

# Introduzione all'Object-Orientation

---

Angelo Gargantini

# 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)

OGGETTO	
dati nascosti	
msg <sub>1</sub>	method <sub>1</sub>
...	...
msg <sub>n</sub>	method <sub>n</sub>

**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

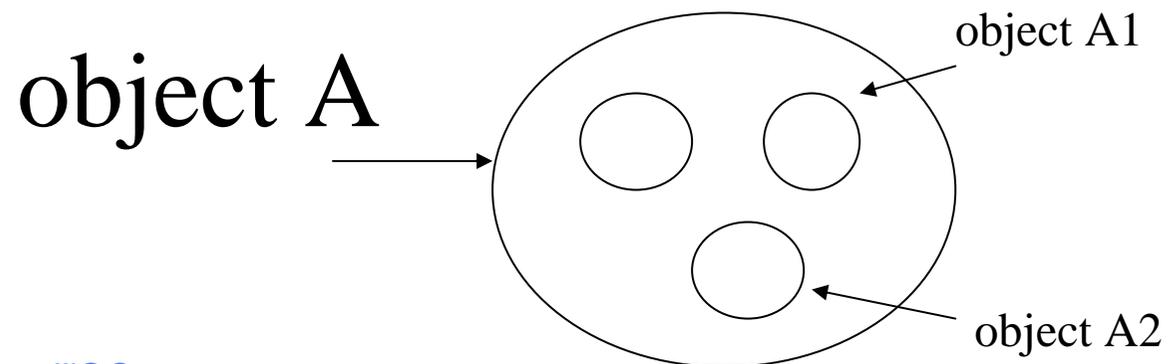
## ◆ Non necessariamente "top-down"

- "livello d'astrazione" a qualsiasi livello

# Progettazione OO

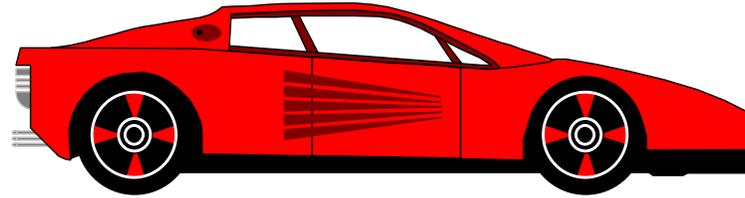
---

- ◆ 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

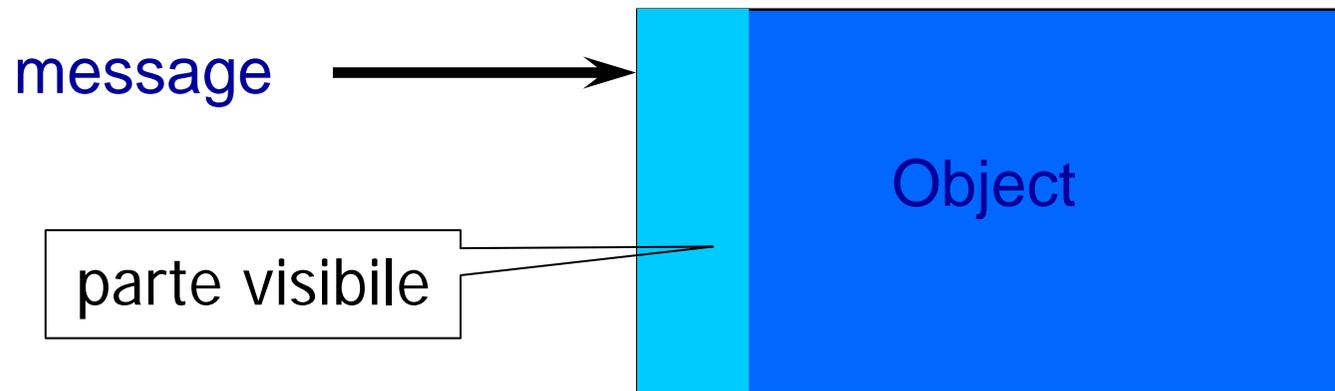
---

- **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



# 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: Quadrato

---

ADT Quadrato with

`mk_Quadrato : unit * point * point -> Quadrato`

`area : Quadrato -> float`

`move : Quadrato * point -> Quadrato`

is ...

