



Programmazione orientata agli oggetti

Oggetti Composti

Oggetti composti - 1

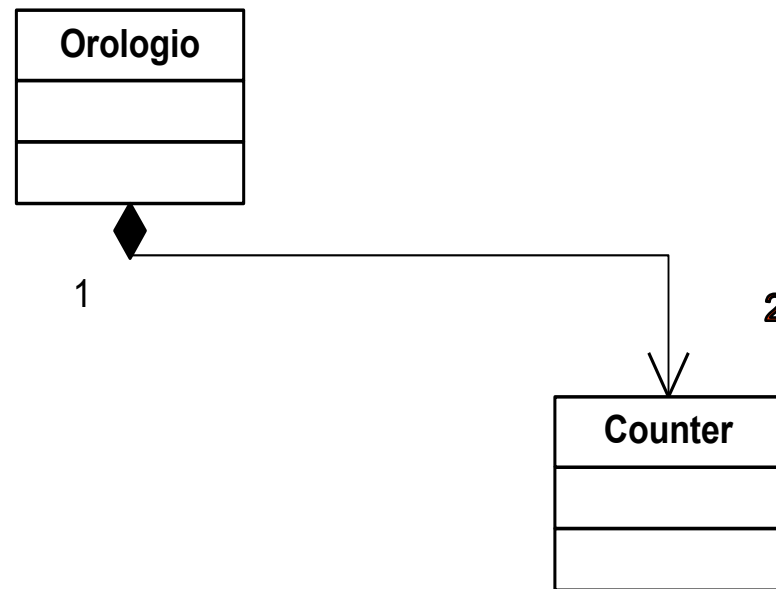
- Negli esempi che abbiamo visto finora gli attributi delle classi erano variabili di tipo primitivo
- E' però possibile definire come attributi dei riferimenti ad oggetti di qualche classe
- In questo modo abbiamo oggetti composti da altri oggetti
- Consideriamo ad esempio una classe Orologio:

```
public class Orologio
{
    private Counter ore, minuti;
}
```

- Ogni oggetto di classe Orologio ha uno **stato composto da due oggetti** di classe Counter: c'è quindi una relazione di **composizione** fra la classe Orologio e la classe Counter

Oggetti composti - 2

- Il diagramma UML che rappresenta questa relazione di composizione è:



- Orologio può usare gli oggetti di classe Counter contenuti al suo interno:
 - può quindi accedere ai metodi pubblici
 - non può accedere agli attributi e ai metodi privati

Costruzione degli oggetti - 1

- Gli attributi ore e minuti, come tutte le variabili che hanno come tipo una classe sono solo dei riferimenti
- La loro dichiarazione non implica creazione di oggetti
- Devono essere creati esplicitamente con new
- **La cosa migliore è definire un costruttore per la classe Orologio e creare i due oggetti al suo interno:**

```
public class Orologio
{
    private Counter ore, minuti;
    public Orologio()
    {
        ore = new Counter();
        minuti = new Counter();
    }
}
```

