

Introduzione all'Object-Orientation

Angelo Gargantini

Capitolo 10 del Mitchell

Schema della lezione

1. Cenni di progettazione Object-oriented
2. Concetti principali dell'object-orientation
 - incapsulamento
 - sottotipazione
 - ereditarietà
 - binding dinamico

Oggetti - Objects

- Un oggetto consiste in
 - dati nascosti
 - dati o variabili (di istanza)
 - anche possibili funzioni
 - operazioni pubbliche
 - metodi o funzioni membro
 - anche possibili variabili



- Sistemi Object-oriented

- oggetti **mandano** messaggi ad altri oggetti
 - (chiamate di funzioni/metodi)

object → msg(arguments)

object.method(arguments)

Cosa c'è di interessante?

- Costrutto di incapsulamento generale
 - Strutture Dati
 - File system
 - Database
 - Window
 - Sistema Operativo ...
- Metafora utilmente ambigua
 - computazione sequenziale o concorrente
 - comunicazione distribuita, sincrona, asincrona

Object-orientation

- Tutto è “Object-Oriented” ?
- Per noi è:
 - metodologia di progettazione/programmazione
 - organizzare concetti in oggetti e classi
 - costruire sistemi estensibili
 - utilizzando i seguenti concetti
 - dati e funzioni sono incapsulati in **oggetti**
 - la **sottotipazione** permette l'estensione dei tipi di dati
 - **l'ereditarietà** permette il riuso delle implementazioni

Progettazione Object-oriented

[Booch]

◆ Quattro passi

1. Identifica gli **oggetti** ad un certo livello d'astrazione
2. Identifica la semantica (cioè il **comportamento** desiderato) degli oggetti
3. Identifica **le relazioni** tra gli oggetti
4. **Implementa** gli oggetti

◆ Processo iterativo

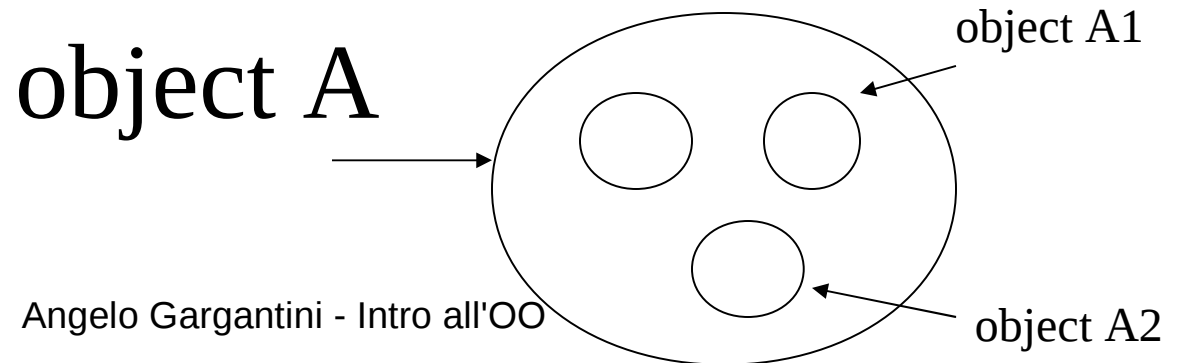
- Implementa gli oggetti (punto 4) mediante i quattro passi

◆ Non necessariamente “top-down”

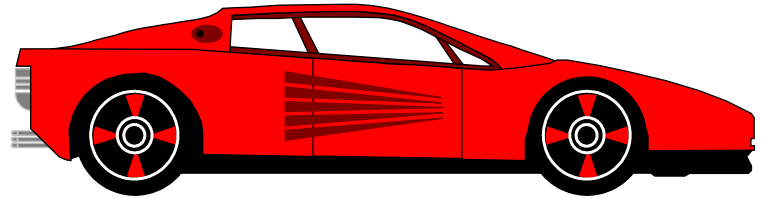
- “livello d'astrazione” a qualsiasi livello

Progettazione OO

- Associa oggetti ai **componenti** o ai **concetti** di un sistema
- Perché iterativo (**raffinamento**)?
 - Un oggetto è tipicamente implementato usando un numero di oggetti che lo costituiscono
 - Si applica la stessa metodologia agli oggetti individuati (componenti o concetti)



