

OGGETTI COMPOSTI

- Una classe può contenere *referimenti a altre classi* (o anche a se stessa):

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

- L'oggetto Orologio ("contenitore") può *usare* gli oggetti Counter ore e minuti...
- ..ma *non può accedere* ai loro *campi privati*
- Può accedere a dati e funzioni *pubbliche* e a quelli con *visibilità package* (se la classe contenitore è definita nello stesso package)

OGGETTI COMPOSTI - COSTRUZIONE

- In fase di costruzione, *il costruttore dell'oggetto “contenitore” deve costruire esplicitamente, con new, gli oggetti “interni”*

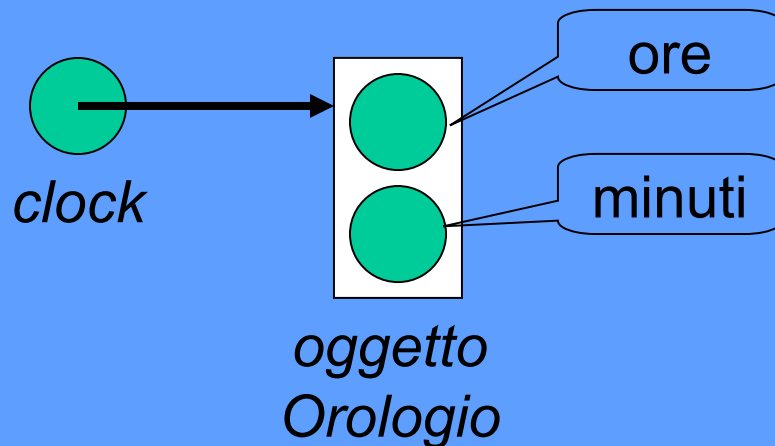
```
public class Orologio {  
    Counter ore, minuti;  
  
    public Orologio() {  
        ore = new Counter(0);  
        minuti = new Counter(0);  
    }  
}
```

OGGETTI COMPOSTI - COSTRUZIONE

Quindi:

- *prima* si costruisce l'oggetto contenitore (fase 1)
...

```
Orologio clock = new Orologio();
```

