Capitolo 12

Il paradigma orientato agli oggetti

Verso un nuovo *paradigma*

- i principi di astrazione richiedono di “trattare assieme” i dati e le operazioni su di essi
- tuttavia:
  - gli ADT sono rigidi
  - permettono solo difficilmente estensioni

- Una diversa prospettiva (e nuovi supporti linguistici)…

Obiettivi

- Costrutti per
  - Information hiding e incapsulamento
  - riuso del codice (ereditarietà)
  - compatibilità tra tipi (sottotipi)
  - selezione dinamica delle operazioni

Oggetti e classi

- Un oggetto è una capsula che contiene
  - dati nascosti: variabili, valori (talvolta anche operazioni)
  - operazioni pubbliche: operazioni, dette *metodi*

- Un programma orientato agli oggetti
  - mandare messaggi agli oggetti
  - un oggetto risponde al msg
  - stato confinato negli oggetti

- *Principi di organizzazione* permettono di raggruppare gli oggetti con la stessa struttura (p.e. le classi). Gli stessi principi consentono *estensibilità* e *riuso*.

- Astrazione sui dati e sul controllo, information hiding e incapsulamento sono presenti nel progetto sin dall'inizio.

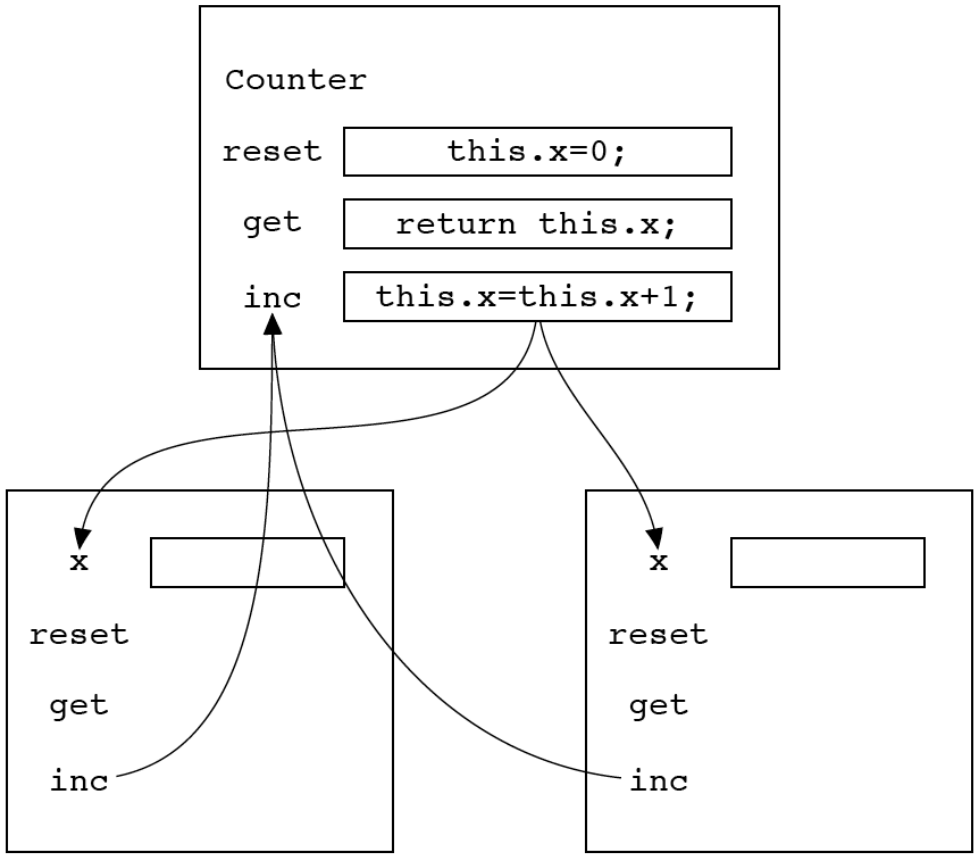
dati nascosti	
msg_1	$metodo_1$
. . .	. . .
msg_n	$metodo_n$

Classi

- Una classe è un modello di un insieme di oggetti:
  - quali dati
  - nome, segnatura e visibilità delle operazioni (metodi)

- Oggetti creati dinamicamente per *istanziazione* di una classe, mediante *costruttori*

```
class Counter{
    private int x;
    public void reset(){
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}
```



Incapsulamento

- Le classi garantiscono incapsulamento
  - opportuni *modificatori* assicurano che determinati campi sono
    - pubblici
    - privati
    - protetti
  - la parte pubblica costituisce l' "*interfaccia*" della classe
  - la parte privata è l'implementazione
- Gli altri meccanismi assicurano che l'incapsulamento sia compatibile con l'estensibilità e la modificabilità del codice

