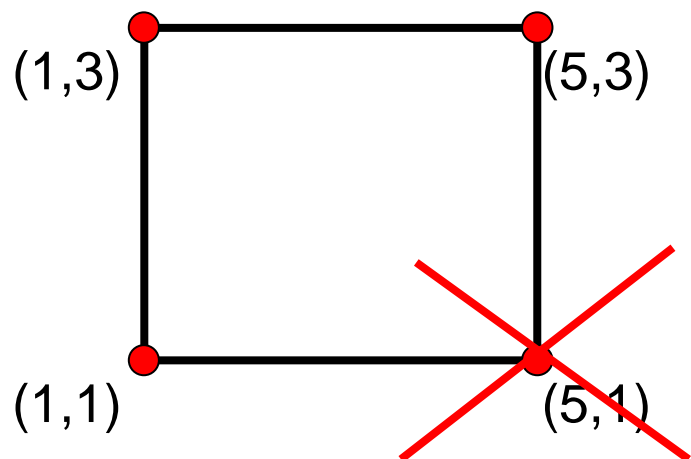
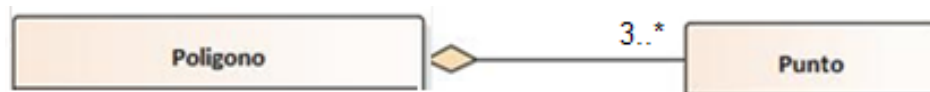
Ogni **Poligono** è caratterizzato da un **nome** (triangolo, quadrilatero, pentagono, ...) che indica implicitamente il numero di vertici e da un **insieme ordinato di punti** detti *vertici*.

ESEMPI:



# ANALISI DEL DOMINIO (2/3)

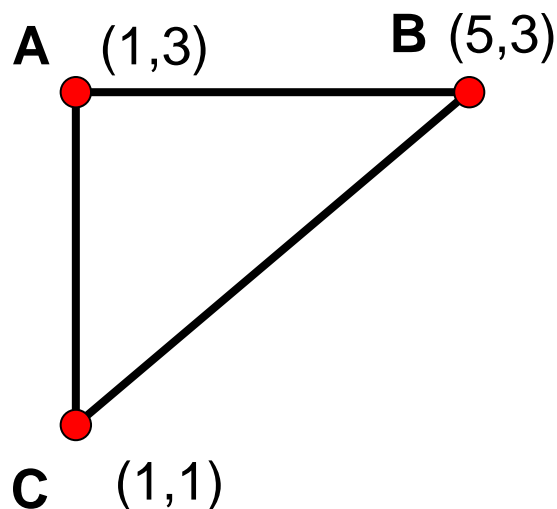
I **punti** che costituiscono i *vertici* sono **almeno 3** e **tutti indispensabili** in **quel ben preciso ordine**, poiché eliminandone uno o cambiando l'ordine dei vertici si ottiene un **poligono diverso** → *composizione*



Ora il poligono è un triangolo! Non più un rettangolo!

# ANALISI DEL DOMINIO (3/3)

- Per calcolare il *perimetro* di un poligono, occorrono i lati.
- *La lunghezza di un lato* è calcolabile a partire dalle *coordinate dei due vertici* che lo definiscono:



$$L[v_1, v_2] = \sqrt{|v_{1x} - v_{2x}|^2 + |v_{1y} - v_{2y}|^2}$$

Quindi, ad esempio:

$$L[A, B] = \sqrt{|1 - 5|^2 + |3 - 3|^2} = 4$$

$$L[B, C] = \sqrt{|1 - 5|^2 + |1 - 3|^2} = 4.47$$



# PUNTI E POLIGONI: REQUISITI (1/2)

Tipo di dato **Punto**

- L'INTERFACCIA deve fornire metodi per:
  - **costruire** un nuovo punto partendo dall'ascissa e dall'ordinata
  - **recuperare** l'ascissa del punto
  - **recuperare** l'ordinata del punto
  - **ottenere** una *rappresentazione sotto forma di stringa* del punto, come ad esempio "(2,1)"
- COLLAUDO: occorre verificare il corretto funzionamento di ogni singolo aspetto dell'astrazione **Punto**