Esempio: calcolo del peso di una automobile



- Oggetto “AUTO” :
 - Contiene una lista delle sue parti principali
 - telaio, motore, ruote,
 - Metodo per calcolare il peso
 - somma il peso dei componenti
- Oggetti componenti:
 - Ognuno può avere una lista delle sottocomponenti
 - Ognuno deve avere un metodo per il calcolo del peso

Confronto con la progettazione top-down

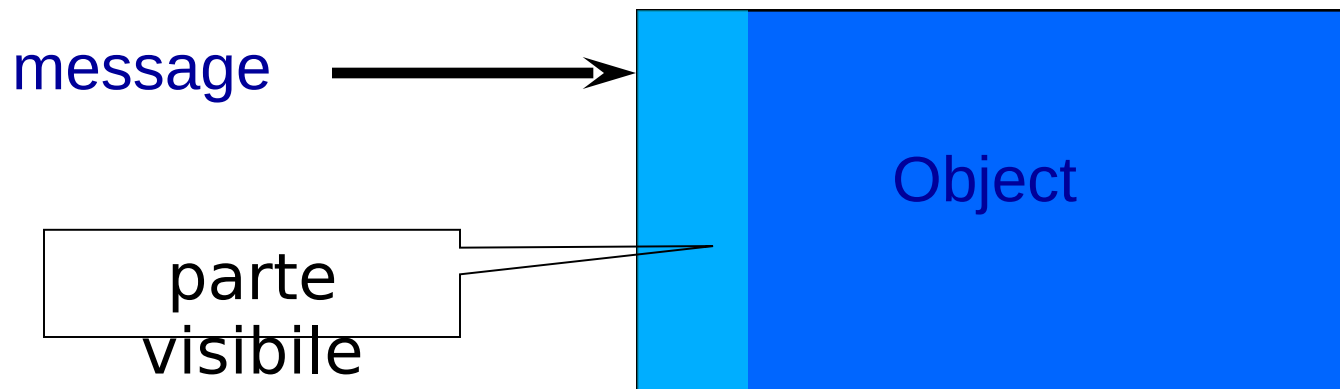
- Somiglianza:
 - Un compito viene portato a termine completando un numero di sotto compiti più piccoli (divide et impera)
- Però:
 - si raffinano non solo le procedure ma anche la rappresentazione dei **dati**
 - **modellare** i concetti (dati e operazioni) del sistema
 - gli oggetti raggruppano dati e funzioni rendendo il raffinamento più naturale

Concetti dell'Object-Orientation

- **incapsulamento - encapsulation**
- sottotipazione - subtyping
 - per estendere i concetti
- ereditarietà - inheritance
 - per riusare le implementazioni
- binding dinamico - dynamic lookup

Incapsulamento

- chi **costruisce** l'oggetto ha (deve avere) una vista dettagliata
- chi usa un oggetto (utente o **cliente**) ha una vista **astratta**
- **L'incapsulamento** è il meccanismo per separare queste due viste

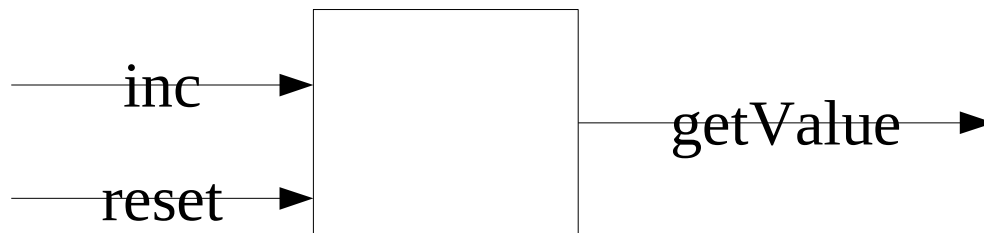


Incapsulamento e ling. di programmazione

- Esistono diverse approcci all'incapsulamento:
- Anche linguaggi come il C offrono dei modi
- Abstract Data Types

Confronto con il C

- Vogliamo realizzare un contatore:
 1. un valore intero, variabile nel tempo
 2. tre operazioni (astrazione sulle operazioni)
 - reset() per impostare il contatore a zero
 - inc() per incrementare il valore attuale del contatore
 - getValue() per recuperare il valore attuale del contatore sotto forma di numero intero



