

Object Orientation

Lezione del 29-09

Come contattare il prof:

- Per quesiti brevi: sergio.dimartino@unina.it
- oggetto ->[OO] e poi l'oggetto
- FIRMARE SEMPRE LA MAIL

Ricevimento -> mercoledì 10-12

Esame:

- Progetto (4-5 trace in comune con Basi) + scritto (esercizi + domande aperte)
 - progetto di gruppo (2 persone che fanno progettazione e implementazione di un piccolo sistema SW)
 - presentazione di gruppo con documentazione + demo max 20 min

Evoluzione del paradigma procedurale-> introduzione del concetto di classe (definisce tipi complessi personalizzati)

Lo scopo è quello di migliorare l'efficienza del processo di produzione e mantenimento del SW

Aggiunta di nuove keyword-> tipo rappresentato dal termine classe

Lezione del 01-05

Dall'approccio procedurale passiamo ad un'approccio bottom-up (OO)

In un progetto di questo tipo vado a creare un oggetto, ognuno dei quali contiene dei dati e le funzionalità che posso fare su questi ultimi

Ciò comporta che il main resti succinto

es. conto corrente

DATI	AZIONI
IBAN	VERSARE
Dati Intestatario	Prelevare
Saldo	Effettuare bonifico

Il C a differenza di OO non permette in alcun modo di proteggersi da attacchi malevoli sul codice

Con l'OO creo un nuovo tipo che contiene i dati da proteggere e le operazioni che definisco:

```
class(keyword) ContoCorrente(nome della classe){  
//ATTRIBUTI DELLA CLASSE (DATI)
```

```

string IBAN;
string INTESTATARO;
double Saldo;

//METODI (PROCEDURE DI ACCESSO AI DATI)

bool preleva(double Importo){
...
}

void versa(double Importo){
...
}

double LeggiSaldo(){
...
}

```

Differenza tra **classi** e **oggetti**:

- classe -> qualcosa di astratto (come è fatto questo tipo?)
- Oggetti -> variabili/istanza della classe

In Java non esistono puntatori esplicativi -> si risolve il problema a grandi linee del segmentatio fault

Con l'operatore . posso accedere ai metodi delle classi

Con Java abbiamo la possibilità di rappresentare nel miglior modo possibile un problema reale

Per le chiamate ai METODI prima si indica su quale oggetto voglio applicare una modifica, e poi la modifica che voglio applicare

Es.

prelevare qualcosa dal conto corrente c_{1} -> c_{1}.preleva(500)

Lezione del 06-10

Introduzione a java e differenze rispetto ad altri linguaggi di programmazione

Portabilità -> presente in Java, mancante nel C

Inizio anni '90 -> si immaginava un'ondata di dispositivi smart

File java .java -> passa da un compilatore -> produce un file chiamato bytecode -> java virtual machine (JVM) -> assembly

In questo modo al netto di una prestazione lievemente minore, ho una portabilità pressochè totale

In questo modo infatti riesco a scrivere un linguaggio che può girare su ogni tipo di os

In console ->

1. Creo un file con estensione .java

2. Con javac *nome file* invoco il compilatore che mi produce un bytecode con estensione .class

3. Con java *nome file* senza estensione lo eseguo

Java API -> insieme delle classi predefinite che costituiscono la base per lo sviluppo dei programmi

Lezione del 08-10

Cosa succede quando provo a compilare qualcosa in java-> dal byte code (.class)

Nel caso in cui il nostro programma provi a uscire dall'area di memoria che gli è stata data, è stesso java a limitarlo -> viene permesso al programmatore cosa fare (in quanto il programma è all'interno di un'ambiente che è quello di java)

Primi esempi:

```
public class ContoCorrente{  
    String IBAN;  
    String Intestatario;  
    double saldo;  
  
    public void Versa(double amount) {  
        Saldo = Saldo + amount;  
    }  
    public void Preleva(double amount) {  
        if (amount <= Saldo){  
            Saldo = Saldo -amount;  
            return true  
        }  
        else  
            return false  
    }  
  
    public void SetIBAN(String ib) {  
        Iban=ib;  
    }  
    public void SetIntestatario(String nome){  
    }  
    public void StampaDettagliConto(){  
        System.out.println("Il conto di "+Intestatario+" con IBAN" +IBAN+  
ha saldo "+Saldo+" £" ) //si usa la key + per concatenare le stringe  
    }  
}
```

keyword **new**

- malloc sizeof() di qualcosa (in C)
- crea in memoria uno spazio per contenere tutto lo spazio che ci serve per una classe

Una stessa classe grazie all'OO può essere usato più volte -> operazioni che vanno ad operare ognuno sul proprio oggetto

Creazione di un costruttore

Cos'è un ambiente di sviluppo integrato:

Si sceglie una directory come workspace -> creazione di un nuovo progetto

Utilizzo dei breakpoint -> interruttori comodi per la fase di debug-> cliccando 2 volte al lato sulla riga

Lezione del 13-10

A differenza del C java inizializza gli attributi di una classe ma non una variabili in locale (impostando il valore a null per le stringhe, a 0 per gli int e così via)

Non è possibile leggere valori "sporchi"

Keyword **private** e **public**

Tutto ciò che è definito con public può essere utilizzato da tutti, con private no

```
public class ContoCorrente{  
    //ATTRIBUTI  
    public String IBAN;  
    public String Intestatario;  
    public double saldo;  
  
    public void Versa(double amount) {  
        Saldo = Saldo + amount;  
    }  
    public void Preleva(double amount) {  
        if (amount <= Saldo){  
            Saldo = Saldo -amount;  
            return true  
        }  
        else  
            return false  
    }  
  
    public void SetIBAN(String ib) {  
        Iban=ib;  
    }  
    public void SetIntestatario(String nome){
```

```

    }
    public void StampaDettagliConto(){
        System.out.println("Il conto di "+Intestatario+" con IBAN" +IBAN+
ha saldo "+Saldo+" £" ) //si usa la key + per concatenare le stringe
    }
}

```

Sostituendo a public private -> limito le modifiche tutelandomi

Gli attributi vanno sempre definiti private per comodità, nei metodi sta a noi la scelta

Questa proprietà prende il nome di *incapsulamento*

#+Lezione del 15-10

Distinzione di tipi primitivi e oggetti

Tipi:

- byte, int, long ⇒ intero;
- float, double ⇒ virgola;
- boolean, char

Nei tipi primitivi in java non è presente il tipo **puntatore** (li nasconde al programmatore-> non si preoccupa dell'allocazione di memoria)

Uno delle criticità del C è la memory leakage -> causata da una mancata free function dopo un malloc

In Java invece questa operazione è gestita automaticamente (non richiede nessuna operazione esplicita da parte del programmatore)

Nel momento in cui uno spazio non è più puntato, quest ultimo viene flaggato in modo che periodicamente parta un *garbage collector* che libera la memoria

In questo modo in java non c'è memory leakage

Il rovescio della medaglia è che per la presenza del garbage collector impedisce a java di essere usato per programmi real-time

Per allocare un nuovo oggetto in java -> keyword **new**

Per allocare un tipo primitivo basta l'assegnazione. Le principali differenze sono:

- ==
 - con i tipi primitivi **confronta i valori**
 - con gli oggetti invece i **puntatori**
- Allocazione
 - Con i tipi primitivi è diretta
 - Con gli oggetti è richiesta la keyword new

- Notazione .dot
 - Con i tipi primitivi non è presente
 - Con gli oggetti si

Un caso particolare sono le stringhe:

- Nel momento in cui io definisco una stringa (allocata tra apici, inizializzandola) creo una sorta di costante (trattate quindi come tipi primitivi)
- Nel momento in cui queste sono inserite in run-time le stringhe diventano degli oggetti a tutti gli effetti (se faccio un confronto -> faccio un confronto di indirizzi) -> risolvo con la notazione .dot , compareTo, STRINGA.equals (etc)

L'operatore stringa quindi va posizionato tra gli oggetti e i tipi primitivi

Si introduce il concetto da leggere dalla tastiera -> Scanner:

```
Scanner tastiera = new Scanner(System.in);
```

Si aggiunge con -> **import java.util.Scanner;**

Per leggere da tastiera fino all'invio

```
tasteria.nextLine();
```

Lezione de 20-10

Array in Java

```
public class EsperimentiVettori{

    public EsperimentiVettori(){

        int[] vettore = {0,20,3,49};

        for(int i = 0; i < vettore.length; i++){
            System.out.println("La cella "+ i +" contiene il valore " +
vettore[i])
        }

        int[] vettore = new int[10];
    }

}
```

Per dichiarare un vettore e fatto riempire da tastiera userò la keyword **new**:

```
int[] vettore = new int[10];
```

Di default anche i vettori verranno inizializzati a 0, null

Ragionamenti per vettori di stringhe:

```
public class EsperimentiVettori{  
  
    public EsperimentiVettori(){  
  
        String[] vettoreAuto = new String[4];  
  
        vettoreAuto[0]="Ferrari";  
        vettoreAuto[1]="Tesla";  
        vettoreAuto[2]="Lambo";  
        vettoreAuto[3]="Maserati"  
  
        for(int i = 0; i < vettoreAuto.length; i++){  
            System.out.println("La cella "+ i +"contiene il valore " +  
vettoreAuto[i])  
        }  
        //FOR POTENZIATO  
        for(String s:vettoreAuto){  
  
            System.out.println("Il vettore contiene"+ s)  
        }  
    }  
}
```

Con il for potenziato ci permette di compiere meno errori inizializzando una variabile del tipo di vettore che abbiamo che a ogni ciclo viene inizializzata alla cella successiva fino alla fine

Per l'ordinamento dei vettori abbiamo -> java.util.Array (che ci permette di fare ricerca e ordinamento)

```
public class EsperimentiVettori{  
  
    public EsperimentiVettori(){  
  
        String[] vettoreAuto = new String[4];  
  
        vettoreAuto[0]="Ferrari";  
        vettoreAuto[1]="Tesla";  
        vettoreAuto[2]="Lambo";
```

```

vettoreAuto[3]="Maserati"

//FOR POTENZIATO (FOR EACH)

System.out.println("Il vettore prima dell'ordinamento:");
for(String s:vettoreAuto){

    System.out.println("Il vettore contiene"+ s)
}
java.util.Arrays.sort(vettoreAuto)

System.out.println("Il vettore dopo l'ordinamento")
for(String s:vettoreAuto){

    System.out.println("Il vettore contiene"+ s)
}
}

}

```

Per i vettori non monodimensionali (matrici):

```

int[][] matrice = new int[10][20];

for(int i = 0; i < matrice.length; i++) //per le righe
    for(int j=0; j<matrice[i].length; j++) //per le colonne
        System.out.println(matrice[i][j]);

```

In java non esistono matrici come blocco unico, ma un vettore di vettori (un vettore che punta ad altri vettori)

Il for potenziato può essere utilizzato anche per le matrici:

```

for(int[] riga:matrice)
    for(int cella:riga)
        System.out.println(cella);

```

I linguaggi OO permettono di introdurre in maniera semplici delle versioni specializzate (più ricche) di un determinato tipo di dato definito precedentemente.

