

# Programmazione a Oggetti

## Modulo B

Dott. Alessandro Roncato

**3/02/2014**

# Esame

Modulo A +

Scritto oppure 2  
compitini

Orale

(progetto facoltativo)

# Come iniziare

Nel Modulo B vedremo degli “esempi” (Design Patterns) che aiuteranno a progettare il codice seguendo i paradigmi della programmazione a oggetti.

Inizieremo oggi cercando di capire come dividere il codice nei vari oggetti.

IMPORTANTE: prima di scrivere codice  
bisogna capire cosa e come scriverlo

# Programmazione a oggetti

In questo corso ci concentreremo di più su **dove mettere il codice** piuttosto che su come scrivere il codice.

Vedremo come spostando il codice si possono ottenere vantaggi (spostare == Cut&Paste).

Assumiamo che gli studenti sappiano programmare, ovvero che istruzioni usare e come usarle per risolvere un problema.

# Esempio

Cominciamo con un esempio: una semplice applicazione per gestire una banca.

Supponiamo di prevedere la gestione dei clienti e dei conti correnti dei clienti.

Supponiamo di memorizzare nome del cliente, numero conto e saldo. Le operazioni possibili sono la creazione di un nuovo conto, il versamento di una somma su un conto, il prelievo di una somma da un conto, il calcolo degli interessi e la chiusura di un conto.

# Java non OO

```
public class Banca {  
    Array<String> nomeCliente = new  
        Array<String>();  
    Array<Integer> numeroConto = new  
        Array<Integer>();  
    Array<Double> saldoConto = new  
        Array<Double>();  
    int prossimoNumeroConto = 1;  
  
    public int apriConto(String nome)  
    {    int numero = prossimoNumeroConto++;  
        nomeCliente.add(nome);  
        numeroConto.add(numero);  
        saldoConto.add(0.0);  
        return numero;  
    }  
}
```

*\* Array => ArrayList (per brevità)*

# Java non OO cont

```
public void versa(int numero, double val)
{   int i = numeroConto.indexOf(numero);
    double saldo = saldoConto.get(i);
    saldo+=val;
    saldoConto.set(i,saldo);
}

public double preleva(int numero, double val)
{   int i = numeroConto.indexOf(numero);
    double saldo = saldoConto.get(i);
    double prelievo = Math.min(saldo,val);
    saldo-=prelievo;
    saldoConto.set(i,saldo);
    return prelievo;
}
```

# Java non OO cont

```
public void accreditaInteressi()
{
    for (int i=0; i<saldoConto.size(); i++)
    {
        double saldo = saldoConto.get(i);
        saldo*=(1.0+0.03);
        saldoConto.set(i,saldo);
    }
}

public double chiudiConto(int numero)
{
    int i = numeroConto.indexOf(numero);
    double saldo = saldoConto.get(i);
    numeroConto.remove(i);
    nomeCliente.remove(i);
    saldoConto.remove(i);
    return saldo;
}
```



# Svantaggi non OO

- 1) Mescolati concetti
- 2) Difficile da usare, debuggare e mantenere:  
esempio se volessimo aggiungere la possibilità  
di gestire un interesse diverso per ogni conto  
corrente
- 3) Arbitrarietà nella divisione del codice
- 4) Difficile riuso del codice

# Banca non OO estesa

```
public class Banca {  
    Array<String> nomeCliente = new  
        Array<String>();  
    Array<Integer> numeroConto = new  
        Array<Integer>();  
    Array<Double> saldoConto = new  
        Array<Double>();  
    Array<Double> tasso = new  
        Array<Double>();  
    ...  
    public void accreditaInteressi()  
    {  
        for (int i=0; i<saldoConto.size();  
            i++){  
            double saldo = saldoConto.get(i);  
            saldo*=(1.0+tasso.get(i));  
            saldoConto.set(i,saldo);  
        }  
    }  
}
```

Che bug è stato introdotto?