costruttore

metodi *accessor*

metodo *toString*



# PUNTI E POLIGONI: REQUISITI (2/2)

Tipo di dato **Poligono**

- L'INTERFACCIA deve fornire metodi per:
  - *costruire* un nuovo poligono dati il *nome* e un *insieme di punti* (vertici) costruttore
  - *recuperare* il nome del poligono metodi accessor
  - *recuperare* l'insieme dei vertici che lo compongono
  - *ottenere* il *perimetro* del poligono metodo *toString*
  - *ottenere una rappresentazione sotto forma di stringa* del poligono, ad es. "triangolo di vertici (0,0) (0,3) (4,0) e perimetro 12"
- COLLAUDO: occorre verificare il corretto funzionamento di ogni singolo aspetto dell'astrazione **Poligono**



# PUNTI E POLIGONI: PROGETTO (1/2)

- Scelte riguardanti lo *stato*:
  - lo stato di ogni **Punto** è rappresentato da *ascissa* e *ordinata*
    - due valori reali
  - lo stato di ogni **Poligono** è rappresentato dal *nome* e dall'*insieme ordinato di punti* che ne costituiscono i vertici
    - una stringa
    - un array di Punti
- Scelte riguardanti i *costruttori*:
  - **Punto**: il costruttore prende in ingresso *l'ascissa* e *l'ordinata*
    - due valori reali
  - **Poligono**: il costruttore prende in ingresso il *nome* e l'*insieme ordinato dei punti* di vertice
    - una stringa
    - un array di Punti