Sottotipi

```
class Counter{
   private int x;
   public void reset(){
      x = 0;
   }
   public int get(){
      return x;
   }
   public void inc(){
      x = x+1;
   }
}
```

```
class NamedCounter extending Counter{
   private String nome;
   public void set_name(String n){
      nome = n;
   }
   public String get_name(){
      return nome;
   }
}
```

- **`NamedCounter`** è una sottoclasse (o classe derivata) di **`Counter`**:
  - ogni istanza di **`NamedCounter`** risponde a tutti i metodi di **`Counter`** (più altri)
  - In quanto tipi, **`NamedCounter`** è *compatibile* con **`Counter`**
  - **`NamedCounter`** è *un sottotipo* di **`Counter`**

Ridefinizione di metodo (overriding)

```
class NewCounter extending Counter{
   private int num_reset = 0;
   public void reset(){
      x = 0;
      num_reset = num_reset + 1;
   }
   public int quanti_reset(){
      return num_reset;
   }
}
```

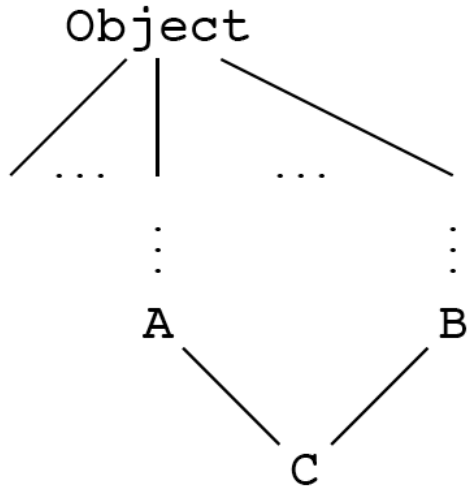
- **`NewCounter`** contemporaneamente:
  - estende l'interfaccia di **`Counter`** con nuovi campi
  - ridefinisce il metodo **`reset`**
- Un messaggio **`reset`** inviato ad un'istanza di **`NewCounter`** causa l'invocazione del *nuovo* codice

La relazione di sottotipo

```
abstract class A{
   public int f();
}

abstract class B{
   public int g();
}

class C extending A,B{
   private x = 0;
   public int f(){
      return x;
   }
   public int g(){
      return x+1;
   }
}
```



- Un tipo può avere *più di un* sovratipo immediato
- La situazione si presenta in Java solo quando i sovratipi sono *classi totalmente astratte* (cioè senza implementazioni: nel gergo Java tali “classi” si chiamano *implementazioni*)
- La gerarchia dei sottotipi non è un albero

Ereditarietà

```
class Counter{
   private int x;
   public void reset(){
      x = 0;
   }
   public int get(){
      return x;
   }
   public void inc(){
      x = x+1;
   }
}
```

```
class NewCounter extending Counter{
   private int num_reset = 0;
   public void reset(){
      x = 0;
      num_reset = num_reset + 1;
   }
   public int quanti_reset(){
      return num_reset;
   }
}
```

- **`NewCounter`** *eredita* da **`Counter`** i metodi (e i campi) non ridefiniti
- L'ereditarietà permette il riuso del codice in un contesto estendibile:
  - ogni modifica all'implementazione di un metodo in una classe è automaticamente disponibile a tutte le sottoclassi.

Ereditarietà e sottotipo

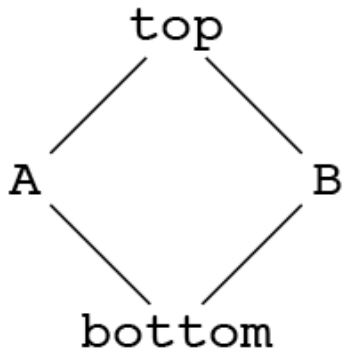
- Sottotipo
  - ha a che vedere con la possibilità di usare un oggetto in un altro contesto
  - è una relazione tra le “interfacce” di due classi.
- Ereditarietà
  - ha a che vedere con la possibilità di riusare il codice che manipola un oggetto
  - è una relazione tra le “implementazioni” di due classi.