# Versione 00

- 1) Raggruppa concetti
- 2) Facile da usare, debuggare e mantenere:  
esempio se volessimo aggiungere la possibilità  
di gestire un interesse diverso per ogni conto  
corrente
- 3) Divisione del codice del codice naturale
- 4) Riutilizzo del codice

# Banca OO

```
public class Banca {  
    Array<Conto> conti = new Array<Conto>();  
    int prossimoNumero = 1;  
    public int apriConto(String nome){  
        int numero = prossimoNumero++;  
        conti.add(new Conto(nome,numero));  
        return numero;  
    }  
    public Conto trovaConto(int numero){  
        for (Conto c: conti)  
            if (c.getNumero()==numero)  
                return c;  
        return null;  
    }  
}
```

# Banca OO cont.

```
...
public void
accreditaInteressi() {
    for (Conto c: conti)
        c.accreditaInteressi();
}
public double chiudiConto(int
numero) {
    Conto c =
trovaConto(numero);
    conti.remove(c);
    return c.getSaldo();
}
}
```

# Banca OO Conto

```
public class Conto{
    String nome;
    int numero;
    double saldo=0.0;
    public Conto(String nome, int
numero){
        this.nome=nome;
        this.numero=numero;
    }
    public void getNumero() {
        return numero;
    }
    public void getSaldo(){
        return saldo;
    }
}
```

# Banca OO Conto cont.

```
...  
    public void versa(double val){  
        saldo+=val;  
    }  
    public double preleva(double val){  
        double prelievo =  
Math.min(saldo,val);  
        saldo-= prelievo;  
        return prelievo;  
    }  
    public void accreditaInteressi(){  
        saldo*=(1.0+0.03);  
    }  
}
```

# Vantaggi OO

- 1) riuso: riduzione costi, riduzione complessità
  - 2) manutenzione: debug, aggiornamenti, personalizzazioni, estensioni.
  - 3) controllo dipendenze (limitare le modifiche)
  - 4) dividere il lavoro tra più persone (anche con skills diversi)
  - 5) applicazioni grandi
- Differenza fra approccio fai da te e approccio Professionale.



# Difficoltà OO

- 1) Come dividere variabili e codice fra le classi  
(Come assegnare le responsabilità)
  - 2) Come interagiscono le classi per risolvere il problema (Nel caso non OO ho tutto nello stesso file, nel caso OO ho il codice separato tra più file)
  - 3) Trade-off tra tanti metodi e pochi
  - 4) Tante Classi
- Necessità di progettare l'applicazione

# A cosa serve la progettazione?

Nei casi reali prima di realizzare un qualsiasi manufatto è buona norma fare un progetto.

Il progetto serve a:

- 1) idea del risultato finale (+ modello in scala)
- 2) stimare tempi e costi
- 3) controllare che le varie componenti siano compatibili tra di loro
- 4) dividere il lavoro tra più persone
- 5) etc.

# Progettazione di un'applicazione

Prima di realizzare una qualsiasi **applicazione** è buona norma fare un progetto.

Il progetto serve a:

- 1) idea del risultato finale (+ prototipo GUI)
- 2) stimare tempi e costi
- 3) controllare che le varie componenti siano compatibili tra di loro
- 4) dividere il lavoro tra più persone
- 5) etc.

# Quanti oggetti?

Se intendiamo gestire tutta la nostra applicazione con un unico oggetto, il risultato sarà molto simile all'approccio non orientato agli oggetti.

Un unico “spazio” e tutto il codice e i dati presenti in questo spazio saranno strettamente dipendenti.

# Svantaggi unico oggetto

Non ho nessun vantaggio PO

Difficile riuso

Interdipendenza codice altissima

Difficile debug

Difficile manutenzione

Difficile da suddividere

l'implementazione con più persone

**NON SCRIVERE CODICE E DIVIDERE POI!**

# Quanti oggetti?

Altra soluzione: un oggetto Java per ogni oggetto “reale”:

- Un oggetto Banca

- Un oggetto Cliente per ogni cliente della banca

- Un oggetto Conto per ogni contocorrente

- Un oggetto Prelievo per ogni prelievo etc.