Creo una nuova classe -> Esempio ContoCorrente -> ContoCorrenteConMutuo e gli diciamo figlia

```

public class ContoCorrenteConMutuo extends ContoCorrente{
//ATTRIBUTI
    private double MutuoResiduo=0;

```

```

//COSTRUZIONE
    public ContoCorrenteConMutuo(String ib, Cliente c, double
mutuoResiduo){
        super(ib,c);
        MutuoResiduo= muotoResiduo;
    }
//METODI
    public void AccendiMutuo(double valore)
    {
        if(valore > 0)
            MutuoResiduo=valore;
    }

    public boolean PagaRata(double ImportoRata){
        if(importoRata <=Saldo)
        {
            Saldo= Saldo-importoRata;
            MuotoResiduo= MutuoResiduo-importoRata;
            return true;
        }
        else
    }
}

```

Lezione del 22-10

È possibile definire all'interno di una sottoclasse una funzione del tipo (stampa dettagli conto) che ci dia informazioni in più

Prende il nome di *overriding* (letteralmente -> prevalere su)

Il metodo della sottoclasse prevale su quello della superclasse-> nel momento che c'è una chiamata al metodo l'OO richiama quello della sottoclasse

Array di oggetti

Sintassi:

```

ContoCorrente[] elencoConti = new ContoCorrente[4];

elencoConti[0] = new ContoCorrente("12345", cliente1);
elencoConti[1] = new ContoCorrenteConMutuo("456jgdk", cliente2, 500000);
//la sottoclasse si può comportare come superclasse
elencoConti[2] = new ContoCorrenteConMutuo("4565jgfk", cliente3, 200000);
elencoConti[3] = new ContoCorrente("789dssa", cliente4);

```

```
for(ContoCorrente conto:elencoConti)
    conto.StampaDettagliConto();
```

Vettore di oggetti -> vettore di puntatori

Click sulla classe -> source -> Override method

Lezione del 27-10

METODI GETTER E SETTER

Getter -> valore che permette la lettura di un'attributo

Setter -> valore che permette di scrivere (e salvare un dato) nell'attributo

```
private tipo NomeAttributo,  
  
public tipo /getNomeAttributo/ (){  
}  
  
public tipo /setNomeAttributo/ (){  
}
```

Nel codice:

1. Evitare le abbreviazioni
2. Per le parole:
 - a. CAMEL CASE (StampaDettagliConto) -> convenzione standard Java
 - b. Underscore Case (stampa_dettagli_conto)
3. Java Obfuscating

Lezione del 29-10

Class e sequence diagram

UML -> unified modeling language

Progettazione SW fondamentale

Caratteristica della tabella:

Nome Classe
attributi

Metodi e funzioni della classe

Commenti (opzionale)

Esempio:

Nome Classe: Prodotto

Attributi: Nome, Prezzo, Disponibilità, Codice, Marca

Dati da inserire:

1. Nome attributo (obbligatorio)
2. Tipo (facoltativo)
3. Visibilità -> Simboli +, - (public e private)

In questo nome avremo una cosa del tipo:

Prodotto
+Nome: String
+Prezzo: double
+Disponibilità: Int
+Codice: String
+Marca: String

Collegato a

Carrello
+DataCreazione: date
+CapienzaMax: int
+Sconto: double
+ApplicaSconto(CodiceSconto: String)
+CalcolaPrezzoFinale(): double
+isVuoto(): boolean
+getNumeroProdotti():int
+AggiungiProdotto(P: Prodotto): boolean
+RimuoviProdotto(P: Prodotto): boolean

Sequence diagram -> diagram per ogni funzionalità

Presenta una componente in più rispetto al diagram: il tempo

Mira a identificare l'ordine delle chiamate dei metodi

Ho un verso di lettura da sinistra verso destra e dall'alto in basso

Classe 1 Classe 2

f() -> <--- return value

Chiamata una func() La classe 1 esegue una evocazione sulla classe 2 (f()), e questa a sua volta restituisce un valore (opzionale) (return value, che viene rappresentato con il trattagliato)

Rettangolo pieno -> computazione -> ritorno value

Asse Verso il basso -> tempo

Asse Verso sinistra -> spazio (?)

I sequence diagram presentano dei tipi molto particolari per le condizioni

Rettangoli:

- OPT [cond] -> solo se true
- ALT [cond] blocco 1 --- 2 -> if / else

Lezione del 03-11

Class diagram -> rappresentazione 1 a 1 con un blocco di codice

Un'associazione 1 a 1 si rappresenta o mixando le due classi o mappare attraverso una chiave (chiave primaria ed esterna)

Caso studente- tesi associata:

Aggiungo un attributo alla classe studente

Classe Tesi:

```
public class Tesi{  
    public Titolo;  
    public Materia;  
}
```

Classe Studente:

```
public class Studente{  
    public String Nome;  
    public String Cognome;  
    public String Matricola  
  
    //AGGIUNTA  
    public *Tesi* LaTesi;  
    //COSTRUTTORE  
    public Studente (String Nome, String Cognome, String Matricola)  
        super();
```

```

Nome = nome;
Cognome = cognome;
Matricola = matricola
}
//GETTER E SETTER {...}

```

La costruzione di un costruttore con parametri nasconde il costruttore di default (quello vuoto) -> per avere quello vuoto bisogna settarli entrambi

Main:

```

public class Driver{
    public static void main(String[] args){
        Studente s = new Studente("Mario", "Rossi", "N86...");
        Tesi t = new Tesi("TitoloDellaTesi", "IngDelSoftware");
        s.setLaTesi(t);
    }
}

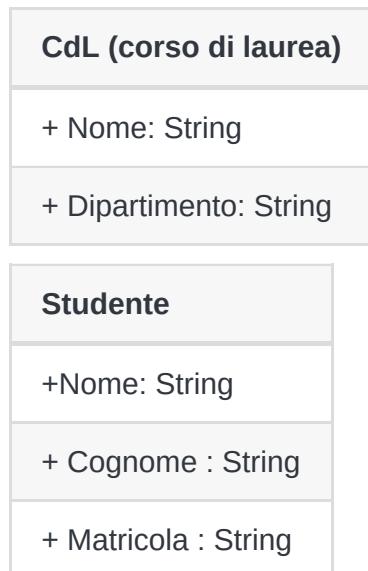
```

In OO le associazioni sono di tipo monodimensionale -> per indicare il senso si scrive una freccia piena nella direzione verso cui punta o una freccia aperta alla fine dell'associazione

Per rappresentare un'associazione di tipo bidimensionale dobbiamo creare riferimenti in entrambi le classi (in questo caso un per lo studente e uno per la tesi)

Queste aggiunte non vengono fatte di nuovo a monte nel class diagram

Cardinalità mutlipla



Associazione di tipo 1 a *

```

public class CorsoDiLaurea{
    public String Nome
    public String Dipartimento
}

```

```

public CorsodiLaurea(String Nome, String Dipartimento){
    super();
    Nome = nome;
    Dipartimento = dipartimento;
}
}

```

Main :

```

public class Driver{
    public static void main(String[] args){
        CorsoDiLaurea cdl1 = new CorsoDiLaurea("Informatica", "DIETI")
        CorsoDiLaurea cdl2 = new CorsoDiLaurea("Ing", "DIETI")
        Studente s1 = new Studente("Mario", "Rossi", "N86...", cdl1);

        System.out.println("Lo studente"+ s1.Cognome+"è iscritto a :"+
s1.IscrittoA.Nome)
    }
}

```

Con la notazione dot posso iterare sui vari passi

Lezione del 10-11

Il vantaggio del' OO è la gestione delle situazioni di errori (come il caso dei segmentation fault). Java mette a disposizione nuove classi e keyword per la gestione di questi problemi.

Queste keyword sono:

```

try {
    ...
}

}
catch{
    ...
}

```

Al di fuori di queste due keyword il programma continua a funzionare normalmente continuando con l'esecuzione successiva

Es.

```

public class Starter{

    public static void main (String[] args){

```

```

int [] vettore = new int [5];

try {
    for(int i=0; i<=5; i++)
        System.out.println("Il vettore nella cella"+ i+"contiene il
valore" +vettore[i]);
}

catch(Exception e){
    System.out.println("Attenzione, scansione array errata");
}

System.out.println("Ho finito");
}
}

```

Nonostante nel try ci sia un errore di segmentation fault, il codice continua all'istruzione successiva, skippando quella errata

Una stessa riga può generare diverse politiche di recovery-> con Exception creiamo un'eccezione
Nel caso in cui nel try ci siano possibilità di errori diversi è possibile inserire un try/multi-catch block (genero più catch per uno stesso try) andando a definire diverse strategie in base all'errore che viene generato

Gestione di problemi a Run-time in Java

Java prevede due classi:

- **Error**, errori che non sono recuperabili dalla macchina e porta alla chiusura dell'esecuzione del programma (es. allocazione eccessiva della memoria-> saturazione della memoria RAM);
- **Exception**, divise in:
 - **Runtime exception**, dette Eccezioni unchecked (array index out..., input mismatch....), che non richiedono obbligatoriamente un try-catch
 - Classi che non rientrano in Runtime-exception, dette Eccezioni checked, che obbligatoriamente richiedono un try-catch

Keyword **throw** -> viene usato per "lanciare" qualcosa-> per funzionare devo definire nella public void di una funzione un throws

Lezione del 17-11

Classi wrapper e Collection

Oltre ai tipi primitivi e Object, viene introdotto il concetto di Autoboxing e Unboxing (Classi Wrapper). Per ogni tipo primitivo si crea una classe equivalente che permette di "renderli" oggetti

L'autoboxing è una cosa che compie il compilatore che nel momento in cui trova un tipo primitivo all'interno di un contesto object oriented sostituisce la riga in maniera trasparente in questo modo da:

```
vett[2] = 5;
```

passiamo a

```
vett[2] = new Integer (5)
```

L'unboxing invece effettua in maniera analoga l'operazione inversa (permette di trasformare ad esempio un Integer in un int)

Esempi

```
public class Starter{  
    public static void main(String args[]){  
        int i = 10;  
        Integer a = i;  
        System.out.println("Il valore di a è:"+a);  
  
    }  
}
```

Integer è disponibile nel package del linguaggio di programmazione out of the box.