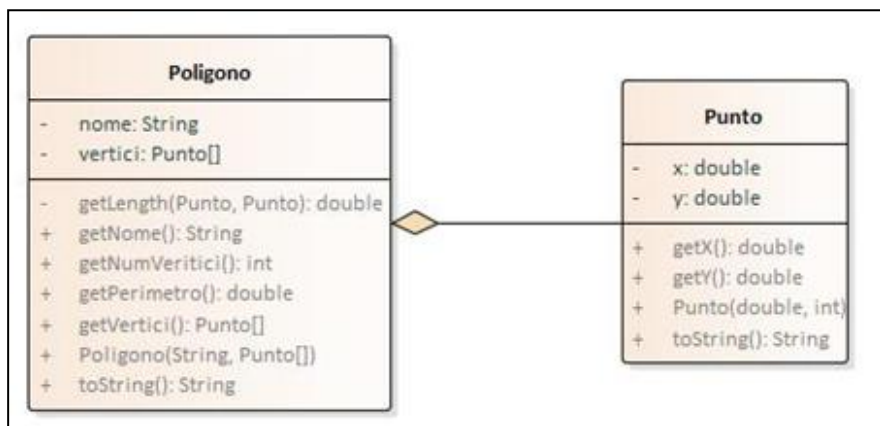


# PUNTI E POLIGONI: PROGETTO (2/2)

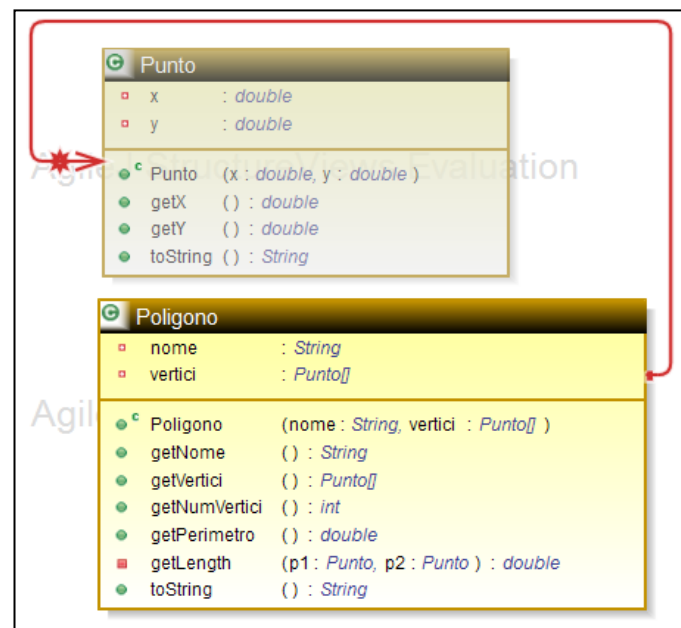
---

- Scelte riguardanti i *metodi*:
  - **Punto**
    - recuperare ascissa e ordinata → **getX**, **getY**
    - ottenere la rappresentazione sotto forma di stringa → **toString**
  - **Poligono**
    - recuperare nome e vertici → **getNome**, **getVertici**
    - ottenere il perimetro → **getPerimetro**
    - ottenere la rappresentazione sotto forma di stringa → **toString**