Sol. 1 Come nuovo tipo (int) di C

- Dichiarazione (in `counter.h`)

```
typedef int contatore;  
void reset(contatore*);  
void inc(contatore*);  
int getValue(contatore);
```

- Uso

```
#include "counter.h"  
main() {  
    int v1, v2;  
    contatore c1, c2;  
    reset(&c1); reset(&c2);  
    inc(&c1); inc(&c1); inc(&c2);  
    v1 = getValue(c1); v2 = getValue(c2); }  
}
```

- Dichiarazione (in `counter.h`)

```
typedef int contatore;  
void reset(contatore*);  
void inc(contatore*);  
int getValue(contatore);
```

- Nota che
- `reset` e `inc` prendono come parametro formale un puntatore (**passaggio per riferimento**) perché devono modificare il contatore.
- `getValue` non modifica il contatore, allora passo il contatore **per valore**.
- Quando chiamo `reset` e `inc` devo usare l'operatore `&` :
- `inc(&c2);`

Definizione di contatore e delle operazioni

- Poi devo definire cosa fanno i metodi (in `counter.cpp`)

```
void reset(contatore* pc) {  
    *pc = 0;  
}  
  
void inc(contatore* pc) {  
    (*pc)++;  
}  
  
int getValue(contatore c) {  
    return c;  
}
```


Vantaggi e svantaggi del typedef

- Consente di separare interfaccia e implementazione
- Rende il cliente indipendente dalla struttura interna del tipo di dato (servitore)
 - **Esercizio**: proviamo a cambiare l'implementazione di contatore
- Permette al cliente di definire tanti contatori quanti gliene occorrono
- Ma non garantisce information hiding: tutti i clienti vedono la typedef, conoscono la struttura interna del contatore e possono violare il protocollo di accesso
 - **Esercizio**: proviamo a scrivere un cliente che incrementa di due

Sol2.- Contatore in C con un modulo

- Dichiarazione in un modulo (`mcounter.h`): contatore come singola risorsa protetta (`int`) dentro a un modulo

```
static int cont;
```

`static` → non va sullo stack

Ogni volta che importo il modulo ho una cella di memoria (`static`) per `cont`

- con operazioni che agiscono implicitamente su essa

```
void reset(void);
```

```
void inc(void);
```

```
int getValue(void);
```

Uso del contatore modulo

- si importano solo le dichiarazioni delle funzioni (mcounter.h) e si usa il contatore definito nel modulonon si definisce un contatore nel

main

```
#include "mcounter.h"
main() {
int v;
reset();
inc();
inc();
v = getValue();
}
```

Vantaggi e svantaggi del modulo

- Separa interfaccia e implementazione
- Rende il cliente indipendente dalla struttura interna del modulo (servitore)
- Garantisce l'incapsulamento
 - i clienti vedono solo le dichiarazioni delle operazioni: non conoscono la struttura interna della risorsa (privata) del modulo
- Offre al cliente una **singola** risorsa (da usare senza doverla definire): non è adatto se servono più risorse

Obiettivo

- poter nascondere i dettagli dell'implementazione (come con l'uso del modulo)
- garantire information hiding e incapsulamento
- permettere modifiche all'implementazione
- poter definire e utilizzare più contatori (con con il typedef)
- poter introdurre tanti contatori e fare le operazioni su di essi

Confronto con gli ADT

- Simile all'approccio tradizionale degli **abstract data types (ADT)**
- Vantaggi degli ADT
 - si può separare l'interfaccia dall'implementazione
- Svantaggi
 - vedi esempio
 - due tipi di figura geometrica: Quadrato e Rettangolo

Abstract Data Types

- is a specification of a set of data and the set of operations that can be performed on the data.
- it is independent of various concrete implementations
- The interface provides a constructor, which returns an abstract handle to new data, and
- several operations, which are functions accepting the abstract handle as an argument.