ELENCARE TUTTI GLI OGGETTI

# Vantaggi

Facilmente divisibili tra i vari sviluppatori

Ogni singolo “componente” è facilmente debuggabile,  
riusabile e mantenibile

Lo “spazio” di dipendenza “interna” del codice è ridotto in  
quanto il codice è diviso in più parti

C'è però interdipendenza “esterna” anche tra gli spazi:  
come limitarla?

# Quanti oggetti?

Visto che passando da un oggetto a tanti, abbiamo ottenuto dei vantaggi, perché non spingersi oltre e suddividere ulteriormente gli oggetti in modo da avere più oggetti Java di quelli del mondo reale?

Aumenta la dipendenza esterna tra oggetti

È più difficile da suddividere il lavoro tra programmatori



# Quante classi?

Tipi diversi di oggetti hanno bisogno di classi diverse, quindi il numero di classi corrisponde al numero di oggetti diversi (questo vale per tutti i linguaggi a oggetti fortemente tipati).

Delle volte non è così semplice capire se due oggetti hanno bisogno di due classi diverse oppure se possono essere gestiti dalla stessa classe

# Che nome dare alle classi?

Il nome delle classi dovrebbe essere un nome (non verbi o aggettivi).

OK: Banca, Conto, Cliente

KO: Pagare, Velocemente

# Cos'è la dipendenza esterna? (\*)

C'è dipendenza esterna quando una classe “ha a che fare” con un'altra classe, cioè:

- 1) Una classe estende un'altra classe
- 2) Gli oggetti di una classe hanno una relazione con oggetti di un'altra
- 3) Un metodo di una classe usa un oggetto di un'altra classe

# Esempio di dipendenza

Estensione: la classe Prelievo estende la classe

Operazione (protected)

Relazione: un oggetto Conto è relativo a un Cliente

(package, public)

Uso: il metodo `accreditaInteressi` dell'oggetto

Banca accede all'oggetto Conto per invocare il  
metodo `accreditaInteressi` dell'oggetto  
stesso. (package, public)

# Perché ridurre dipendenza?

Una modifica (per debug, manutenzione, riuso etc.) ad una classe implica una possibile modifica alle classi dipendenti

Quindi se due classi NON sono dipendenti, ho libertà di modificarle senza problemi

# Come ridurre la dipendenza

Riducendo la parte “visibile” dalle classi.

L'incapsulamento dei linguaggi ad oggetti permette al compilatore di controllare la dipendenza tra classi

La parte visibile è costituita dagli attributi pubblici

# Incapsulamento

Incapsulamento codice =  
funzioni e procedure

Incapsulamento dati =  
strutture (es. `struct` del C)

Incapsulamento di codice  
(metodi) + dati (variabili) +  
controllo visibilità = Object  
Oriented

# A cosa serve?

L'incapsulamento permette di controllare le dipendenze esterne rendendo **invisibili** alcune parti (che chiameremo interne o private)

Oltre che al controllo della dipendenza, permette anche di evitare l'accesso non controllato rendendo più “affidabile” l'uso delle classi stesse