# MODELLO DEL PROGETTO



Notazione alternativa  
(Eclipse-like) di uno  
strumento fuori standard





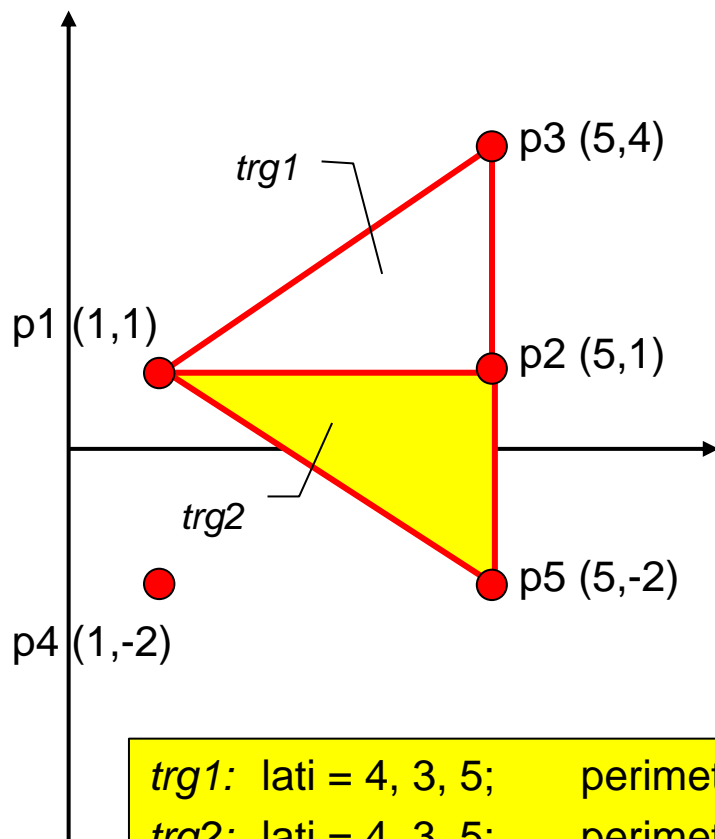
# PUNTI E POLIGONI: COLLAUDO

---

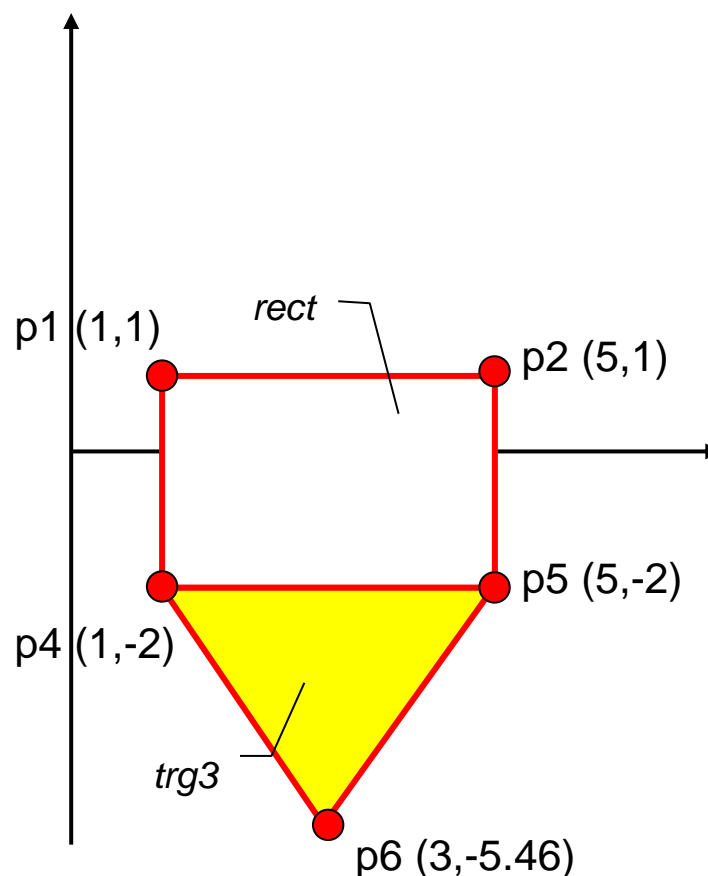
## PIANO DI COLLAUDO

1. definite le classi **Punto** e **Poligono**, collaudare innanzitutto il funzionamento dei *costruttori* (anche nei casi "particolari"...)
2. collaudare quindi il funzionamento dei *metodi accessor*
3. collaudare poi il funzionamento dei metodi **toString**, verificando che restituiscano *in tutti i casi più significativi* la stringa attesa
4. infine, collaudare il metodo **getPerimetro** verificando che restituisca *in tutti i casi più significativi* il valore atteso
  - NB: attenzione ai valori double.. e a delta di tolleranza conseguente!

# PIANO DI COLLAUDO (1/2)

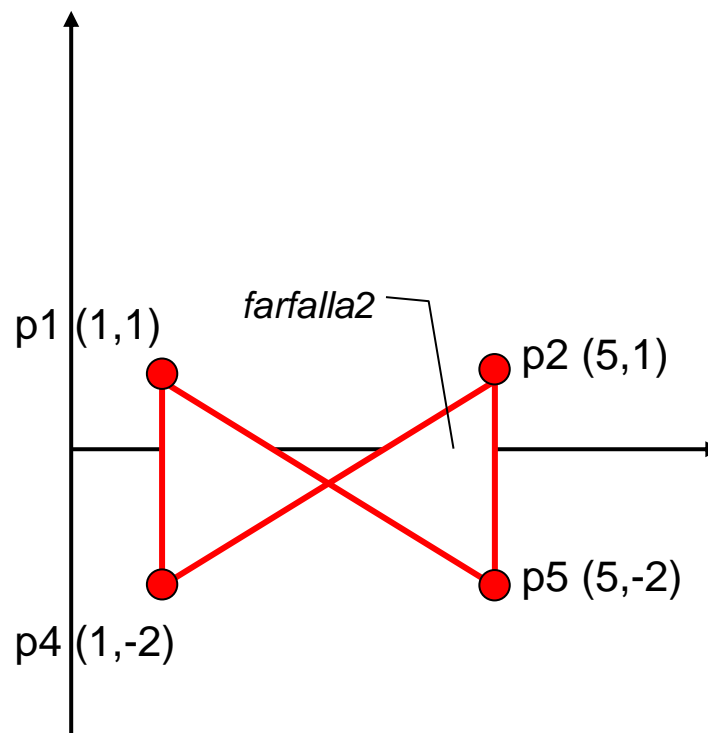
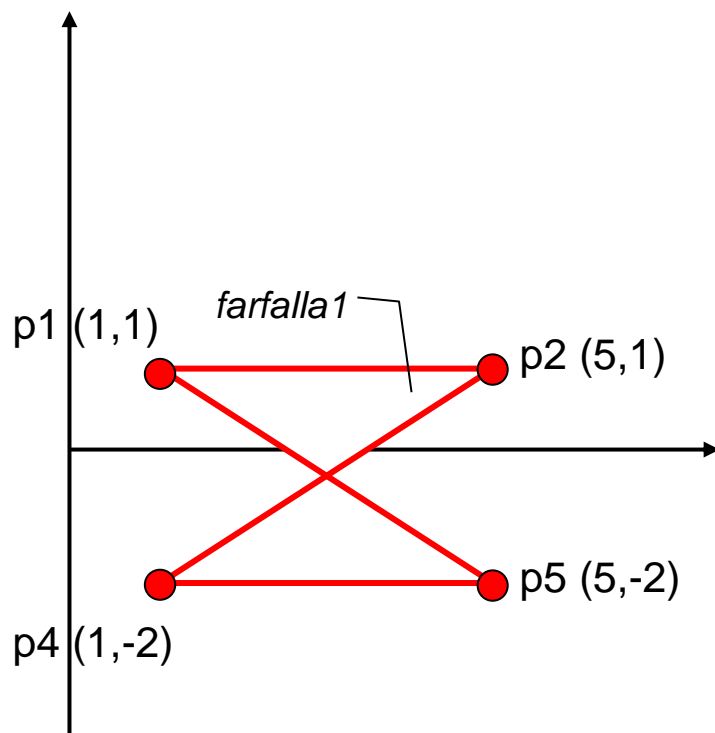