Esercizio

- Fai lo stesso con una automobile
 - getVelocità
 - ferma
 - accelera
 - rallenta

Abstract data types: Quadrato

ADT Quadrato with

mk_Quadrato : point * point -> Quadrato

area : Quadrato -> float

move : Quadrato * point -> Quadrato is

in

program

end

Rettangolo, simile a Quadrato

ADT Rettangolo with

mk_Rettangolo : point * point -> Rettangolo

area : Rettangolo -> float

move : Rettangolo * point -> Rettangolo is ...

in

program

end

Problemi con gli Abstract Data Types

- Non posso mischiare **Quadrato** con **Rettangolo**
 - anche se le operazioni sono uguali
 - se dichiaro una variabile devo sapere se è di un tipo o di un altro
- “riuso” limitato
 - non posso riusare un codice scritto per un ADT per un altro ADT
- Data abstraction è una parte importante dell'OO ma viene proposta in modo **estensibile**
 - mediante i meccanismi di ereditarietà e sottotipazione

Concetti dell'Object-Orientation

- incapsulamento - encapsulation
- **sottotipazione - subtyping**
 - per estendere i concetti
- **ereditarietà - inheritance**
 - per riusare le implementazioni
- binding dinamico - dynamic lookup

Sottotipazione ed Ereditarietà

- Interfaccia
 - La vista **esterna** di un oggetto (del cliente)
- **Sottotipazione**
 - Relazione tra interfacce
- ◆ Implementazione
 - La rappresentazione **interna** di un oggetto
- ◆ **Ereditarietà**
 - Relazione tra implementazioni

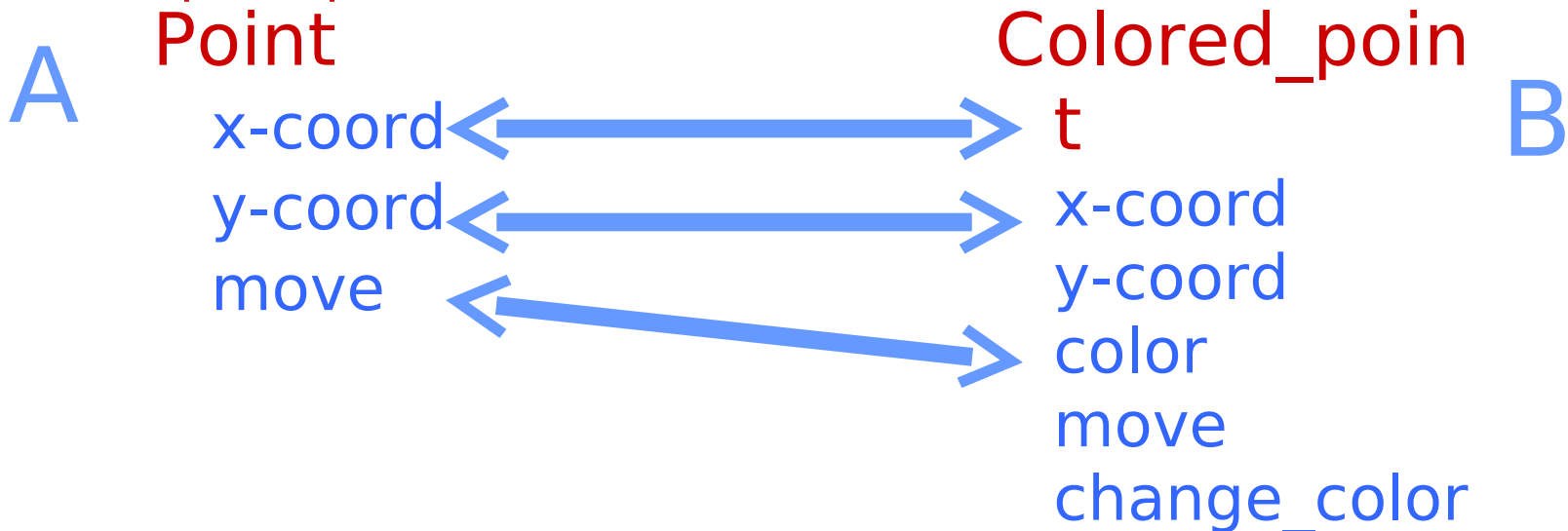
I due concetti sono strettamente legati ma distinti