In questo modo il compilatore mi stampa a senza problema (lo si vede come a.intValue())

Il casting al volo (di un integer ad un double ad esempio) non è possibile ma è necessario farlo a "mano":

```
Integer i=10;  
Integer j=10;  
  
Double result = new Double(i+j)
```

L'operatore == :

- Nei tipi Object confronta gli indirizzi;
- Nei tipi primitivi invece confronta il valore;
- Nei tipi Wrapper invece si comporta negli oggetti (confronta gli indirizzi di memoria)-> analogamente agli object utilizzo la notazione .equals()

Per una questione di ottimizzazione i primi 128 numeri sono gestiti come tipi primitivi (pertanto la notazione == si comporterà in maniera analoga a dei tipi primitivi). Questo può tornare utile nei cicli

Per collection si intende banalmente la struttura dati (contenitori di informazioni).

Le liste e gli array presentano pro e contro inversi:

- Pro

- Gli array hanno accesso diretto alle informazioni
- Le liste presentano una modifica alla struttura istantanea
- Contro
 - Gli array sono molto lenti nella modifica
 - Le liste presentano un accesso sequenziale alle informazioni (che è più lenta dell'array)
 -

Collections Map (conservano due informazioni in ogni cella)

Collection
Set
List
Queue (code)

Le tre macrofamiglie delle collections presentano a loro volta famiglie specializzate:

- Per le liste introdurremo le **ArrayList** e le **LinkedList**
 - Sono da considerare come degli oggetti (e quindi presentano dei metodi)

```
import java.util.*;

public class Starter{
    public static void main(String args[]){
        ArrayList vett = new ArrayList();
        vett.add("Ciao");
        vett.add("Mondo");
        System.out.println(vett.get(1)); //stampa la parola Mondo (come
un vettore normale conta da 0)

        for(Object o:vett)
            System.out.println(o); //mi stampa tutto il contenuto
sfruttando il for potenziato

        vett.removeAll(vett); //nel caso volessi eliminare tutti i
dati nell'arraylist
        if(vett.contains("Ciao")) //ci restituisce un vettore booleano
che restituisce vero se la struttura contiene il valore cercato
            System.out.println("Trovato");

    }
}
```

}

Questo tipo di struttura dati è dinamica -> non devo preoccuparmi della dimensione (ci pensa la virtual machine) e non descrivo il tipo (posso inserirci qualsiasi cosa che sia un oggetto)

Posso velocemente passare da ArrayList a LinkedList-> il grande vantaggio è che le collezioni presentano gli stessi metodi

OO Lezione 19 11

OO Lezione 19 11

Framework -> insieme di strumenti

ArrayListi

Metodi:

- .add(elem) -> aggiunge un elemento in coda
Java distingue in una struttura dati due informazioni:
 - la capacità (quanta memoria è stata assegnata ad una determinata struttura dati):
 -

Una volta saturata la capacità java crea un array del 50% più grande.

Un arraylist è una classe -> la sua implementazione è disponibile in rete

Trovare il codice della linked list e capire come venga implementato il metodo add

Posso scansionare una collezione sfruttando il for potenziato, ma posso utilizzare anche l' **iteratore** (detto anche cursore), che presenta due funzionalità fondamentali:

- bool hasNext() -> restituisce un valore booleano
- elem Next() -> restituisce il contenuto della struttura dati

Es.

```
Iterator i = vett.iterator();
while (i.hasNext())
    System.out.println(i.Next());
```

Nel momento in cui l'iteratore termina il ciclo devo invocare un nuovo iteratore per ciclare di nuovo.

L'iteratore è la struttura standard dell'informatica per leggere strutture dati

Sulla linked list ho a disposizione il ListIterator (che funziona in maniera analoga all'iteratore normale), ma solo su queste ho la possibilità di definire sull'iteratore il comando **hasPrevious()**. Analogamente avremo quindi

```
ListIterator li = vett.listiterator();
while (li.hasPrevious())
    System.out.println(li.previous());
```

Cioè è possibile perchè le linked list sono liste doppiamente linkate

La sintassi a parentesi angolari permette di specificare il contenuto di una determinata collezione. Ad esempio:

```

ArrayList<Cliente> arr = new ArrayList<Cliente>();
arr.add(cliente1);
arr.add(cliente2);
Iterator i = vett.iterator();
while (i.hasNext())
    System.out.println(i.Next());

```

In questo caso mi stampa tutto il contenuto del cliente 1 e 2

Lezione del 24-11

Keyword This e Super

This viene utilizzata per:

- Gestione dello shadowing delle variabili (coprire con un ombra)
- Gestione delle associazioni bidirezionali

```

public class ContoCorrente{
    private String IBAN;
    private Cliente Intestatario;

    //COSTRUTTORE
    public ContoCorrente(String IBAN, Cliente Intestatario){
        this.IBAN = IBAN; //attributo iban della classe
        this.Intestatario = Intestatario;
    }

}

```

Il parametro effettua l'operazione di shadowing sulla classe

Nel momento in cui utilizzo la keyword this ci riferiamo all'istanza attuale, è un modo elegante per indicare l'indirizzo di memoria dell'oggetto a cui devo fare riferimento

Con this intendiamo l'indirizzo di memoria dell'oggetto corrente (l'indirizzo di memoria dell'oggetto). Con questa keyword riusciamo a gestire le associazioni bidirezionali (caso proprietario-auto ad esempio)

Possiamo vedere super come un this per la superclasse (ha un utilizzo più limitato)

Quando creo un oggetto sottoclasse per creare in memoria un qualcosa che estenda, sono obbligato a creare un oggetto padre. La VM crea la classe padre, poi crea lo spazio per le variabili aggiuntive (le extend).

Nel costruttore quindi la prima cosa da fare è istanziare l'oggetto padre (con super());)

Lezione del 26-11

Versioning

Quando va fatto il commit?

- Solo nel caso in cui il codice sia stabile e funzionante
- Evitare di fare commit nel caso in cui il codice non compili

Esistono due tipi di strumenti:

- SVN (SubVersioN) (centralizzato)
 - richiede un server (repository)
 - client
 - Integrato con gli IDE
 - Integrato con S.O. (ad esempio Tortoise SVN)
- GIT (De-centralizzato)
 - ogni programmatore in locale mantiene l'intero DB di tutte le versioni

Passi per il versionamento

(lato client)

1. Check-out → creazione delle cartelle in locale

2. Update/Pull

3. Commit - Push

Interface

Astrazione quanto più possibile dai tipi che andremo a rappresentare.

Polimorfismo → capacità dei linguaggi OO di mettere all'interno di una variabile padre qualunque tipo discendente da quello (una classe può assumere più forme).

In questo modo posso richiamarmi con una volta soltanto un'invocazione una funzionalità di queste classi indipendentemente dal tipo di classi.

Classi che non possono essere stanziate (interface)

Si tratta di classi che io definisco e posso essere utilizzate solo ai fini del polimorfismo, ma non possono essere utilizzate in maniera reale.

In una interfaccia ho dei metodi in cui non esiste un' implementazione, ma esiste esclusivamente la firma (signature).

Possono contenere solo la definizione di metodi. Esiste solo la signature.

Le interfacce non si estendono ma si implementano (**implements**).

```
public class Rettangolo implements FormaGeometria{  
    // IMPLEMENTS = IMPLEMENTO TUTTI I METODI DEFINITI NELL'INTERFACCIA  
}
```

L'interfaccia non è figlia di Object, ma tutte le classi che implementano l'interfaccia sì.

Le interfacce non contengono codice, solo firme.

Le classi che contengono la keyword **implements** devono offrire l'implementazione di tutto ciò che contiene FormaGeometrica.

Il termine interface viene molto spesso usata come sinonimo di "**contratto**", nel senso che:

- Tutte le classi che implementano un' interfaccia permettono di rispettare un contratto di implementazione → promettono di offrire il codice di tutti i metodi definiti nell'interfaccia

Mentre il concetto di ereditarietà è singola, il concetto di interfacce è multiplo

Un esempio di interfaccia è **List** implementata da 2 classi concrete (ArrayList e LinkedList)

Lezione del 01-12

Interfacce grafiche

Modo standard di definire un interfaccia grafica → Java Swing a cui si affiancano le java FX (più recenti)

JFrame → una "cornice" che contiene gli elementi da inserire.

Ogni elemento di interfaccia grafica è un oggetto della corrispondente classe offerta dal jdk.

All'interno del JFrame vanno inseriti gli oggetti con cui l'utente andrà ad interagire. Questi prendono il nome di **Controlli**. Un esempio può essere:

- jTextFielded (che permette di inserire testo);
- JLabel (testo normalmente non interattivo che il programmatore utilizza per dare indicazioni testuali all'utente)
- Bottoni (classe JButton);
 - combobox;
 - checkbox;
 - roundbutton;

Lo sviluppo di interfacce grafiche richiede molto codice → è impensabile svilupparla senza un IDE.

IntelliJ e VS code di default offrono un supporto a questo.

Eclipse necessita il pacchetto window builder (dal marketplace)

Per creare:

File > other > SVT designer > SVT JAVA project

Per creare un JFrame → other → JFrame

Di default crea una finestra come estensione di JFrame, un main e un costruttore.

Presenta un:

- setBounds → coordinate e dimensioni delle finestre

Per gestire la finestra utilizzo la voce **design** dove posso settare ogni caratteristica e gestione della finestra (comprese le azioni della finestra)

La schermata è suddivisa in:

- **Palette** → contenere l'elenco di tutti gli elementi che possono essere inseriti all'interno della finestra;
- **Componenti**;
- **Proprietà**, (attributi di classe del JFrame) come ad esempio il Titolo.