$trg1$ : lati = 4, 3, 5;	perimetro = 12
$trg2$ : lati = 4, 3, 5;	perimetro = 12
$trg3$ : lati = 4, 4, 4;	perimetro = 12
$rect$ : lati = 4, 3, 4, 3;	perimetro = 14





# PIANO DI COLLAUDO (2/2)



*rect:* lati = 4, 3, 4, 3; perimetro = 14  
*farfalla1:* lati = 4, 5, 4, 5; perimetro = 18  
*farfalla2:* lati = 3, 5, 3, 5; perimetro = 16

# Output “grafico”: il componente **Finestra**

Java



# IL COMPONENTE *Finestra*

- Ci piacerebbe poter scrivere *non sulla console*, ma su una vera *finestra grafica*
- Il componente **Finestra** svolge questo compito
  - non serve sapere *come è fatto per usarlo* (d'altronde, non vi do il sorgente ☺ .. )
  - bastano il file `.jar` e la documentazione Javadoc

ed.utils
Finestra
+print(s:String) +Finestra() : Finestra +Finestra(titolo:String) : Finestra

ed.utils	
Class Finestra	
Constructors	
Constructor and Description	
Finestra()	Costruisce una Finestra con il titolo predefinito
Finestra(java.lang.String titolo)	Costruisce una Finestra con il titolo specificato
Parameters:	
titolo - Il titolo della finestra	
Methods	
Modifier and Type	
Method and Description	
	print(java.lang.String txt)
	Stampa la stringa data nell'area di output della finestra.
Parameters:	
txt - Il testo da stampare	
void	



# Finestra: MANUALE D'USO

---

Per usare **Finestra** bisogna:

- **sapere che è nel package `ed.utils`**
  - ricorda: non si mette *mai* una libreria nel default package, perché non sarebbe richiamabile da clienti in altri package
- **costruire l'oggetto**
  - il costruttore permette di specificare il titolo desiderato
  - in aggiunta, un costruttore di default usa un titolo predefinito
- **scriverci sopra mediante il metodo `print`**
  - come `System.out`, anche **Finestra** permette solo di *aggiungere nuove scritte*, non di cancellare le precedenti.

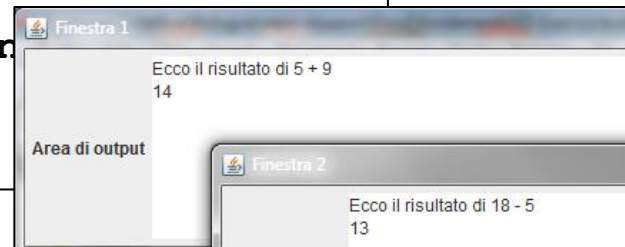


# ESEMPIO D'USO

Esempio d'uso:

```
public static void main(String args[]){  
    Finestra f1 = new Finestra("Finestra 1");  
    f1.print("Ecco il risultato di 5 + 9\n");  
    f1.print(""+(5+9));  
    Finestra f2 = new Finestra("Finestra 2");  
    f2.print("Ecco il risultato di 18 - 5\n");  
    f2.print(""+(18-5));  
}
```