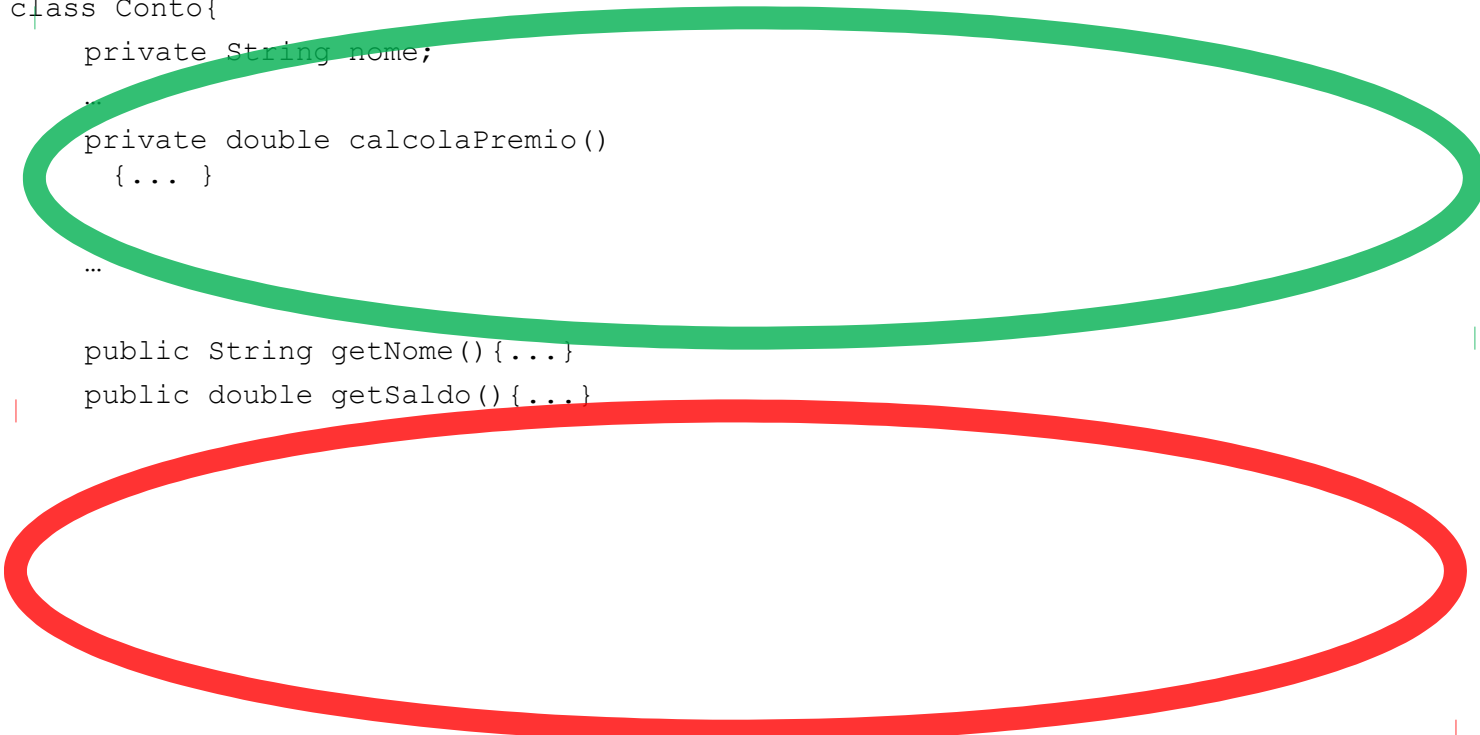


# Analogia con il mondo reale

Gli oggetti di uso più comune (lavatrice, radio etc.) vengono incapsulati per rendere inaccessibili la parte interna. L'involucro permette di aumentare la semplicità d'uso dell'oggetto stesso e ne impedisce un uso improprio.

# In pratica?

```
public class Conto{  
    private String nome;  
    ...  
    private double calcolaPremio()  
        {... }  
    ...  
    public String getNome(){...}  
    public double getSaldo(){...}
```



Dipendenza

# Java Bean

Convenzione Java:

1) non ci sono campi (variabili d'istanza)  
pubblici;

```
public Type getNome()
```

2.1) proprietà in *lettura* “nome” di tipo  
Type

```
public void setNome(Type a)
```

2.2) proprietà in *scrittura* “nome” di  
tipo Type

(se il tipo è boolean get diventa is)

```
public Oggetto ()
```

3) deve essere definito il costruttore di  
default

# Ridurre dipendenza

```
public class Conto{  
    private String nome;  
    private double saldo;  
    public String getNome()  
    {return nome;}  
    public void setNome(String  
    n) {nome=n;}  
    public void  
    accreditaInteressi ()  
        {saldo*=(1.0+0.03);}  
    ...  
}
```

# Come riduce dipendenza?

Successivamente si vuole gestire gli interessi in modo che ogni conto corrente possa avere assegnato delle condizioni migliori e/o peggiori (e non unico per tutta la Banca).