Interfaccia di un oggetto

- Interfaccia
 - i **messaggi** che l'oggetto può ricevere
- Esempio: point
 - x-coord : returns x-coordinate of a point
 - y-coord : returns y-coordinate of a point
 - move : metodo per spostare un punto
- **L'interfaccia di un oggetto è il suo tipo**

Sottotipi

- Se un interfaccia **B** **contiene** l'interfaccia **A**, allora un oggetto **B** può essere usato al posto di un oggetto **A**
- **B** è un **sottotipo** di **A**
 - principio di sostituibilità



- ◆ L'interfaccia di **Colored_poin** contiene **Point**

Polimorfismo (di sottotipo)

- Se **B** è un **sottotipo** di **A**
dove c'è un termine di tipo **A** posso mettere un oggetto di tipo **B**
 - tutte le operazioni continueranno a funzionare
 - nella definizione di **variabili**
 - es. dichiaro var di tipo A: **A var;**
 - var potrebbe essere un oggetto di tipo B: **var = new B;**
 - es. dichiaro X di tipo Point: **Point X;**
 - X potrebbe essere un Colored_point
 - **X = new Colored_point;**
 - **variabili polimorfiche**

In java

```
class A{}
```

```
class B extends A {}
```

```
...
```

```
Object o = new A(); // A è sottotipo di Object
```

```
A h = new B();
```

```
B j = new B();
```

```
h = j;
```

```
int x = 0;
```

```
long l = x;
```

```
B k = new A();
```

```
B t = (B)h;
```

Ereditarietà - Inheritance

- Nuovi oggetti possono essere definiti **riusando** (anche parzialmente) implementazioni di altri oggetti
- Meccanismo relativo alle **implementazioni**
- Ad esempio una classe **B** (figlio) può ereditare definizioni (codice) di una classe **A** (padre) evitando duplicazione di codice
 - B riusa codice di A

Potenzialità dell'ereditarietà

```
class A { int function (int x) ... }
```

B eredità da A: `class B inherits A`

- B eredita il codice (membri: metodi e variabili) da A

- A può nascondere qualcosa a B (**private**)

- B può introdurre nuovi membri

```
class B { float foo (String x) ... }
```

- B può **ridefinire** alcuni membri di A

- *in genere* senza cambiare segnatura

```
class B {float function (float x) ... }
```

```
class B {int function (int x) ... }
```

non ridefinisce **fun**
di A

OK: ridefinisce **fun** di
A

- B potrebbe **nascondere** alcuni membri di A

Sottotipazione e Ereditarietà sono diverse

Esem

EREDITARIETÀ NON È SOTTOTIPAZIONE

```
class Point
```

```
private
```

```
float x, y
```

```
public
```

```
point move (float dx, float dy);
```

```
class Colored_point
```

```
private
```

```
float x, y; color c
```

```
public
```

```
point move(float dx, float dy);
```

```
point change_color(color newc);
```

◆ Subtyping

- Colored points possono essere usati al posto di points
- interessa il **cliente**

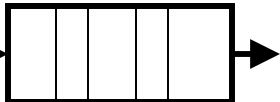
◆ Inheritance

- Colored points possono essere implementati usando l'implementazione di point
- **Interessa l'implementatore**


Esempio: ereditarietà \neq sottotipazione [Snyder]

- Ho le seguenti tre strutture dati


- Coda

- posso inserire e rimuovere un elemento 
- il primo elemento che inserisco che è il primo che tolgo (FIFO)

- Pila

- posso inserire e rimuovere un elemento 
- il primo elemento che tolgo è l'ultimo inserito (LIFO)

- Lista (già implementato)

- posso inserire in testa: **insert_at_head**
- posso inserire in coda: **insert_at_tail** 
- posso rimuovere dalla coda: **remove_at_tail**

Esempio: ereditarietà != sottotipazione

- implemento (ad esempio in C++) **Coda** e **Pila** **riutilizzando** l'implementazione di **Lista**:
 - Coda.**insert** = Lista.**insert_at_head**
 - Coda.**remove** = Lista.**remove_at_tail**
 - Pila.**insert** = Lista.**insert_at_tail**
 - Pila.**remove** = Lista.**remove_at_tail**
- e nascondo in Pila e Coda le operazioni **insert_at_X** della Lista
- Coda e Pila ereditano da Lista però non sono sottotipi: non posso più usare Pila al posto di Lista
 - anzi concettualmente Lista è un sottotipo di Pila e di Coda perchè contiene l'interfaccia, cioè le operazioni di Pila e Coda
 - sotto alcune condizioni "forti" ereditarietà e sottotipazione coincidono