Java



Come si vede, è possibile costruire anche più finestre



# UN PRIMO BILANCIO

- Non serve il sorgente per usarla: basta il **jar**
  - magari scaricato dalla rete
  - magari sviluppato su *diversa piattaforma*
  - ricordate come si usano le librerie?
    - per compilare: `javac -cp Finestra.jar;. Prova1.java`
    - per eseguire: `java -cp Finestra.jar;. Prova1`
  - non occorre "linkare" la libreria al main esplicitamente: il collegamento della classi, in Java, è dinamico
- È facile e immediato **sfruttare funzionalità che non si sarebbe in grado di costruirsi da soli**
  - l'architettura agevola il *riuso effettivo* dei componenti

Java

# PERÒ...

---

- ...è davvero opportuno lasciar costruire più finestre?
- Dipende dallo scenario d'uso:
  - se si tratta di finestre di uscita indipendenti, pilotate da un cliente che «sa il fatto suo», why not?
  - MA se rappresentano un dispositivo fisico (es. schermo, console) *che in quanto tale è unico*, allora no!  
Pensate se facesse anche input: a quale sarebbe connessa la tastiera...?
- Lesson learned: **non sempre è opportuno lasciar costruire all'utente «*tanti oggetti quanti vuole*»**
  - potremmo avere necessità di controllare / impedire le costruzioni
  - per imporre il rispetto di *vincoli logici* o *tecnologici*

# IL PUNTO CHIAVE

- In molte situazioni pratiche **è inopportuno lasciar costruire direttamente gli oggetti all'utente**
  - Rischio che non rispetti i vincoli
  - Oggetti complessi hanno costruttori complessi, con molti argomenti
  - Oggetti complessi richiedono *una competenza e conoscenza fuori dalla portata degli utenti* per essere costruiti bene
- IDEA: **far costruire gli oggetti a una fabbrica apposita**
  - Incapsula la conoscenza necessaria
  - Nasconde i dettagli del procedimento di costruzione



# IL PATTERN "FACTORY" (fabbrica)

- È un altro famosissimo *design pattern*
  - non è il primo che incontriamo: nel codice fiscale avevamo già visto (un caso semplice di) Façade..
- La *fabbrica* nasconde agli utenti la costruzione di oggetti
  - l'utente non può costruirli da solo: deve *farseli fare dalla fabbrica*
  - solo la fabbrica sa come costruirli
  - tecnicamente: la fabbrica espone una o più *funzioni statiche (factory method)* che *incapsulano e nascondono la new*
  - nomi tipici dei metodi factory:
    - una volta, `get**`
    - oggi più spesso `of`



# PERCHÉ UNA FABBRICA?

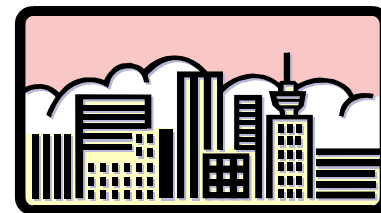
- Perché costruire un oggetto complesso spesso richiede *molto di più di una "semplice new"*
- Perché *una fabbrica può applicare decisioni strategiche* su cosa e come costruire
  - può decidere di costruire *oggetti fatti in un modo o in un altro* in base ai parametri passati o alla situazione contingente
  - può *legare la costruzione alla disponibilità di risorse*, pagamenti, o condizioni al contorno
  - può perfino *"costruire per finta" un oggetto*, prendendolo in realtà da un insieme prestabilito di oggetti già pronti
- Perché potrebbe essere utile *costruire tipi diversi (ma compatibili..) di oggetti* senza che il cliente lo debba sapere 😊