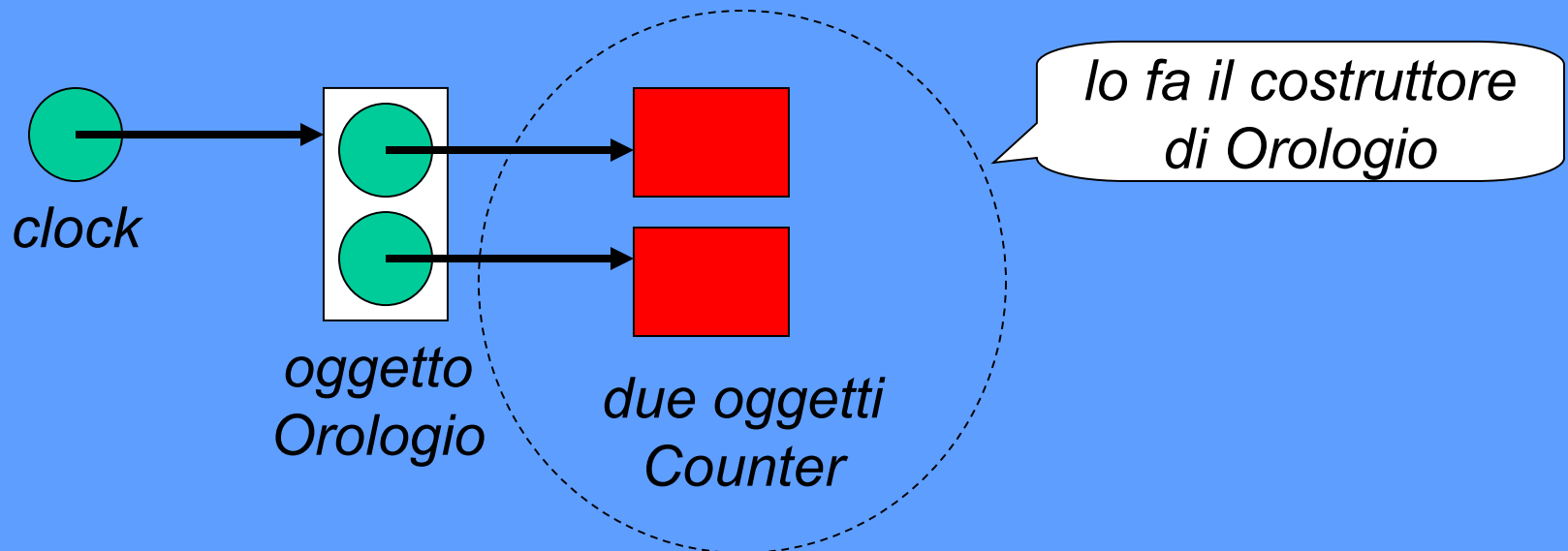


OGGETTI COMPOSTI - COSTRUZIONE

Quindi

- *prima* si costruisce l'oggetto contenitore (fase 1)...
- *poi* esso costruisce gli oggetti interni (fase 2).

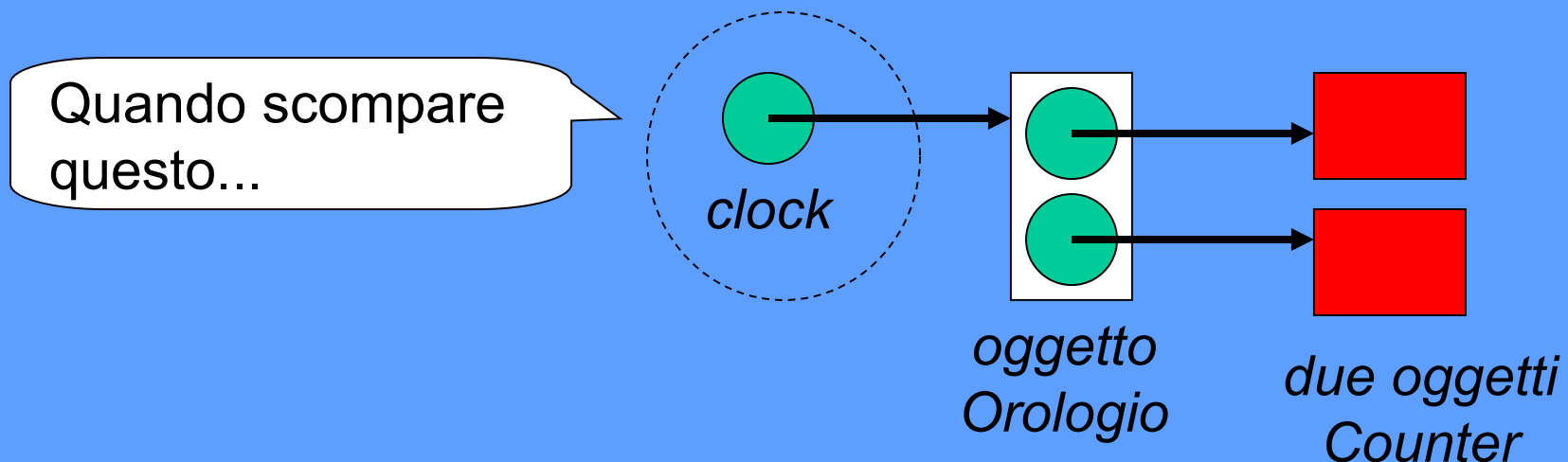
```
ore = new Counter(0);  
minuti= new Counter(0);
```



OGGETTI COMPOSTI - DISTRUZIONE

In fase di distruzione:

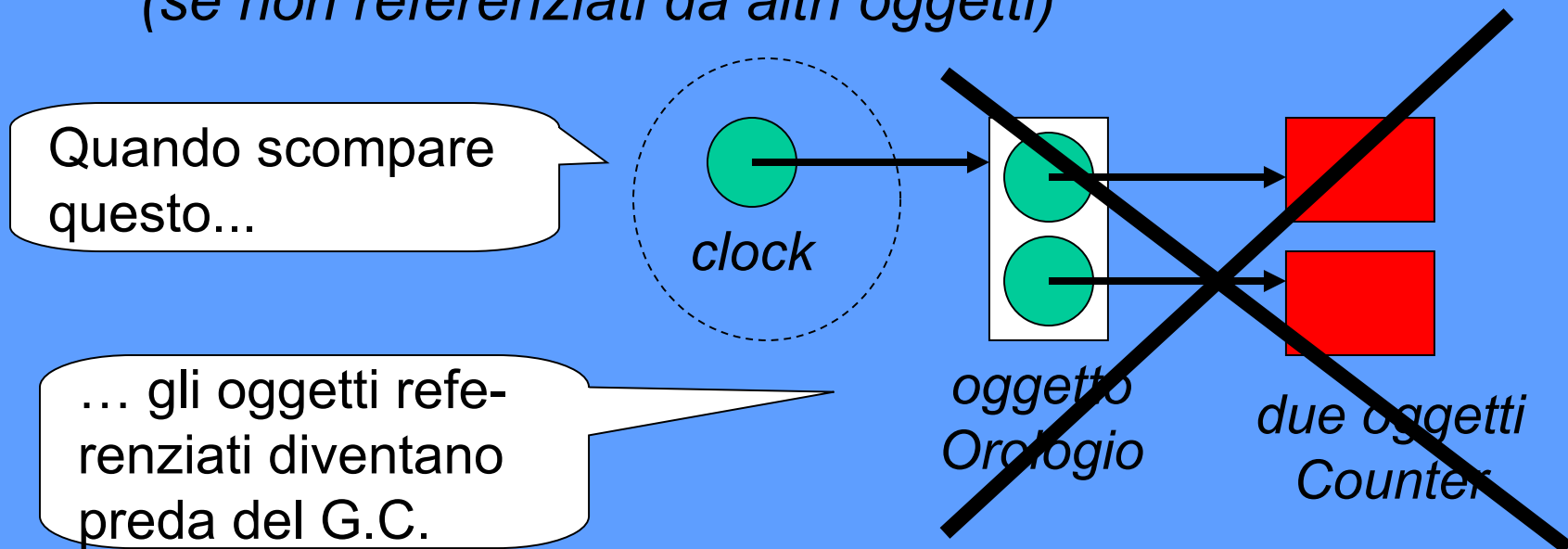
- quando il riferimento all'oggetto contenitore viene distrutto, l'oggetto passa al garbage collector, che lo distruggerà quando vorrà
- la stessa fine fanno gli oggetti referenziati da esso
(se non referenziati da altri oggetti)



OGGETTI COMPOSTI - DISTRUZIONE

In fase di distruzione:

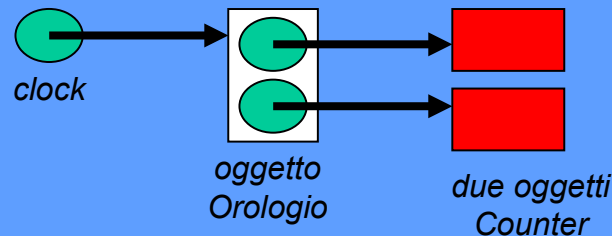
- quando il riferimento all'oggetto contenitore viene distrutto, l'oggetto passa al garbage collector, che lo distruggerà quando vorrà
- la stessa fine fanno gli oggetti referenziati da esso
(se non referenziati da altri oggetti)



OGGETTI COMPOSTI - USO

Solo i metodi dell'oggetto “contenitore”
possono accedere agli oggetti “interni”

```
public class Orologio {  
    Counter ore, minuti;  
  
    public void tic() {  
        minuti.inc();  
        if (minuti.getValue()==60) {  
            minuti.reset(); ore.inc();  
            if (ore.getValue()==24) ore.reset();  
        }  
    }  
}
```



VANTAGGI E LIMITI

Gli oggetti composti **consentono**:

- di *aggregare componenti complessi* a partire da componenti più semplici già disponibili
- di mantenere l' "unitarietà" del componente, assicurarne la protezione e l'incapsulamento

E se invece occorre

- *specializzare componenti già esistenti ?*
- *aggiungere loro nuovo dati o metodi ?*

Ad esempio, un contatore *con decremento*?