Ereditarietà non è sottotipazione

- nei linguaggi OO **sottotipazione** e **ereditarietà** sono legate
 - in Java la sottotipazione è espressa mediante il meccanismo delle interfacce
 - **interface** A; B **implements** A: B è sottotipo di A ma non eredita nulla
 - in C++ subtyping ed ereditarietà pubblica coincidono
- Se si mettono vincoli sull'ereditarietà, possono coincidere
 - In Java posso ridefinire un metodo solo senza cambiare la segnatura -> sottoclasse è sottotipo
- sono però due concetti distinti
 - **sottotipazione** è riferito alle **interfacce**
 - **ereditarietà** è riferito alle **implementazioni**

Concetti dell'Object-Orientation

- incapsulamento - encapsulation
- sottotipazione - subtyping
 - per estendere i concetti
- ereditarietà - inheritance
 - per riusare le implementazioni
- **binding dinamico - dynamic lookup**
 - codice diverso per oggetti diversi

Binding Dinamico

- nell'approccio OO
 - object -> message (arguments)
 - il codice eseguito dipende da **object** e **message**
 - il tipo di object può variare runtime (grazie al polimorfismo)
- nei linguaggi di programmazione (tipo Pascal), ma anche con gli ADT
 - operation (operands)
 - il significato è sempre lo stesso

Esempio

- in OO `move` di un punto `x`
`x -> move (3,2)`

non mi preoccupo che `x` sia `Point` o `Colored_point`: viene deciso runtime

- in Pascal `move (x,3,2)`
so quando compilo quale `move` viene chiamata

Overload e binding dinamico

- spesso si confonde binding dinamico con l'overload di un metodo, però
- **overload**: un metodo o operazione con lo stesso nome si applica a diversi tipi
 - esempio: **+** va bene per interi e float
- L'overloading viene risolto al tempo di **compilazione**
 - esempio a + 2
 - 2.0 + 3.0 : viene utilizzato il + dei float

Single dispatch

- $x \rightarrow \text{message}(y)$
il codice eseguito dipende runtime da x non da y

Si dice “single dispatch”

STATE ATTENTI, vedi esempio

Single dispatch 2 - Java

Object definisce un metodo **equals** con par. **Object**
class **Object** { boolean **equals** (**Object** o) ... }

A eredita **Obj** e definisce **equals** con parametro **A**
class **A** extends **Object** { boolean **equals** (**A** a) }

A non ridefinisce il metodo **equals** di **Object** !!!

Creo due oggetti **A**

```
Object a1 = new A();   Object a2 = new A();  
a1.equals(a2); // quale equals è eseguito?
```

Se voglio essere sicuro di usare **equals** di **A** devo ridefinire **equals**:

```
class A extends Object { boolean equals (Object a) }
```

Esercizio

- Scrivi una classe A con un membro intero x.
- Con costruttore con un intero da assegnare a x
- Aggiungi il metodo boolean equals(A a) che restituisce true se a.x è uguale a this.x
- Cosa succede se fai
- Object a1 = new A(1), a2 = new A(1)
- A a3 = new A(1);
- System.out.println(a1.equals(a1));
- System.out.println(a1.equals(a2));
- System.out.println(a3.equals(a1));
- System.out.println(a1.equals(a3));

Altro esercizio

- `class A { foo(A a){...} }`
- `class B extends A { foo(B b){...} }`
- `class C extends A { foo(A a){...} }`

`A x = new A();`

`B y = new B(); A z = new B();`

`C w = new C(); A v = new C();`

`x.foo(x); x.foo(y); ...`

`y.foo(x); y.foo(y); z.foo(x); z.foo(y);`

`w.foo(x); w.foo(w); v.foo(x); v.foo(w);`

OO in pratica

- Esercizio


Esercizio: libreria geometrica

- Definisco il concetto generale `Figura`
- Implemento due forme: `Cerchio`, `Rettangolo`
- Implemento le seguenti funzioni
`center`, `move`, `rotate`, `print`, `equals`
- Come estendere la libreria?
 - Aggiungo `Quadrato` come estensione di `Rettangolo`
- Prova a implementarlo nel tuo linguaggio OO preferito !