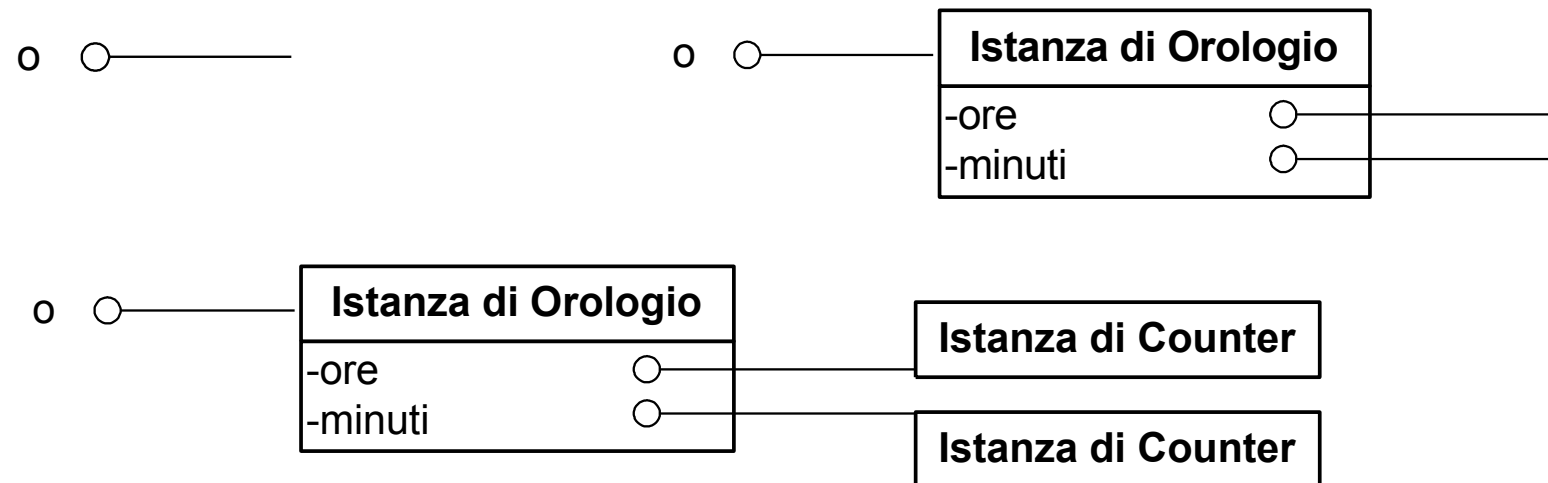
Costruzione degli oggetti - 2

- In questo modo quando creiamo un oggetto di classe Orologio abbiamo la creazione automatica degli oggetti contenuti

- Infatti se scriviamo:

```
Orologio o;  
o = new Orologio();
```

- Generiamo una sequenza di allocazioni di questo tipo:



Distruzione

- In fase di distruzione le cose sono altrettanto automatiche
- Quando l'oggetto di classe Orologio non è più usato da nessuno viene deallocato dal Garbage Collector
- La deallocazione dell'istanza di Orologio fa sparire i due attributi ore e minuti
- Questi erano gli unici riferimenti esistenti alle due istanze di classe Counter
- Quindi le due istanze non sono più utilizzate da nessuno e vengono distrutte dal Garbage Collector

Esempio: 1.Specifiche

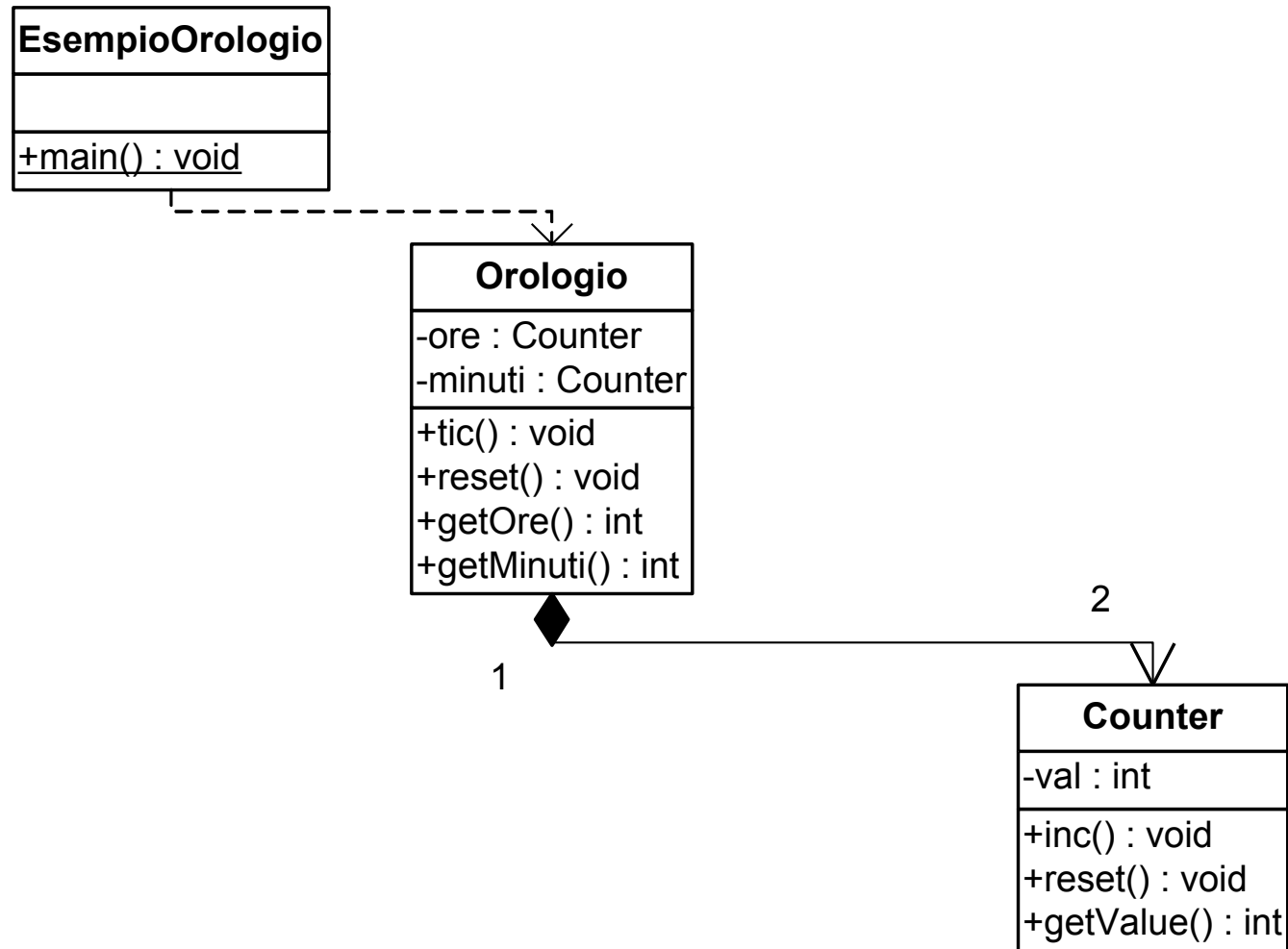
- Come sempre partiamo dalla definizione delle classi e dei comportamenti:
- La classe **Orologio** implementa un orologio con ore e minuti
- L'Orologio deve esporre i seguenti metodi pubblici:
 - **reset()** che azzerà il conteggio di ore e minuti
 - **tic()** che fa avanzare l'orologio di un minuto. Se il conteggio dei minuti arriva a 60 il contatore dei minuti si azzerà e si incrementa quello delle ore. Se il contatore delle ore è arrivato a 24 si azzerà il contatore delle ore
 - **getMinuti()** e **getOre()** che restituiscono il valore di ore e minuti
- La classe **EsempioOrologio** ha un metodo main che crea un orologio e invoca i suoi metodi