Per questa ragione viene aggiunto al Conto anche l'interesse applicato.

# Come ridurre dipendenza?

```
public class Conto{  
    private String nome;  
    private double saldo;  
    private double interesse;  
    public String getNome(){return nome;}  
    public void setNome(String n) {nome=n;}  
    public void calcolaInteressi ()  
        {saldo*=(1.0+interesse);}  
    public void setInteresse(double i){  
        this.interesse=i;}  
}
```

# Come riduce dipendenza?

Anche se l'implementazione della classe è cambiata (aggiunta di un attributo e modifica di un metodo), dato che la parte pubblica (interfaccia) non è cambiata, le altre classi non vengono influenzate da questa modifica.

In realtà la parte pubblica è cambiata perché è stato aggiunto anche il metodo pubblico **setInteresse**. Ma l'aggiunta di metodi alle classi non crea problemi al codice esistente: tutti i metodi usati da altre classi continuano a essere presenti.

Regola: partire con pochi metodi, eventualmente aggiungere poi.

# Tipi di relazioni

Estensione (IS-A)

Associazioni

Dipendenze (uso di  
attributi)



# UML

Uml è un linguaggio visuale semiformale che  
permette di descrivere vari aspetti del progetto  
di un'applicazione

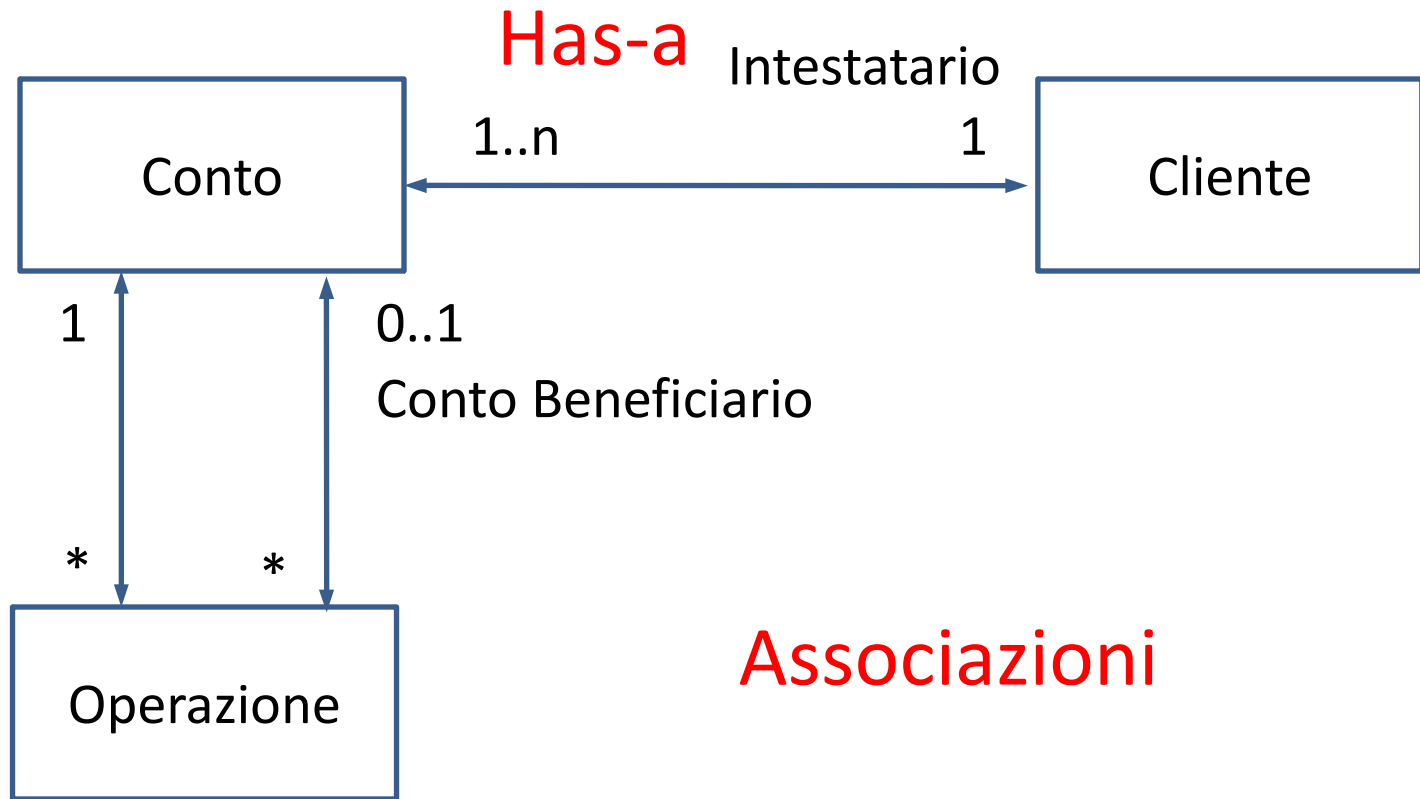
Permette anche di “abbozzare” delle soluzioni per  
gestire le fasi iniziali della progettazione