OO Program Structure

- Group data and functions
- Class
 - Defines behavior of all objects that are instances of the class
- Subtyping
 - Place similar data in related classes
- Inheritance
 - Avoid reimplementing functions that are already defined

Code placed in classes



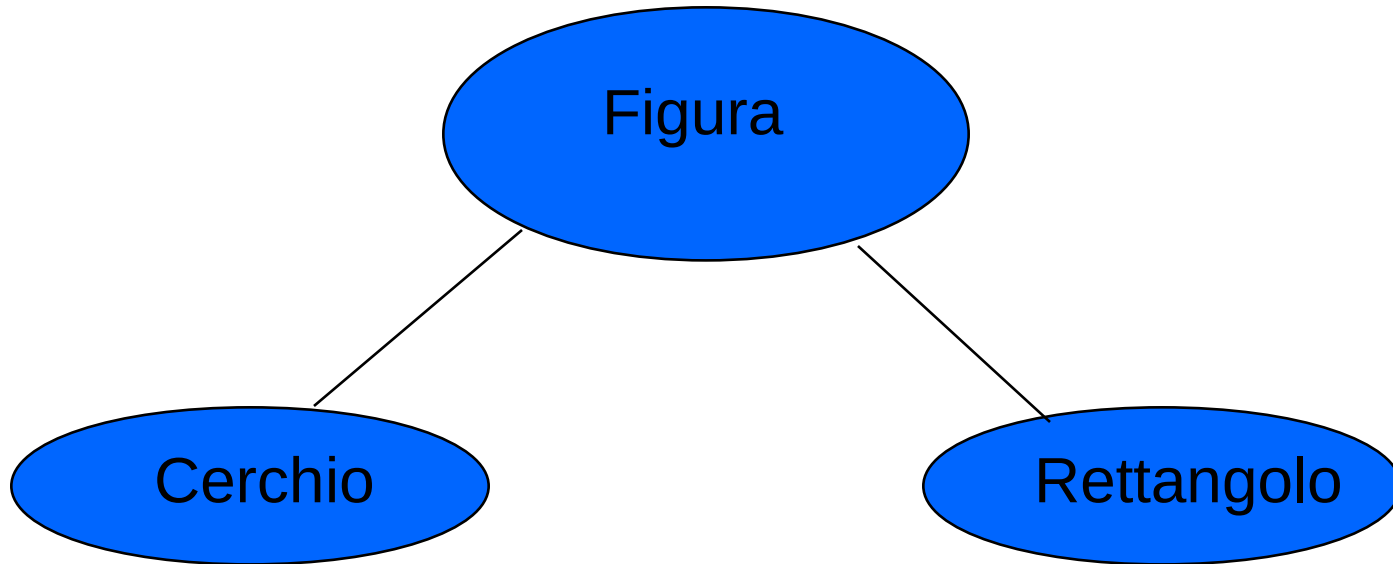
	center	move	rotate	print
Circle	c_cente r	c_move	c_rotate	c_print
Rectangl e	r_center	r_move	r_rotate	r_print

- Dynamic lookup
 - circle → move(x,y) calls function c_move
- Conventional organization
 - Place c_move, r_move in move function

Figura

- L'interfaccia di ogni **Figura** include
`center`, `move`, `rotate`, `print`, `equals`
- Diversi tipi di Figura hanno implementazioni diverse
 - **Rettangolo**: i quattro vertici
 - **Cerchio**: centro e raggio

Sottotipi



- L'interfaccia generale è definita in **Figura**
- Implementazioni sono definite in **Cerchio, Rettangolo**
- Si aggiungono facilmente nuove forme

Sommario

1. Cenni di progettazione Object-oriented
 2. Concetti principali dell'object-orientation
 - incapsulamento
 - sottotipo
 - ereditarietà
 - binding dinamico
- ◆ Prossime lezioni
- Confronto tra i diversi linguaggi, come supportano l'OO: C++, Java, ...