Esempio: 2. Scelte implementative

- Vediamo se nelle classi viste in precedenza abbiamo qualcosa di riutilizzabile
- La classe **Counter** è quello che ci serve per realizzare i contatori delle ore e dei minuti
- Quindi la classe **Orologio** definirà al suo interno due sue variabili di tipo **Counter** per implementare il comportamento richiesto
- Quando creiamo un'istanza della classe **Orologio** dovremo creare le due istanze di **Counter**
- I metodi di **Orologio** invocheranno i metodi di **Counter** delle due istanze
- Abbiamo quindi un meccanismo di **composizione** di oggetti

Esempio: 3.Modello

- Rappresentiamo la situazione appena descritta:



Esempio: 4.Implementazione di Orologio

```
public class Orologio
{
    private Counter ore, min;
    public Orologio()
    { ore = new Counter(); min = new Counter(); }
    public void reset()
    { ore.reset(); min.reset(); }
    public void tic()
    {
        min.inc();
        if (min.getValue() == 60)
        {
            min.reset();
            ore.inc();
        }
        if (ore.getValue() == 24)
            ore.reset();
    }
    public int getOre() { return ore.getValue(); }
    public int getMinuti() { return min.getValue(); }
}
```

Esempio: 5.Implementazione di EsempioOrologio

```
public class EsempioOrologio
{
    public static void main(String args[])
    {
        Orologio o;
        o = new Orologio();
        o.tic();
        o.tic();
        System.out.println(o.getOre());
        System.out.println(o.getMinuti());
        o.reset();

        System.out.println(o.toString());
    }
}
```

// DEFINIRE toString()!!!!

Esempio: 4.Implementazione di Orologio

```
public class Orologio
{
    private Counter ore, min;
    public Orologio()
    { ore = new Counter(); min = new Counter(); }
    public void reset()
    { ore.reset(); min.reset(); }
    public void tic()
    {
        min.inc();
        if (min.getValue() == 60)
        {
            min.reset();
            ore.inc();
        }
        if (ore.getValue() == 24)
            ore.reset();
    }
    public int getOre(){return ore.getValue();}
    public int getMinuti(){return min.getValue();}
    public String toString()
        {return "Ore"+ore.getValue()+" Min"+min.getValue();}
}
```

Esempio: 4.Implementazione di Orologio

```
public String toString()  
    {return ore.toString()+min.toString();}
```

Esempio: 6.Considerazioni

- E' importante notare che nel costruire questa applicazione abbiamo applicato in modo esteso il principio di separazione fra interfaccia e implementazione
- La classe **Orologio** usa due istanze della classe **Counter** basandosi solo sui metodi pubblici (interfaccia)
- Non viene fatta la minima ipotesi su come **Counter** sia fatta al suo interno (implementazione)
- Lo stesso accade per **EsempioOrologio** : usa i metodi pubblici di **Orologio** e non si preoccupa minimamente di come questo sia fatto
- In questo esempio c'è l'essenza del modo di procedere con il modello ad oggetti: **abbiamo costruito per strati la nostra applicazione**

- **Astrazione (separazione interfaccia e implementazione)**
- **Riuso**

Riutilizzo di componenti e loro raffinamento

- Spesso si incontrano problemi che richiedono **componenti simili** ad altri già disponibili, **ma non identici**
- Altre volte, **l'evoluzione dei requisiti** comporta una corrispondente **modifica dei componenti**:
 - necessità di **nuovi dati e/o nuovi comportamenti**
 - necessità di **modificare il comportamento** di metodi già presenti
- Come fare per non dover rifare tutto da capo?