in

*program*

end

# Rettangolo, simile a Quadrato

---

ADT Rettangolo with

`mk_Rettangolo : unit * 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

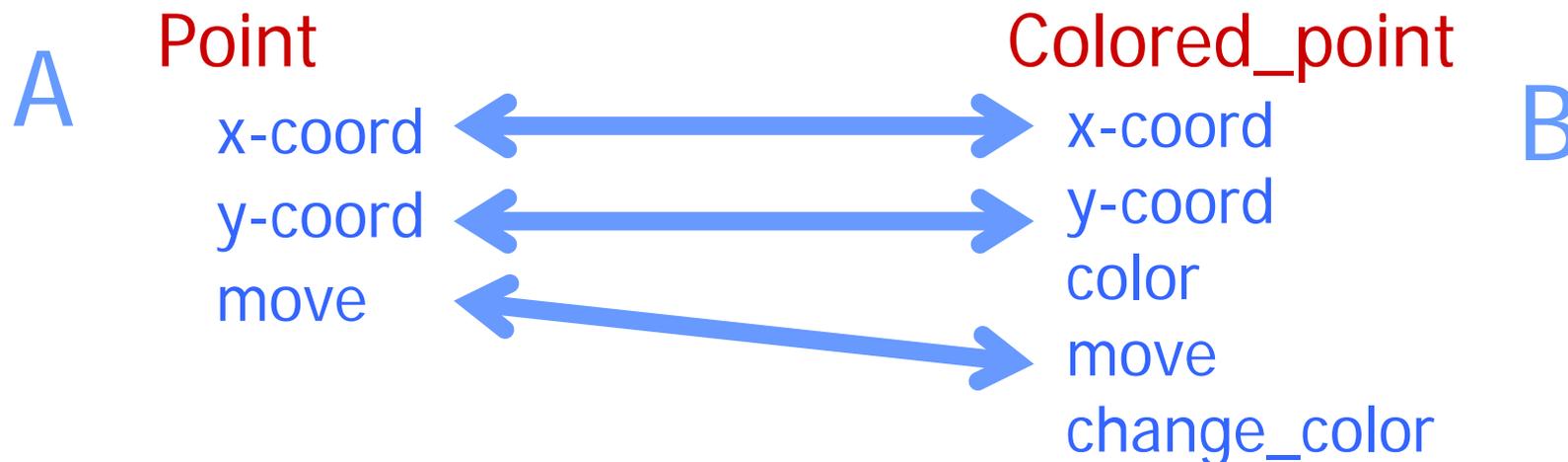
## ◆ Esempio: point

- 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\_point** contiene **Point**
  - **Colored\_point** è un sottotipo di **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**

# 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) ... }
```

non ridefinisce **fun** di A

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

OK: ridefinisce **fun** di A

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

# Esempio

---

```
class Point
```

```
private
```

```
float
```

```
public
```

```
point
```

```
class Color
```

```
private
```

```
float
```

```
public
```

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

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

## ◆ Subtyping

- Colored points possono posto di

Sottotipazione e Ereditarietà sono diverse

**EREDITARIETÀ NON È SOTTOTIPAZIONE**

nte

possono

entati

mentazione

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** e **remove** 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
- ◆ 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**

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

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

# 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?
  
- ◆ Prova a implementarlo nel tuo linguaggio OO preferito !

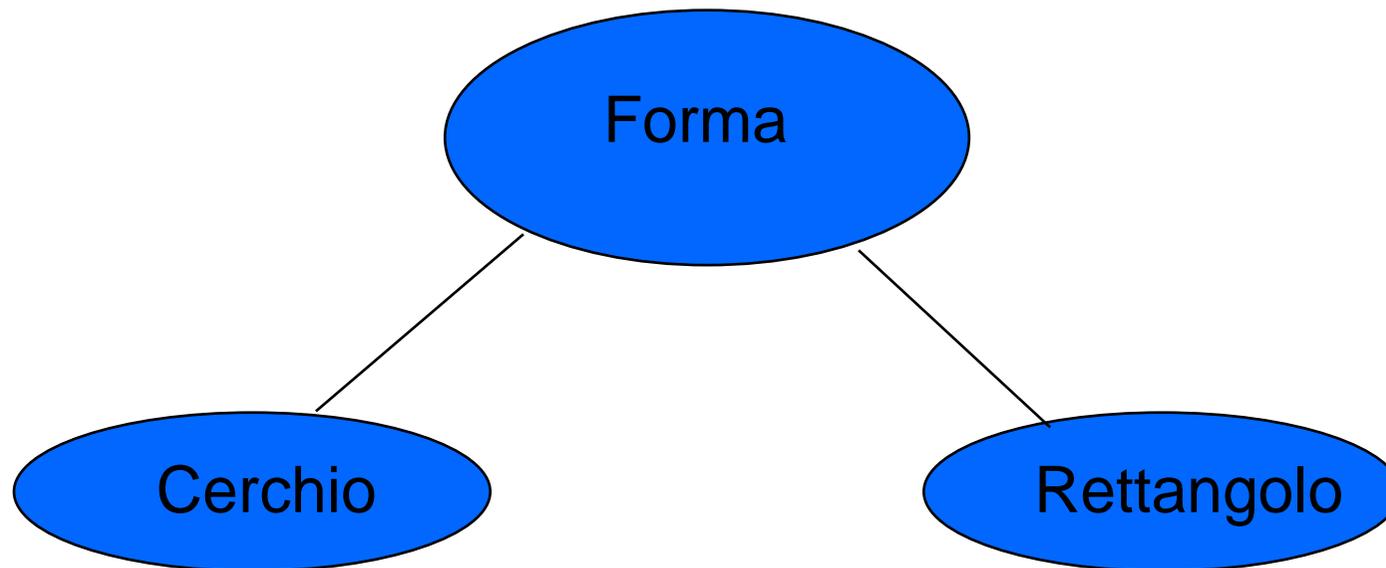
# Forme

---

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

# Sottotipi

---



- ◆ L'interfaccia generale è definita in **Forma**
- ◆ 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](#)
- ◆ vedi altre slide su java