Il problema fondamentale delle interfacce grafiche è il ridimensionamento

Pertanto nel momento in cui vado a specificare un frame e voglio inserirci dei controlli ne devo specificare le sue dinamiche (come voglio gestire il ridimensionamento degli elementi). Posso affrontare il ridimensionamento in due modi:

- Fisso la posizione degli elementi (in maniera assoluta)
- Scelgo un layout fluido (dinamico)

Definisco la keyword **layout**. Quando creo delle interfacce grafiche inserisco se necessario dei pannelli. Per ogni pannello posso decidere che tipo di layout debba assumere. In Java sono presenti diversi tipi di layout:

- **Assoluto** → gli elementi sono fissati indipendentemente dal ridimensionamento;
- **Border** → gli elementi vengono messi all'interno di panelli;
- **Flow** → flusso accodato (nel caso in cui la finestra venga rimpicciolita inizia a spostare gli oggetti uno sotto l'altro)

Nel caso in cui non si definisca un absolute layout l'IDE chiede la posizione degli oggetti

Una prassi di programmazione è la rinomina degli elementi di interfaccia grafica, per rendere più semplice la modifica. Come suffisso al nome dell'oggetto si è soliti aggiungere il tipo dell'elemento della GUI.

TF → TextField

Eventi

Un evento è qualcosa che succede nell'interfaccia grafica. Java ci permette di definire quali elementi dell'interfaccia grafica debbano essere sensibili agli eventi che compie l'utente

Quando viene scritto il codice per un interfaccia grafica nella classe dell'elemento ho:

- definizione attributi e costruttore (che contiene il codice per aggiunta elementi grafici)
- Metodi, ognuno dei quali corrisponde ad una determinata azione che vogliamo compiere, detti **Gestori di eventi**

In maniera intelligente la JAVA VM attiva i gestori degli eventi. Sfruttiamo l'utilizzo di

```
Button.addActionListener(new ActionListener());
```

Nei sistemi con interfaccia grafica una volta partito il main del programma, si avvia un loop infinito che mantiene l'interfaccia attiva e reattiva a tutte le azioni che il programmatore ha definito.

Lezione del 03-12

Gli eventi e le funzioni che un bottone può svolgere

All'interno dell>window builder nella tabella proprietà alla voce show events sono presenti tutte le azioni che è in grado di svolgere

All'interno di una GUI è possibile scegliere di creare più finestre che partano con una visibilità false → setVisible(false); → vengono sbloccate nel momento opportuno

Si avranno quindi una catena di JFrame, da gestire in maniera interattiva in base agli eventi.

Si consiglia di utilizzare una classe Controller per il passaggio da una finestra ad un'altra (evitare di lanciare un comando diretto tra le finestre)

Lezione del 07-12

Contatto OO-BDD

Problema: L'obiettivo di fondo è quello di realizzare dei sistemi informativi (gestionali) (come far comunicare i due mondi). I due problemi fondamentali sono:

- Le interfacce grafiche possono cambiare;
- La base di dati sottostante può cambiare.

Questa necessità è molto spesso ripetuta durante lo sviluppo di un programma (un'esigenza molto forte).

Il concetto di base è scrivere codice pronto ad evolversi.

Si introducono pertanto delle classi che hanno lo scopo di mediare l'interazione tra il DB e l'interfaccia grafica (il codice JDBC è scritto esclusivamente in queste classi) → anelli di congiunzione tra questi due mondi → centralizzo ciò che andrà ad essere modificato.

Queste classi sono collegate alle interfacce attraverso delle collezioni (collections tipo ArrayList)

Le classi JDBC si occuperanno di tradurre in modo corretto i dati

Il modo più diffuso di procedere è quello di **DAO** (Pattern Date Access Object)

Pattern

Design Pattern → meccanismo per descrivere un'idea di buona progettazione OO (riusare design)

Un esempio di Design Pattern può essere l'iterator

Un Design Pattern presenta 4 parti:

- Nome;
 - Problema;
 - Soluzione;
 - Conseguenze e compromessi della scelta.
-

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Observer → design pattern per le Swing

Singleton → serve per creare un numero limitato di istanze

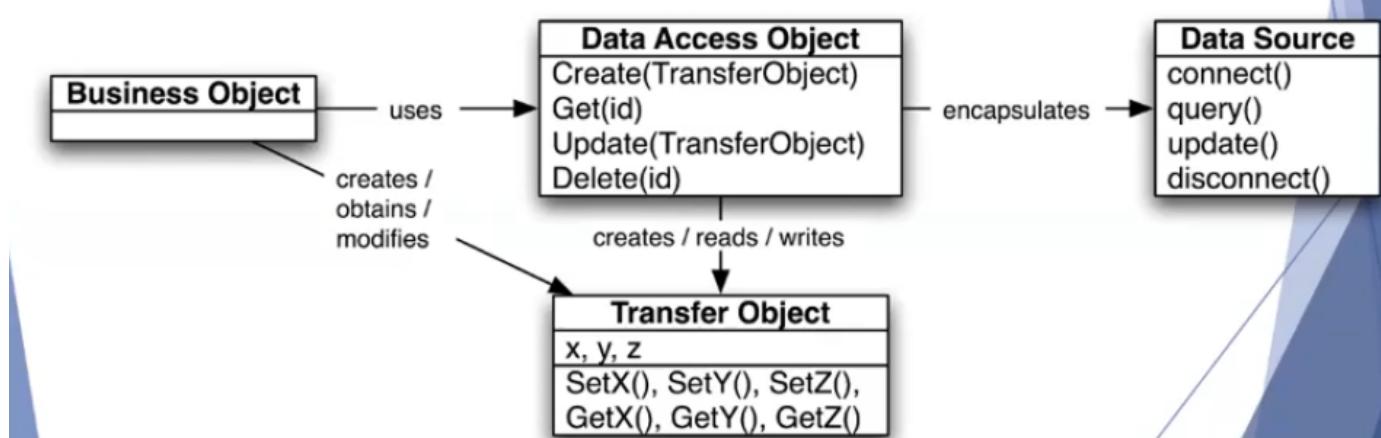
DAO
(data access object)

Problema:

- Accesso ai dati varia in base al tipo di storage;

Soluzione

- Usare IL DAO per attrarre e encapsulare gli accessi alla fonte dei dati. Ogni DAO gestisce la connessione con la data source per ottenere e gestire i dati
- Il DAO implementa l'accesso richiesto per lavorare con la fonte dei dati



- Un DAO class per ogni classe
 - Operazioni di astrazione CRUD (Create, Retrieve, Update, Delete)
- **Vantaggi**

- Implementazione di diversi tipi di storage (inseriti con un minimo impatto sul resto del sistema)
- Riuso del codice
- Disaccoppiamento dei layer persistenti

Per ogni classe che rappresenta un'entità del mio problema va creata una classe corrispondente (con suffisso DAO) → ingloba tutti i metodi per gestire la persistenza della classe; un esempio di metodo (riguardo il caso classe Studente) potrebbero essere:

- `save(Studente) -> bool`
- `getbyMatricola(String M) -> Studente`

In particolare possiamo vedere la classe DAO come un interfaccia dove vengono sponsorizzati tutti quanti i metodi del data souce che stiamo gestendo

All'interno della classe DAO inserisco tutte le operazioni che posso fare al di sopra del DB.

Es.

```
public class Driver{
    public static void main(String[] args){
        StudenteDAO SD = new StudenteDAO();
        Studente s = SD.getStudenteByMatricola("N86001234");
        System.out.println(s);
    }
}

import java.util.List;
public class StudenteDAO{
    public boolean SalvaStudente(Studente s){
        //codice per salvare su DB lo studente s
        System.out.println("Salvataggio su db");
        return true;
    }
    public Studente getStudenteByMatricola(String matr){
        //eseguo query su db, con where per la matricola
        //Scandisco il resultset, che dovrebbe avere al più un record
        //Se il resulset contiene un rigo, creo uno studente con i valori
        presi dal resulset
        String matr="N8600123"; //primo campo resultset
        rs.getString(Matricola);
        String nome="N8600123"; // primo campo resultset
        rs.getString("Nome");
        String cognome="N8600123"; // primo campo resultset
    }
}
```

```

rs.getString("Cognome")
    System.out.println("Ho letto sul DB una tupla di uno studente e la
restituisco");
    Studente s = new Studente(matr, nome, cognome);
    return s;
}

public List<Studente> getStudentiByCdl(String Cdl){
    //Va importata List
    //eseguo query su db, con where matricola LIKE "Cdl%"
    //Scandisco il resultset, che dovrebbe avere da 0 a n valori
    //Se il resultset contiene valori, creo una collection di studenti
    //con i valori presi dal resultset
    ArrayList<Studente> theList = new ArrayList();
    Studente s1 = new Studente("N8600123","Luca","Verdi",);
    Studente s2 = new Studente("N8600123","Viola","Bianchi",);
    theList.add(s1);
    theList.add(s2);
    return theList;
}
}

```

Sulla classe StudenteDAO alla voce Refactor → extract interface in cui è possibile creare un interfaccia con i metodi contenuti nella classe. Creo un contratto in cui qualunque mezzo che mi offra persistenza di studenti, mi dovrà implementare i metodi contenuti e la loro specifica signature

L'applicazione del DAO è praticamente obbligatoria.

Lezione del 10-12

Progettazione di sistemi informativi con interfacce grafiche e DB

La soluzione in cui i frame interagiscono direttamente tra di loro non è molto vantaggiosa nell'ipotesi in cui si debbano fare poi delle modifiche.

Per questo motivo si scegli di delegare ad una nuova classe la supervisione dell'interazione di tutti i frame (Controller). Contiene una serie di metodi che verranno richiamati dai frame.