Useremo per fare una “bozza” del progetto

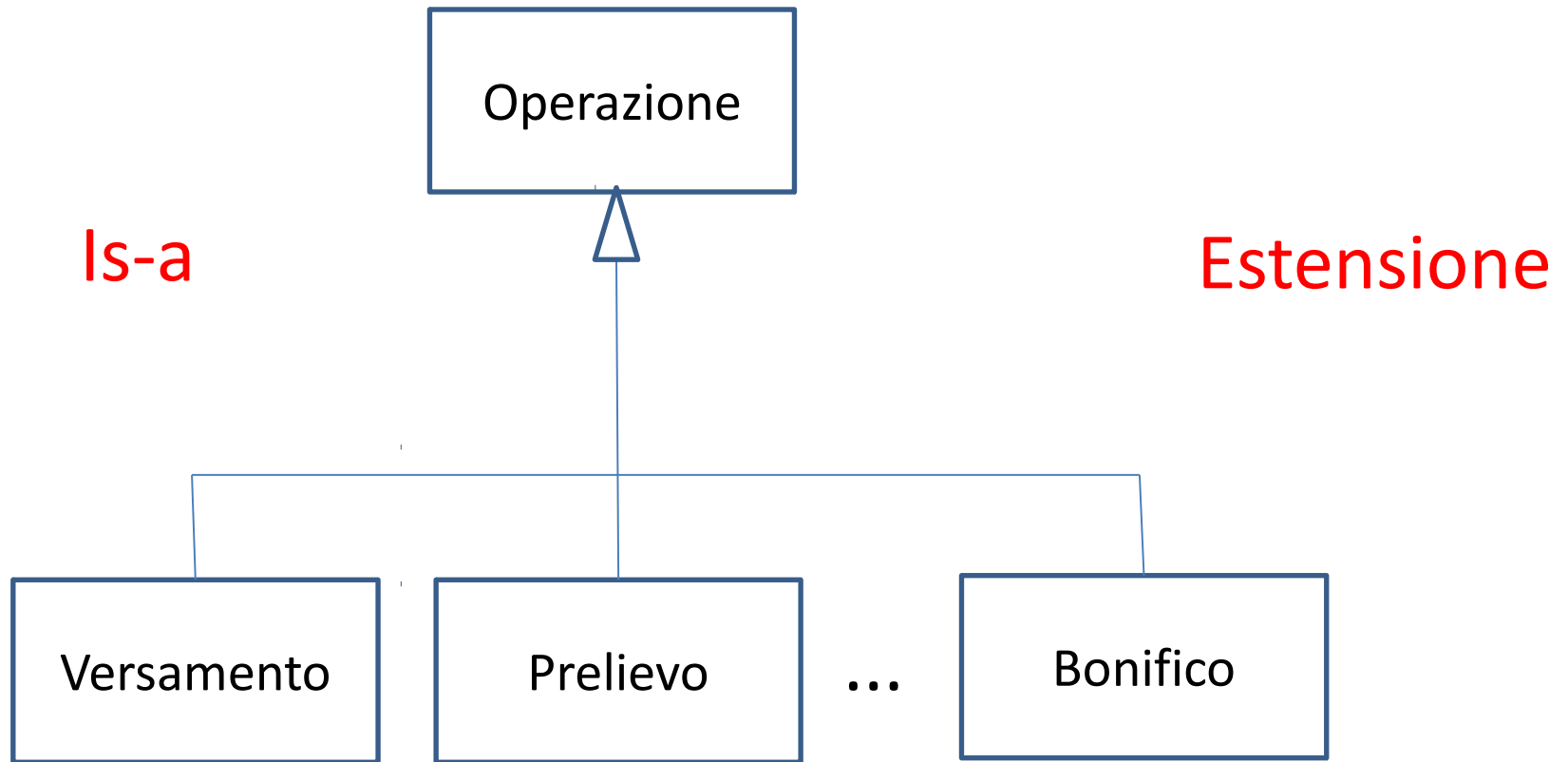
Diagramma delle classi visualizza le classi, le  
estensioni e le associazioni