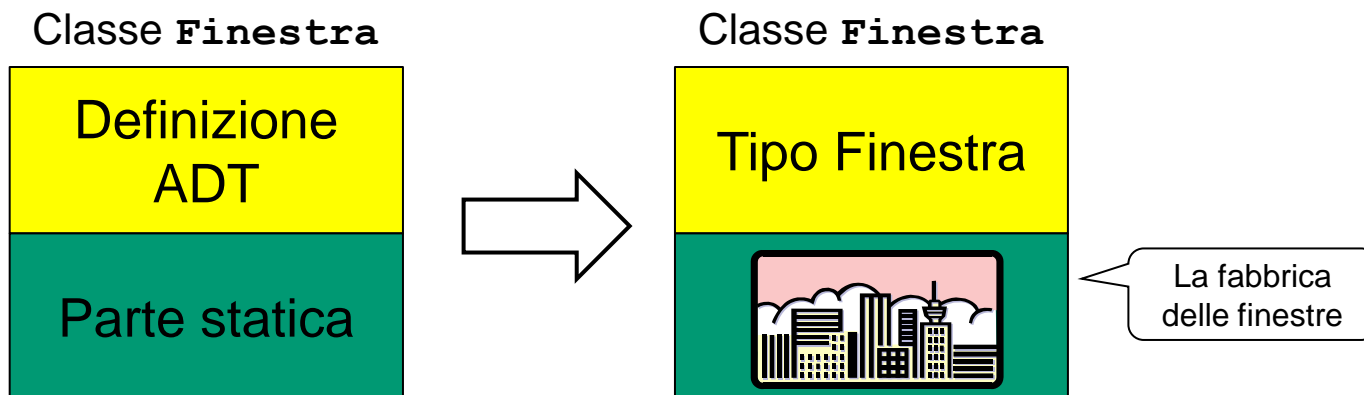
# COME È FATTA UNA FABBRICA?

- Esternamente, una fabbrica si presenta come un insieme di funzioni statiche (*factory methods*) che incapsulano la new
  - nome tipico: **of**
  - calibrano la costruzione in base agli argomenti ricevuti
  - restituiscono un oggetto creato *secondo le politiche aziendali*
- Se presente, la fabbrica è tipicamente **la sola entità** autorizzata a costruire quel certo tipo di oggetti
  - per questo, i costruttori di tali oggetti non sono più pubblici
  - la fabbrica e l'oggetto vengono *progettate insieme* (stessa classe o stesso package) per lavorare in tandem.



# LA FABBRICA DELLE FINESTRE

- Per assicurare il rispetto del vincolo di creazione (non si vuole che sia possibile creare più di una **Finestra**):
  - i due costruttori di **Finestra** *non devono più essere pubblici*
  - si aggiunge a **Finestra** una *funzione statica of* che assicuri di *creare l'oggetto solo una volta* MA cosa fa la seconda volta?
  - la fabbrica è incorporata nella stessa classe che definisce il tipo







# REFACTORING DEL COMPONENTE Finestra

## Method Summary

All Methods

Static Methods

Instance Methods

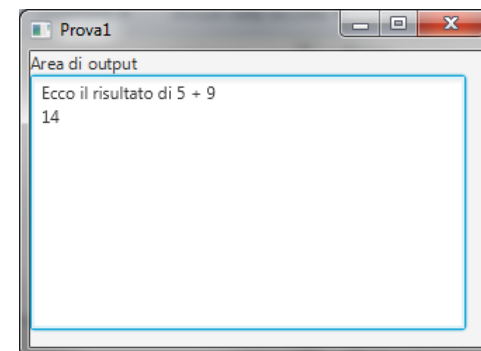
Concrete Methods

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

```
import ed.utils.Finestra;  
  
public class Prova1Refactored {  
    public static void main(String args[]){  
        Finestra f = Finestra.of("Prova1");  
        f.print("Ecco il risultato di 5 + 9\n");  
        f.print( "" + (5+9) );  
    }  
}
```

Java

```
java -cp FinestraFactory.jar;. Prova1
```





# LA FABBRICA DELLE FINESTRE

---

- Che si fa se l'utente cerca di creare un'altra **Finestra**?
  - o si restituisce *la finestra già creata*
  - o si restituisce *null* (o altro segnale di «errore»)
- Pro & contro
  - restituire la finestra già esistente garantisce che l'applicazione prosegua, ma proprio per questo *non fa emergere il design flaw*
    - l'applicazione mischierà sulla stessa finestra informazioni che avrebbero dovuto finire in posti diversi.. e ciò emergerà solo a runtime!
  - restituire *null* garantisce che *emerge il design flaw* ma naturalmente così facendo genera errore nell'applicazione
    - un errore «dovuto», ma che comunque genera un null che causerà altri disastri...