Il controllore è una classe che per natura è associata e collegata al resto delle classi

Nel ragionare nella creazione di classi è fondamentale pensare al ruolo che questo deve assumere.
(Scrivere codice facile da gestire per future modifiche2)

Blocchi di classi:

- GUI;
- Controllo/ Supervisione (Logica);
- Gestione della persistenza dati

Nell'IDE creiamo un JFrame e gestiamo in questo modo il Controller

```
public class Controller{

    LoginFrame lf;
    FinestraDue F2;
    public static void main(String[] args){
        Controller = new Controller();
    }

    public Controller(){
        lf = new LoginFrame(this); //nella classe LoginFrame devo passarci
        Controller c
        lf.setVisible(true);
    }
}
```

Gestiamo quindi il caso di action performed. Nella classe LoginFrame

```
public void actionPerformed(ActionEvent e){
    theController.ControlloCredenziali(UserName_TF.getText(),
    Pwd_TF.getText());
}
```

Nello stesso tempo creo all'interno di Controller una classe **ControlloCredenziali**.

```
public void ControlloCredenziali(String username, String password){  
  
    System.out.println("L'utente ha inserito "+username+", "+password)  
    //stampa nella console  
  
    // Chiedi alla classe Utente se esiste qualcuno con username e  
    password  
    // if(Utente.exist(username,password))  
    lf.setVisible(false);  
    F2 = new FinestraDue();  
    fs.setVisible(true);  
}
```

Qualunque finestra dovrebbe avere un collegamento al Controllore

Una particolare finestra (popup) è il JDialog che presenta la caratteristica ontop.

Gli actionPerformed devono avere una quantità di linee di codice ≈ 1 (il loro scopo è solo di richiamare)

I package

In Java esiste il concetto di **package** → un modo che offre Java per raggruppare le classi che compiono cose simili. È possibile creare dei raggruppamenti all'interno di un package (corrisponde ad una directory sul disco) in modo da gestire meglio le cose:

- DAO;
- Entità;
- Interfacce grafiche;
- Controllore.

In Java qualunque libreria è un package.

MEMO:

Una classe DAO per ogni classe di entità

Responsibility driven Design

Progettazione guidata dalla responsabilità → Nel momento in cui creiamo una classe siamo obbligati a esplicitare il senso di questa classe.

Uno strumento utile per far questo è la CRC Card (da associare nella documentazione):

Lezione del 15-12

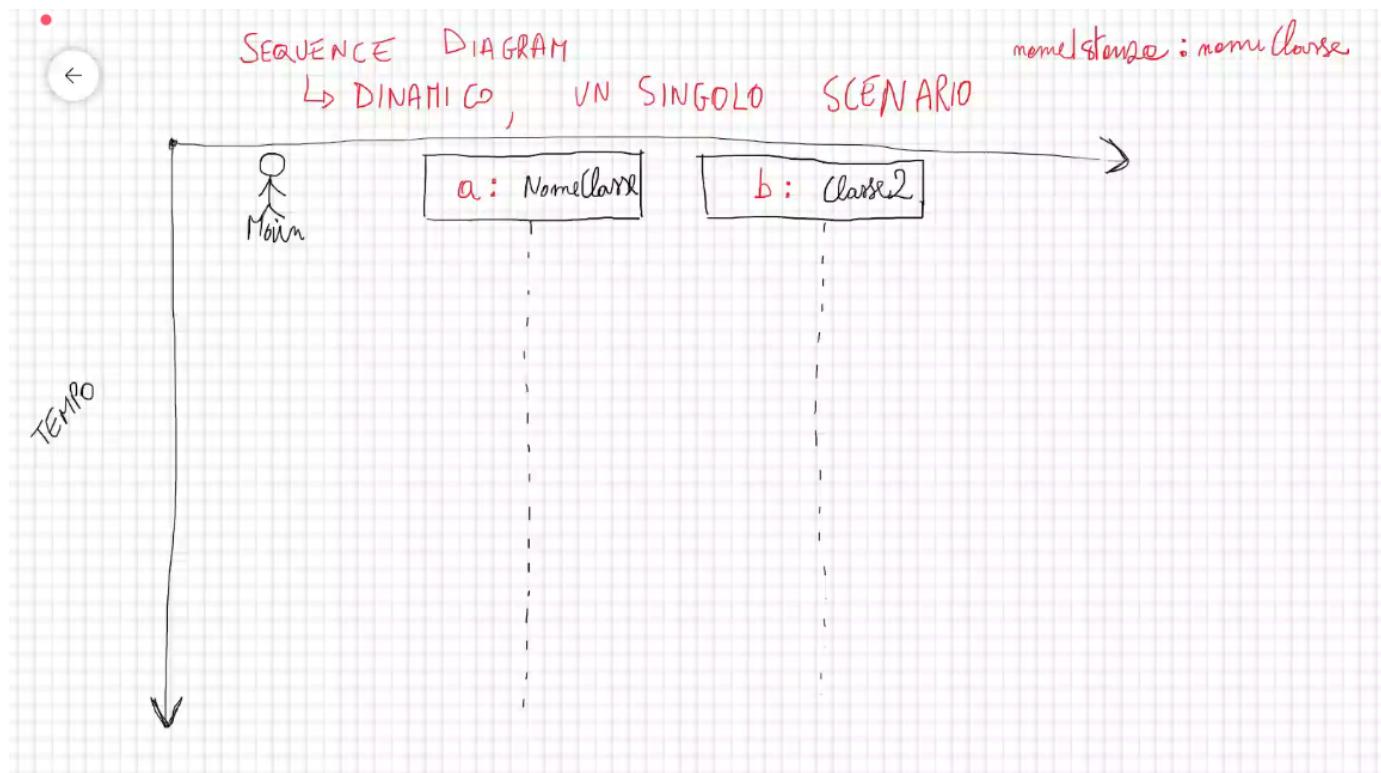
Lezione del 15-12

Sequence diagram

Dinamico → rappresenta un singolo scenario

Sull'asse delle y troviamo il tempo (si avanza con il tempo verso il basso). Sull'asse delle x sono presenti le varie entità coinvolte (lifeline) rappresentate da un rettangolo contenente il nome della classe da cui parte una linea tratteggiata (indica la quantità di tempo in cui esiste questa classe)

È presente anche un tipo di classe diversa (con la forma di un omino che rappresenta l'user)

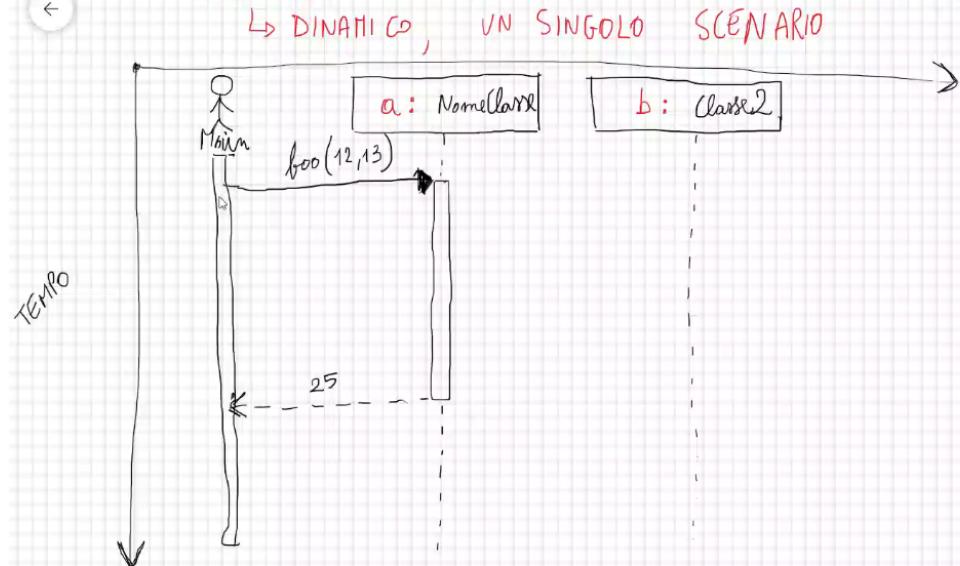


Con la freccia indichiamo l'invocazione di un metodo dell' oggetto la cui lifeline sto raggiungendo con la freccia Quando un oggetto è attivato da un metodo si rappresenta la sua lifeline con un rettangolino

SEQUENCE DIAGRAM

↳ DINAMICO, UN SINGOLO SCENARIO

modelstamps : nomeClasse



class NomeClasse {

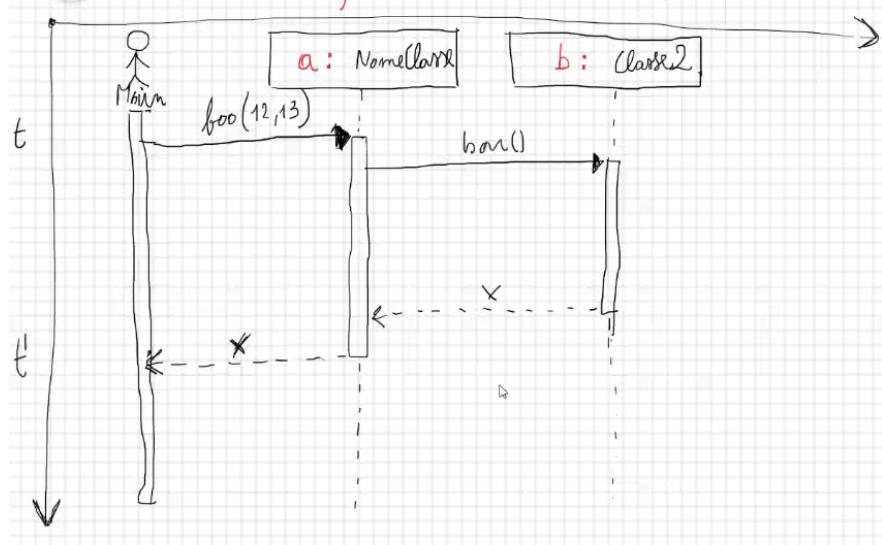
```
int foo (int a, int b){  
    return a+b;  
}
```

Nel caso in cui il metodo foo invochi un'altro metodo (nella classe 2) ci troveremo in questa situazione