- Due meccanismi del tutto indipendenti
- (spesso) linguisticamente collegati nei linguaggi o.o
  - sia C++ che Java hanno costrutti che introducono *contemporaneamente* entrambe le relazioni tra due classi
  - p.e.: **`extends`** in Java introduce un sottotipo e *può* introdurre ereditarietà se ci sono metodi della superclasse che non sono ridefiniti nella sottoclasse

Ereditarietà singola e multipla

- Singola (Java)
  - ogni classe eredita *al più* da una sola superclasse immediata
- Multipla (C++)
  - una classe *può* ereditare da più di una superclasse immediata
  - Problemi:
    - name clash: un'istanza di **`Bottom`** eredita **`f`** da **`B`** o da **`Top`**?
    - implementazione

```
class Top{
   int w;
   int f(){
      return w;
   }
}
class A extending Top{
   int x;
   int g(){
      return w+x;
   }
}
class B extending Top{
   int y;
   int f(){
      return w+y;
   }
   int k(){
      return y;
   }
}
class Bottom extending A,B{
   int z;
   int h(){
      return z;
   }
}
```



Selezione dinamica dei metodi

- Un metodo m viene invocato su un oggetto ogg:
  - vi possono essere più versioni di m (per overriding e sottotipi)
  - come avviene la scelta di quella davvero invocata?
- Selezione *dinamica*
  - a tempo d'esecuzione, in funzione *del tipo dell'oggetto* che riceve il messaggio
- Attenzione: tipo dell'oggetto, non tipo del riferimento (o nome) di quell'oggetto

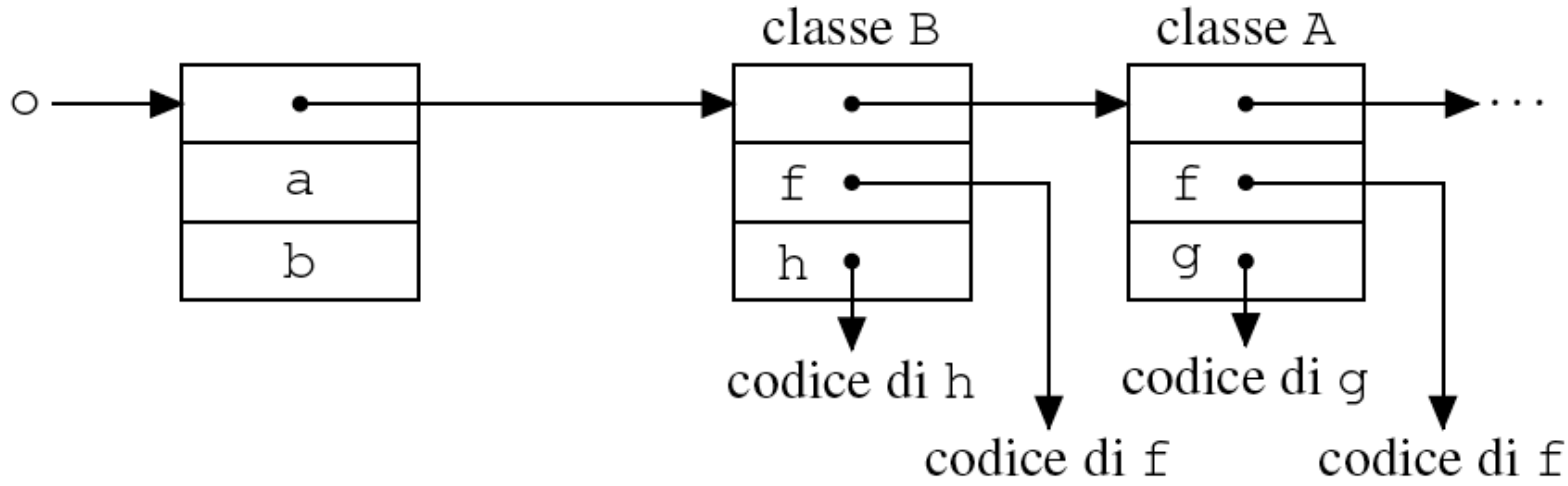
```
class Counter{
   private int x;
   public void reset(){
      x = 0;
   }
   public int get(){
      return x;
   }
   public void inc(){
      x = x+1;
   }
}

  class NewCounter extends Counter{
     private int num_reset = 0;
     public void reset(){
        x = 0;
        num_reset = num_reset + 1;
     }
     public int quanti_reset(){
        return num_reset;
     }
  }
```

```
NewCounter n = new Counter();
Counter c;
c = n;
c.reset();
```

Semplice implementazione dell'ereditarietà

```
class A{
    int a;
    void f(){...}
    void g(){...}
}
class B extending A{
    int b;
    void f(){...} // ridefinito
    void h(){...}
}
B o = new B();
```



Ereditarietà singola con tipi statici

```
class A{
    int a;
    char c;
    void g(){...}
    void f(){...}
}
class B extending A{
    int a;
    int b;
    void h(){...}
    void f(){...}
        // ridefinito

}
B pb = new B();
A pa = new A();
A aa = b;
aa.f();
```