Vista statica del progetto

# Diagramma UML classi (parziale)

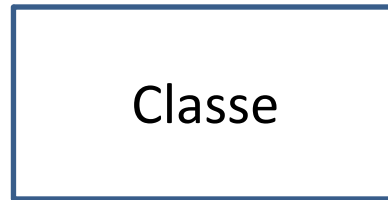


# Diagramma UML classi (parziale)



# Diagramma UML classi

(notazione principale)



0..1

molteplicità

Associazione

ruolo



Is-a

Estensione



Dipendenza

# Che campi nelle classi?

I campi (o variabili d'istanza) delle classi devono permettere di rappresentare l'oggetto stesso

Esempio: per rappresentare un Conto servono l'intestatario del conto, il saldo, etc.

Java Bean, i campi sono privati, quindi sono un dettaglio implementativo, si stabiliscono quali sono le proprietà delle classi

# Che metodi nelle classi

I metodi devono permettere di usare in maniera semplice  
l'oggetto stesso;

Ogni metodo (che non sia `get` e `set`) dovrebbe avere il  
nome di un verbo ed eventualmente un complemento:

Esempio: `stampa`, `creaOperazione`, `accreditaInteresse`, ...

Il verbo è imperativo o passivo (l'oggetto esegue il  
comando o subisce l'azione)

# Esempio

- In Italiano (linguaggio naturale)
- Il cassiere della banca (Soggetto) immette un prelievo su un conto corrente (Oggetto).
- In Java (linguaggio a oggetti)
- La classe Conto ha un metodo preleva.
- L'interfaccia grafica invoca il metodo preleva dell'oggetto Conto.

# Che metodi nelle classi

Generalmente non è bene che un metodo abbia  
il nome composto da due verbi: es.

CalcolaEStampa

Significa che probabilmente fa 2 cose distinte,  
meglio invece avere 2 metodi distinti: il  
metodo Calcola e il metodo Stampa

(Vedremo casi in cui ha senso avere 2 verbi)



# Che metodi nelle classi

Semplici da usare:

- Nome individua chiaramente la funzione

- Pochi parametri: quelli indispensabili per dare senso al metodo

- Altri parametri possono essere “impostati” grazie lo stato dell'oggetto:

  - Invece che avere `o.stampa(testo, position, font, size, color, shadow, threeDim, rotate)`

  - Meglio un metodo `o.stampa(testo, position)` e gli altri parametri impostabili con altri metodi tipo `o.setFont(font)` da invocare prima di `o.stampa`

# Che metodi nelle classi

Avere pochi metodi pubblici aumenta  
l'indipendenza

Ma quali sono i metodi che sono  
necessari e pubblici?

Diagrammi di sequenza permettono  
di “trovare” i metodi necessari

# Domande?