SEQUENCE DIAGRAM

↳ DINAMICO, UN SINGOLO SCENARIO

modelstamps : nomeClasse



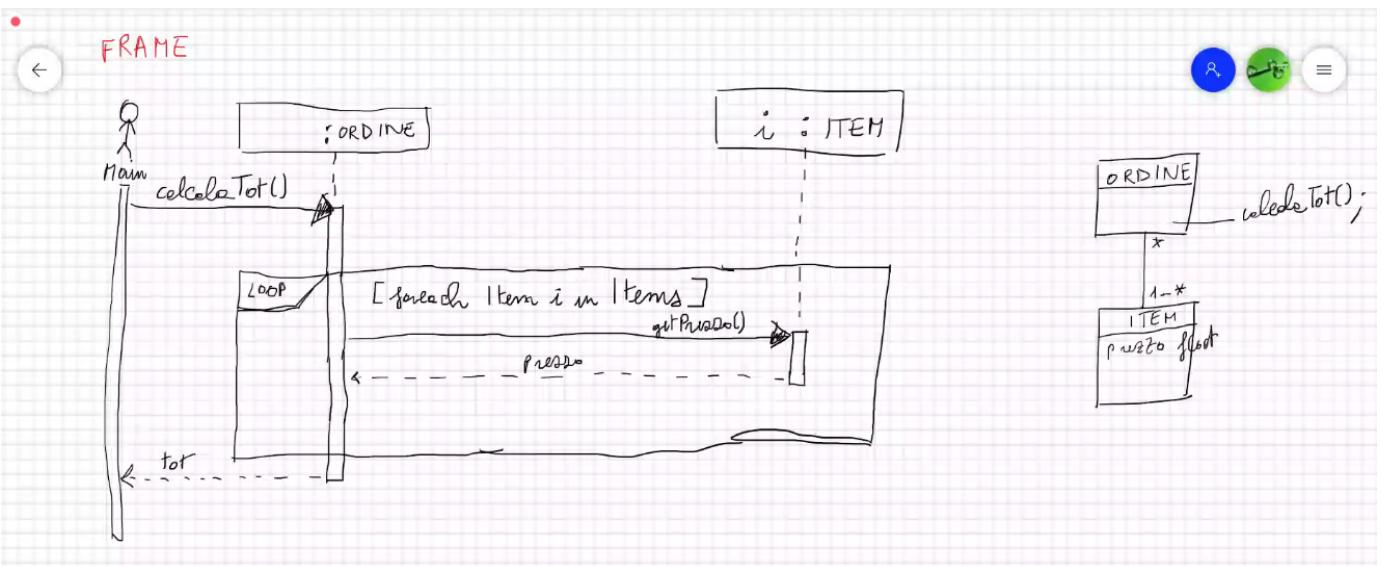
class NomeClasse {

```
int foo (int a, int b){  
    int x = b.bar();  
    return x;  
}
```

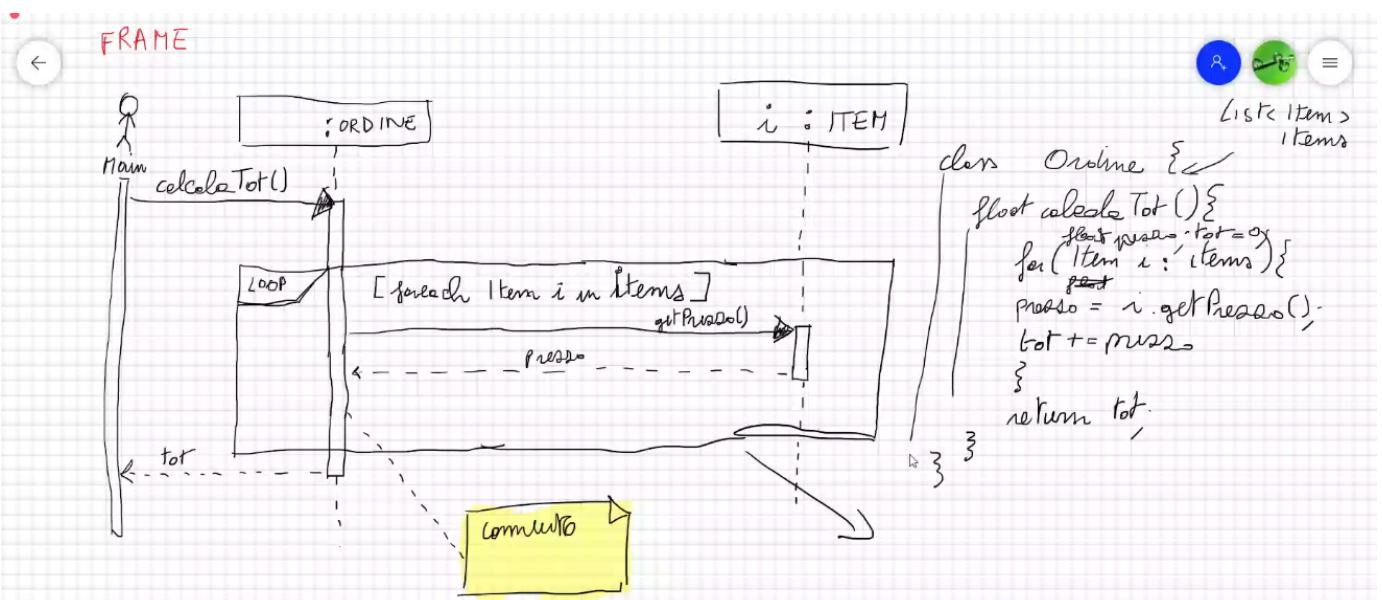
Frame nei Sequence Diagram

Consideriamo il caso di un ordine che presenta una relazione (1 a molti) con gli item e se ne voglia calcolare il totale.

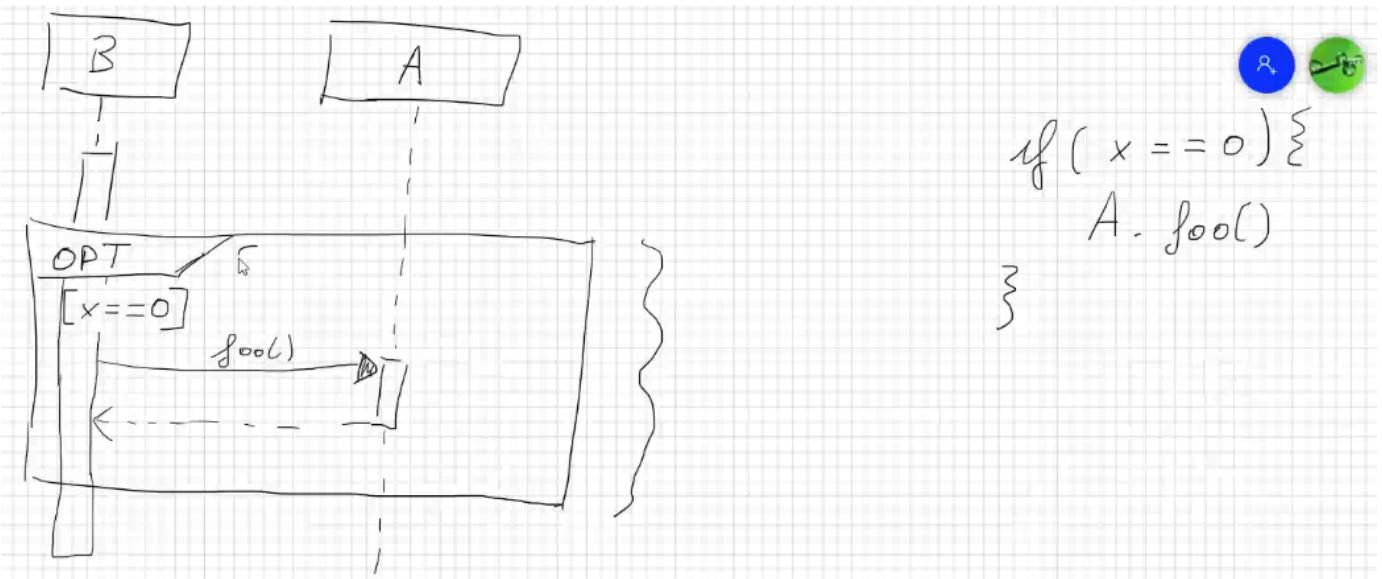
I Frame si rappresentano come quadrati con in alto una descrizione. In questo caso vediamo un Frame di tipo Loop



In particolare per rappresentare dei commenti si utilizza una specie di "post-it" ancorato all'oggetto che sta commentando (con lo scopo di aggiungere una informazione in più)



Frame OPT (optional) → descrive una serie di attivazioni che possono verificarsi o meno in base a determinati fattori