PROGETTAZIONE INCREMENTALE

- Spesso si incontrano problemi che richiedono **no 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 evitare di dover riprogettare tutto da capo?

PROGETTAZIONE INCREMENTALE

Finora, abbiamo solo due possibilità:

- *ricopiare manualmente il codice della classe esistente e cambiare quel che va cambiato*
- *creare un oggetto composto*
 - 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*

ESEMPIO

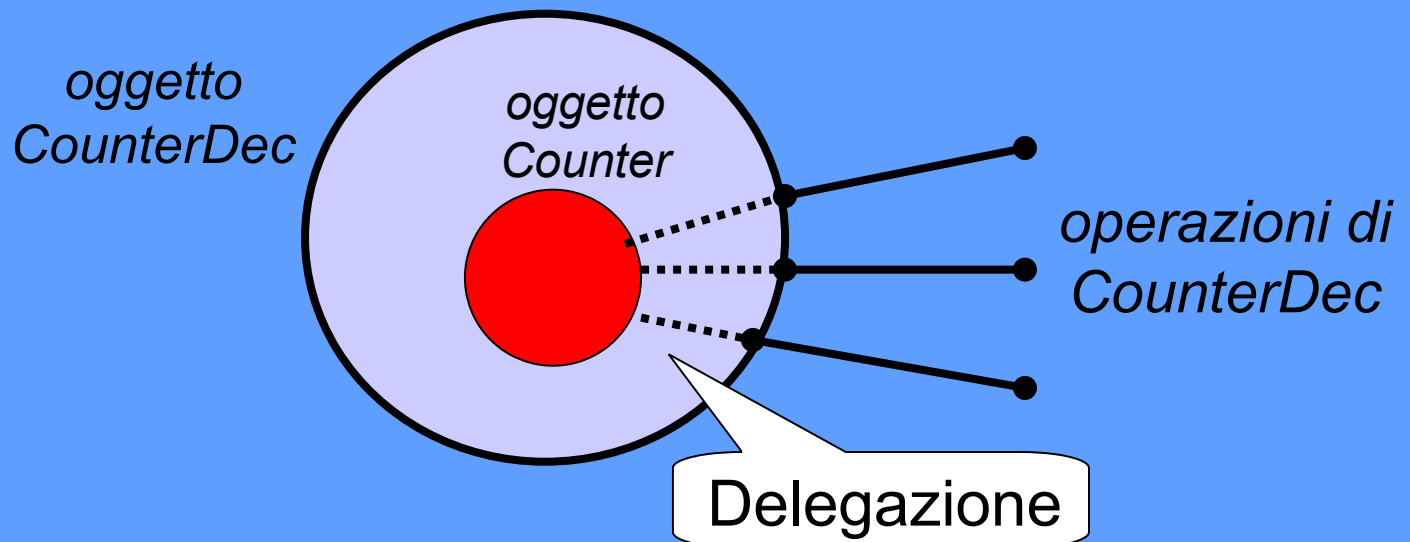
Dal contatore (solo in avanti) ...

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

ESEMPIO

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

- Concettualmente, ogni oggetto CounterDec **ingloba un oggetto Counter al suo interno**
- Ogni operazione richiesta a CounterDec viene **delegata** all'oggetto Counter interno



ESEMPIO

... 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() { ... }  
}
```

Delegazione

ESEMPIO

Il metodo `dec()` può essere così definito:

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

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

In alternativa, si può sfruttare il costruttore con un parametro per ri-costruire `c` direttamente al nuovo valore desiderato.

CONCLUSIONE

- Poiché i campi privati non sono accessibili, *bisogna riscrivere anche tutti i metodi che concettualmente rimangono uguali*, procedendo per delegazione.
- *Non è detto* che le operazioni già disponibili consentano di *ottenere qualsiasi nuova funzionalità* si renda necessaria (potrebbe essere necessario accedere ai dati privati)
- **Occorre poter riusare le classi esistenti in modo più flessibile.**