Riutilizzo: approcci

- ricopiare manualmente il codice della classe esistente e cambiare quel che va cambiato

Riutilizzo: approcci

- ricopiare manualmente il codice della classe esistente e cambiare quel che va cambiato
- **creare un oggetto composto** (e usare delega)
 - che incapsuli il componente esistente...
 - ... gli “inoltri” le operazioni già previste...
 - ... e crei, sopra di esso, le nuove operazioni richieste (eventualmente definendo nuovi dati)
 - sempre che ciò sia possibile!

Riutilizzo: approcci

- ricopiare manualmente il codice della classe esistente e cambiare quel che va cambiato
- **creare un oggetto composto** (e usare delega)
 - che incapsuli il componente esistente...
 - ... gli “inoltri” le operazioni già previste...
 - ... e crei, sopra di esso, le nuove operazioni richieste (eventualmente definendo nuovi dati)
 - sempre che ciò sia possibile!
- **specializzare (per ereditarietà) il tipo di componente**

Esempio (esercizio tutorato)

Dal contatore Counter (solo avanti) ...

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

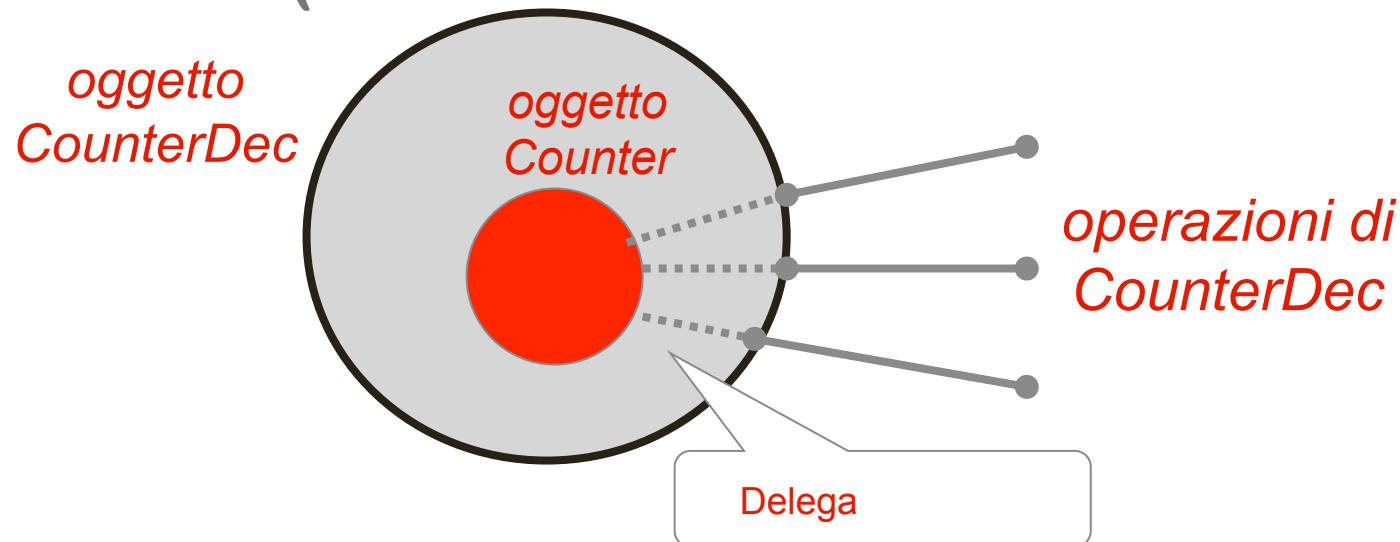
Esempio (esercizio tutorato)

Dal contatore Counter (solo avanti) ...

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

... al contatore avanti/indietro (CounterDec)

- **creare un oggetto composto di tipo CounterDec**
 - che incapsuli il componente esistente (di tipo Counter) ...
 - ... gli “inoltri” le operazioni già previste...
 - ... e crei, sopra di esso, le nuove operazioni richieste (eventualmente definendo nuovi dati)



... al contatore avanti/indietro (CounterDec)

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

```
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() {  
        int n=c.getValue();  
        c.reset();  
        for (int i=1; i<n; i++) c.inc();  
    }  
}
```



... al contatore avanti/indietro (CounterDec)

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

```
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() {  
        int n=c.getValue();  
        c.val = n-1; }  
}
```