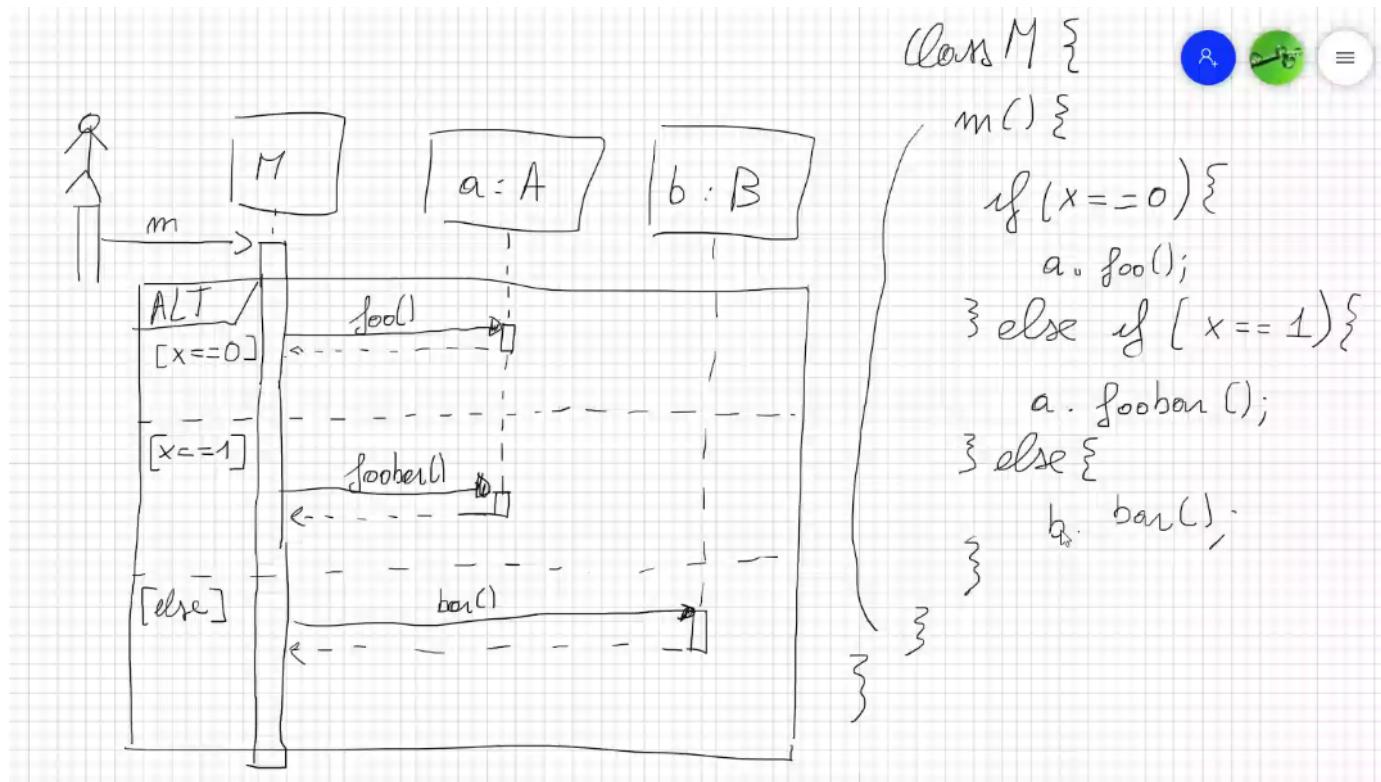


$\text{if } (x == 0) \{$

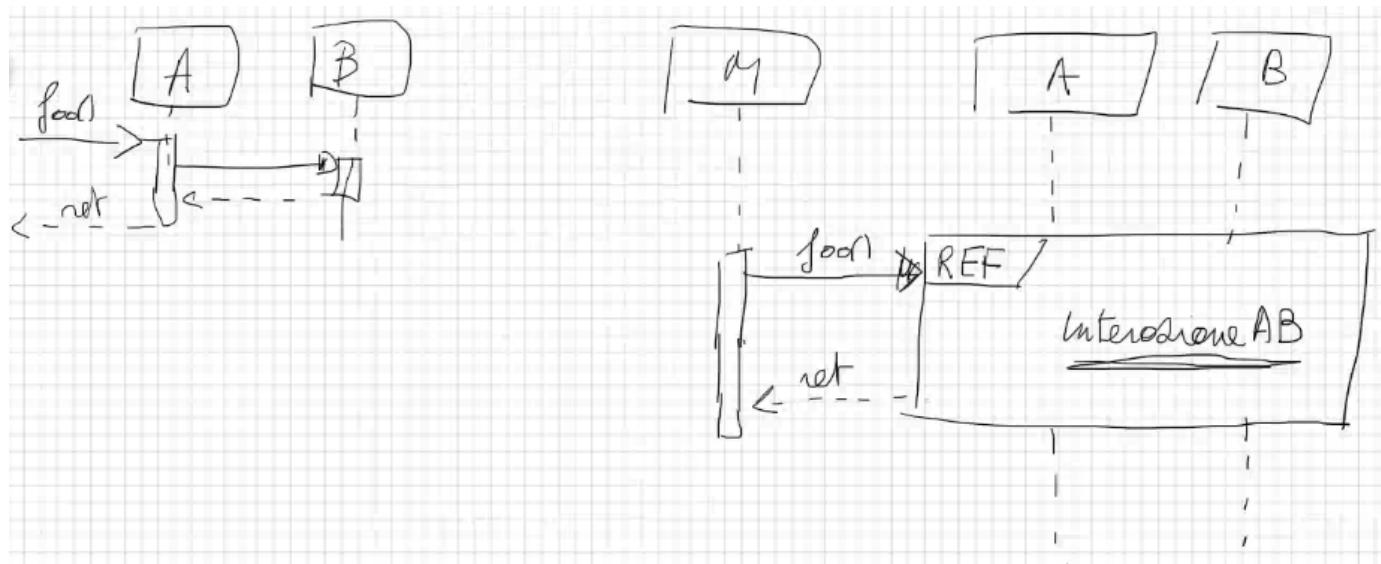
A. fool()

Frame ALT

A differenza di OPT, ALT invece rappresenta delle alternative (mutualmente esclusive tra loro)



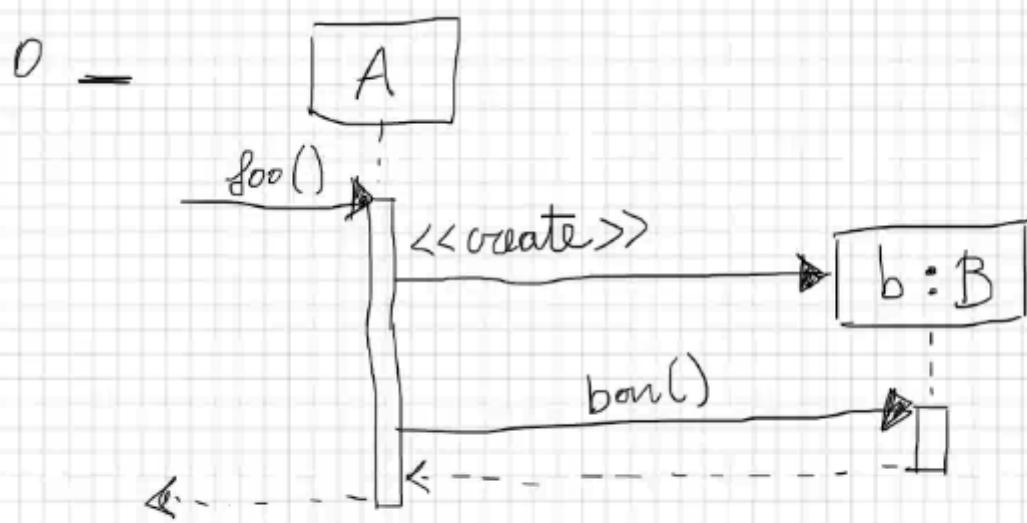
Frame REF → lo si usa per riferimenti ad invocazione a metodi troppo grandi da essere rappresentati su un foglio (si sceglie di rappresentarlo da un'altra parte)



Metodo Create

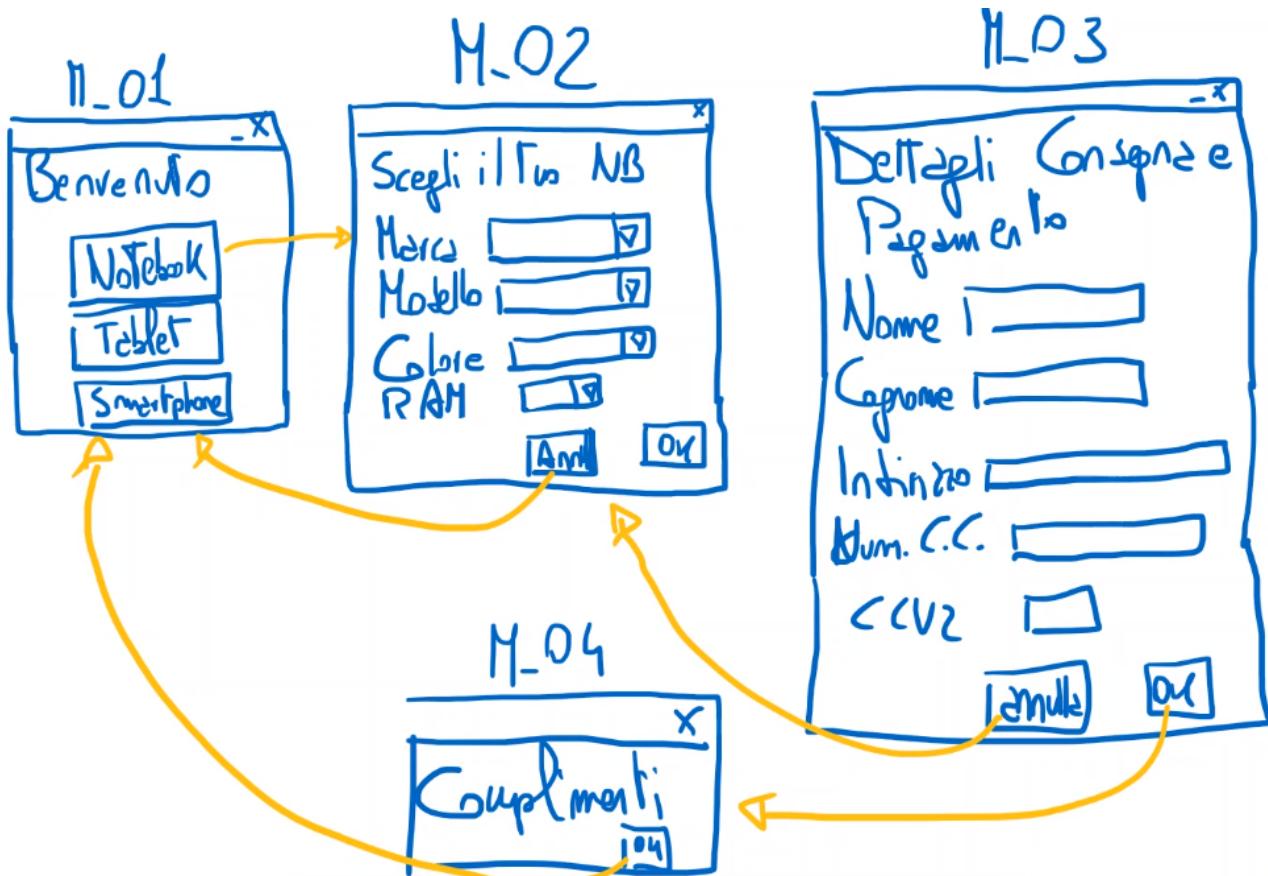
```

Class A{
    void
    foo(){
        B b = new B();
        b.bar();
    }
}
  