# OPZIONE 1

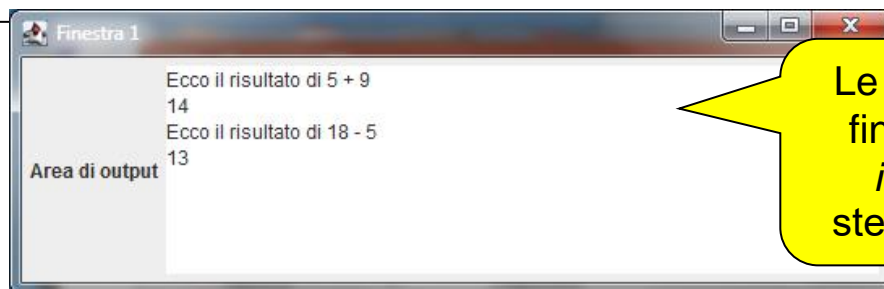
Java

```
public class Prova1Fbis {  
    public static void main(String args[]){  
        Finestra f1 = Finestra.of("Finestra 1");  
        f1.print("Ecco il risultato di 5 + 9\n");  
        f1.print(""+(5+9)+"\n");  
        // la factory restituisce la Finestra 1 originale  
        // quindi si scrive su quella  
        Finestra f2 = Finestra.of("Finestra 2");  
        f2.print("Ecco il risultato di 18 - 5\n");  
        f2.print(""+(18-5)+"\n");  
    }  
}
```

```
java -cp FinestraFactory.jar;. Prova4
```

La prima volta, crea la finestra

La seconda volta no, ma restituisce comunque una finestra valida



Le due stampe, destinate a finestre diverse, finiscono *inaspettatamente* nella stessa! Guai all'orizzonte...

# OPZIONE 2

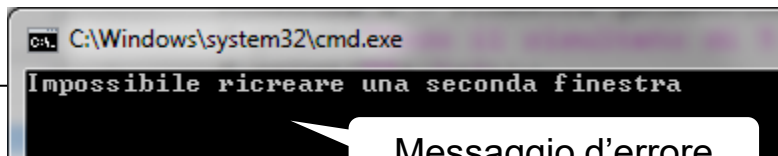
Java

```
public class Prova1Fbis {
    public static void main(String args[]){
        Finestra f = Finestra.of("Prova1");
        f.print("Ecco il risultato di 5 + 9\n");
        f.print(""+(5+9));
        Finestra g = Finestra.of("Prova2");
        if ((g!=null)) {
            g.print("Ecco il risultato di 3 + 4\n");
            g.print(""+(3+4));
        }
        else {
            System.err.println("Impossibile ricreare una seconda
            finestra");
        }
    }
}
```

La prima volta, crea la finestra

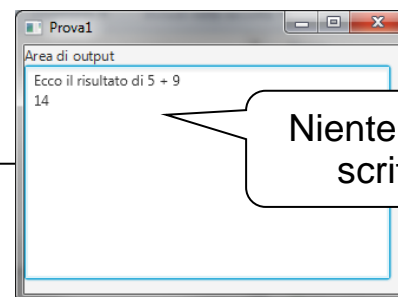
La seconda volta, no!

Contratto d'uso: tocca al cliente verificare di aver ricevuto una finestra valida, prima di usarla



Impossibile ricreare una seconda finestra

Messaggio d'errore



Ecco il risultato di 5 + 9  
14

Niente pasticci, niente scritte mischiate