```

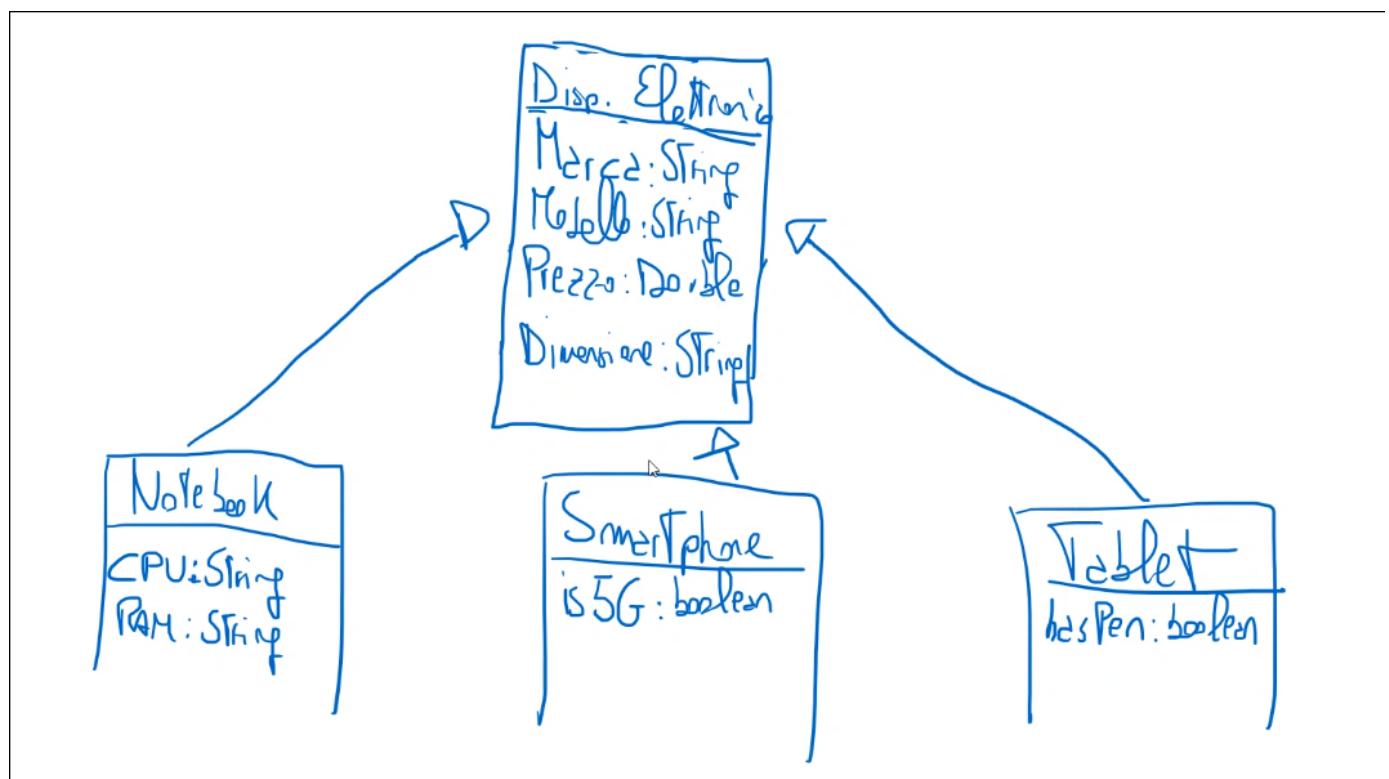


Lezione del 22-12 (esercitazione)

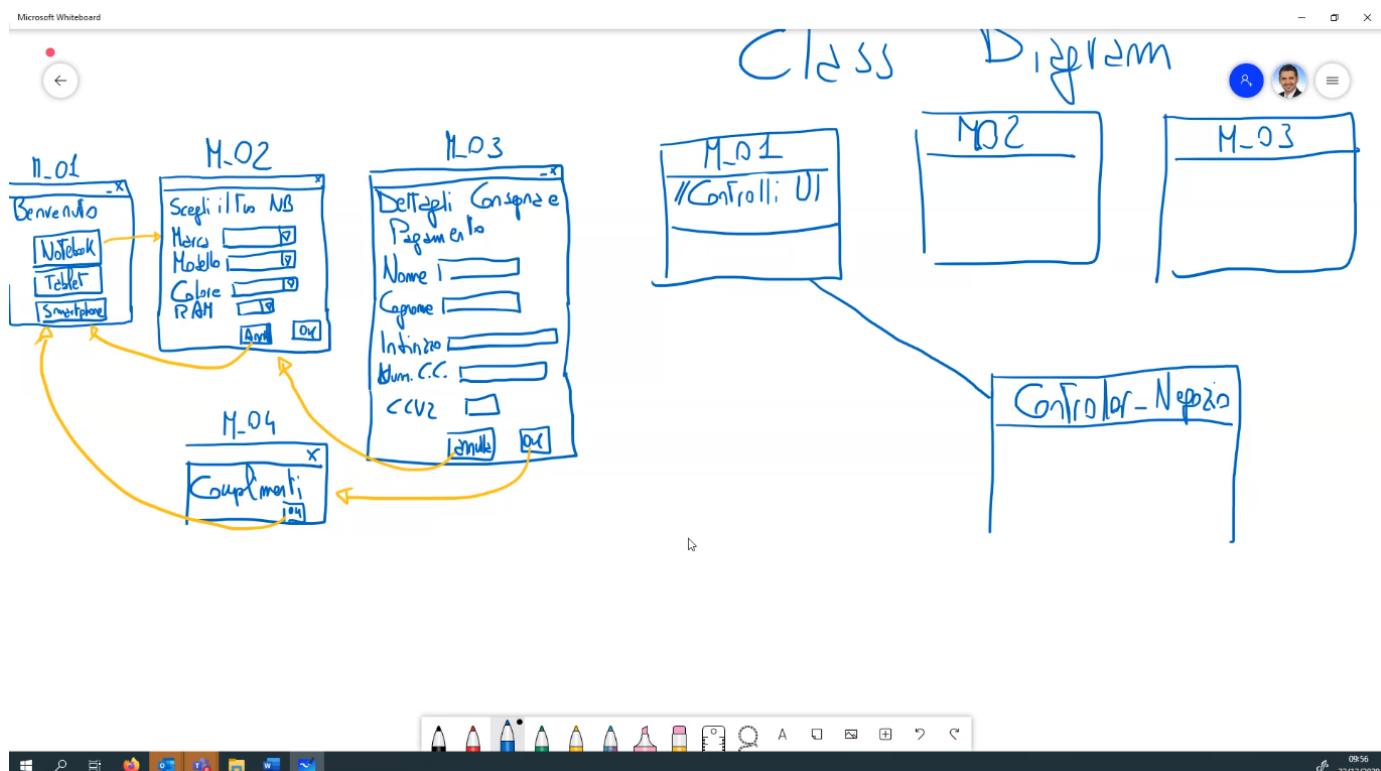
Lezione del 22-12 (esercitazione)



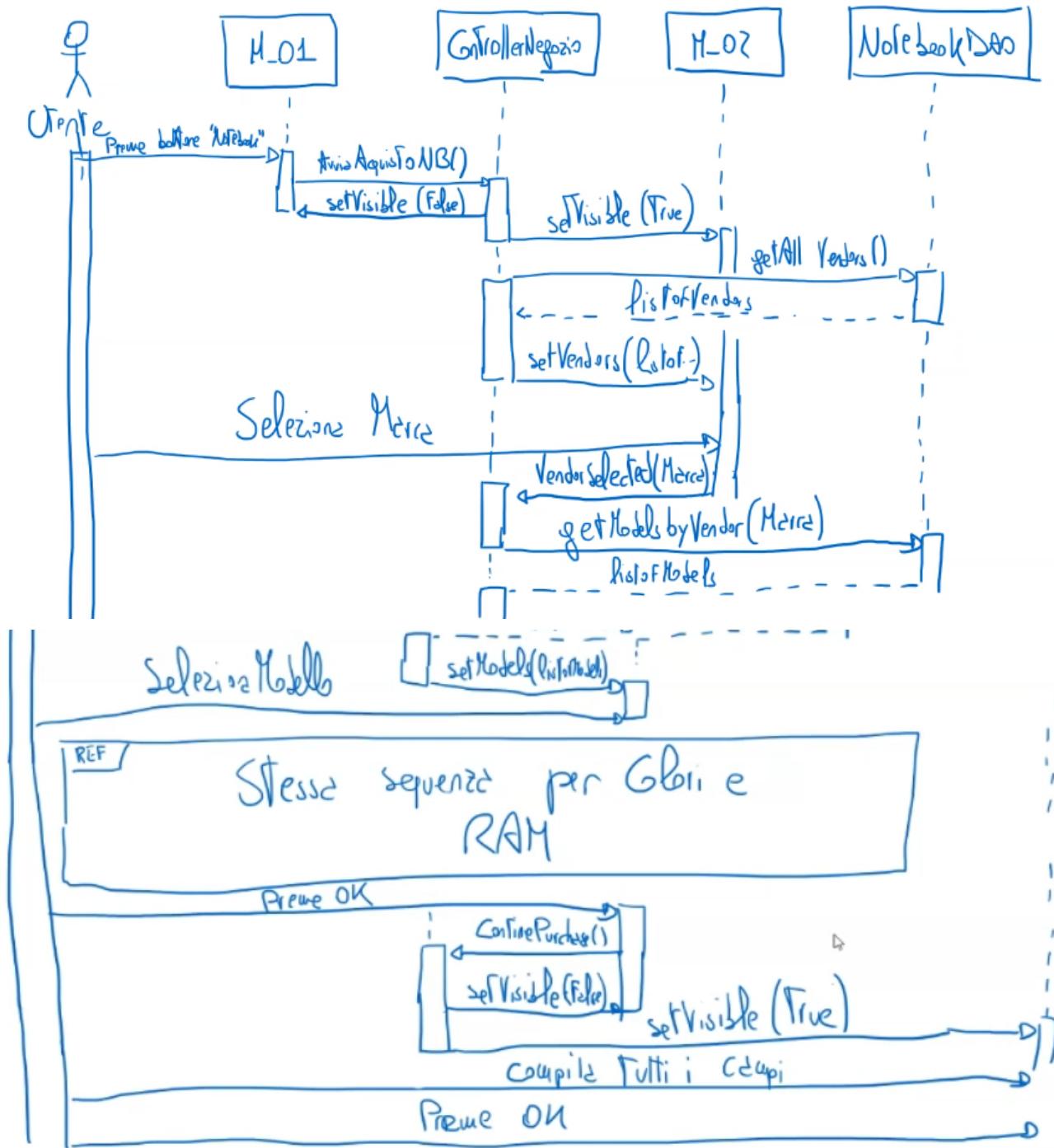
Dato questo scenario immaginiamo il class diagram



I class diagram si differenziano fortemente in base alle scelte che si compiono durante il suo sviluppo



Sequence diagram



Il sequence completo è :

