Riassunto del corso di Programmazione orientata agli oggetti

Alessandro Gerotto

Universitá degli studi di Udine

Docenti *Giorgio Brajnik*

Anno accademico *2022-23*

La programmazione orientata agli oggetti **2**
Definizione “Orientato agli oggetti” 2
Incapsulamento **2**
Perché? 3
Esempio nel mondo reale 3
Esempio nel codice 3
Attenzione! 4
Modificatori di accesso 4
Ereditarietá **4**
Usi e vantaggi dell’ereditarietá 5
Specializzazione (sottotipi) 5
Ridefinizione (overriding) 5
Estensione 5
Riutilizzo del codice 6
Vincoli posti dalla programmazione basata sull' ereditarietà 6
Unicitá 6
Staticitá 6
Visibilità 6
Polimorfismo **6**
Polimorfismo per inclusione (method override) 6
Polimorfismo ad hoc (method overloading) 7
L’astrazione e i suoi livelli **8**
Astrazione per specifica 8
Astrazione via specifica per le classi 9
Astrazione via specifica per i metodi (procedurale) 9
Come Java usa la memoria? **10**
Uguaglianza per valori o per identità 10
Passaggio di parametri ai metodi 10
Test di unità **11**
Junit e i suoi componenti 11
Esecuzione del test 1
Annotazioni precedute da @ 12
Dove dovrebbero stare i test nel sorgente java? 13
Esempio 13
Gestione degli errori **13**
Affidabilità vs robustezza 14
Funzioni parziali 14
Trasformare funzioni parziali in totali 14
1. Aggiungendo nuovi output al programma 15
2. Usando try, catch, throw 15
3. Asserzioni 15
Uso appropriato della asserzioni 15
Differenza tra asserzioni ed eccezioni 15
ADT Mutabili e immutabili **15**
Classi immutabili 15
Costruttore 16
Getters 17
Producers (per ADT immutabili) 17
Mutators (per ADT mutabili) 17
Iteratori in Java **17**
Metodi supportati 17

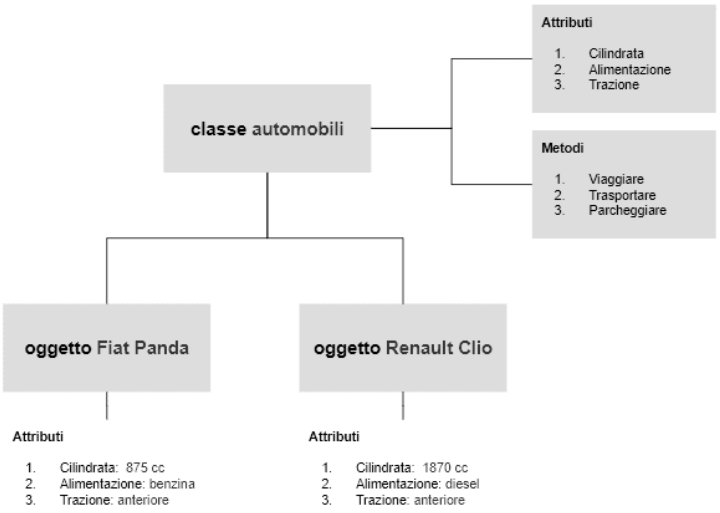
Inizializzazione 17
Esempio 17
Interfacce, classi astratte e metodi astratti **18**
Interfacce 18
Classi astratte 18
Classi astratte vs interfacce 18
Le classi wrapper **19**
Boxing e unboxing 19
Autoboxing 19
Tipi reali e apparenti **20**
Enumerazioni **20**
Esempio 20
Linee guida per lo sviluppo orientato agli oggetti: principi Solid **21**
S - Single Responsibility 21
O - Open/Closed 23
L - Liskov Substitution 24
I - Interface Segregation 24
D - Dependency Inversion 24
Design pattern **25**
Factory method Design Pattern 25
Funzionamento 26
Vantaggi 26
Esempio 26
Builder Design Pattern 26
Vantaggi 26
Procedimento di uso 26
Esempio 26
Observer Design Pattern 28
Decorator Design Pattern 28
Struttura 28
Esempio 29
Lambda calcolo... **30**
...Come filtro 30

La programmazione orientata agli oggetti

La programmazione orientata agli oggetti è un paradigma di programmazione che prevede di **raggruppare** in un'unica entità (la classe) sia le **strutture dati** che le **procedure** che operano su di esse, creando un **oggetto** (istanza della classe) dotato di proprietà (dati) e metodi (procedure) che operano sui dati dell'oggetto stesso. La differenza tra classe e oggetto è la stessa che c'è tra tipo di dato e dato.

Si è detto che un oggetto è definito da:

- **Attributi**: le sue **caratteristiche e proprietà fisiche** utili a definire il suo stato (variabili e costanti);
- **Metodi**: i **comportamenti ammissibili**, cioè le funzionalità dell'oggetto;



Definizione "Orientato agli oggetti"

Un linguaggio di programmazione è definito **orientato ad oggetti** quando permette di implementare:

Incapsulamento

Separazione dell'interfaccia di una classe dalla corrispondente implementazione, in modo che i client di un oggetto di quella classe possano utilizzare la prima, ma non la seconda;

Eredità

Permette di definire delle classi a partire da altre già definite;

Polimorfismo

Significa che un oggetto può avere molte forme. Il polimorfismo in Java ha due tipi:

Incapsulamento

É un meccanismo del linguaggio di programmazione atto a **limitare l'accesso diretto** agli elementi dell'oggetto. Grazie all'incapsulamento, è possibile mantenere tutti gli **attributi** all'interno delle classi come **privati**. Per potervi accedere si useranno dei **metodi get** (accessor methods) e **set** (mutator methods).

Perché?

Quando il codice sorgente di un programma è suddiviso in blocchi modulari in modo da tener conto dei principi dell'occultamento delle informazioni, le operazioni di modifica sono molto più facili perché i cambiamenti sono **localizzati** e non globali.

Esempio nel mondo reale

Un esempio basato sulla realtá é quello di un cellulare: gli utenti sanno usare un telefono, ma ne ignorano il funzionamento interno. Evidentemente per usare un telefono non serve una laurea in ingegneria informatica, ma basta usare le sua **interfaccia pubblica** (costituita da monitor, altoparlante, microfono ...).

Esempio nel codice

```
public class Persona {

   private String nome;
   private String cognome;

   Persona(String nome, String cognome) {
      this.nome = nome;
      this.cognome = cognome;
   }

   // Interfaccia pubblica
   public String getNome() { return nome; }
   public String getCognome() { return cognome; }

}
```

Attenzione!

La pratica di **includere automaticamente** un getter e un setter per ogni singolo membro è **sbagliata** e bisognerebbe sempre evitare. Questo perché farlo **espone l'implementazione** della classe al mondo esterno **violando l'occultamento/astrazione** delle informazioni. In linea di massima è buona pratica tenere **tutto privato** e rendere disponibile esternamente solo lo stretto necessario.

Modificatori di accesso

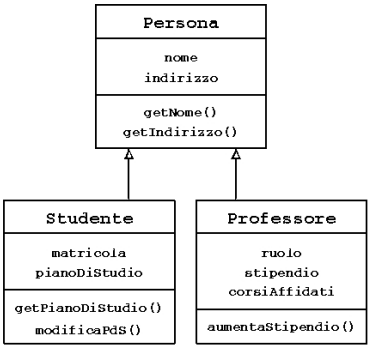
	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
`public`	+	+	+	+	+
`protected`	+	+	+	+	
no modifier	+	+	+		
`private`	+				

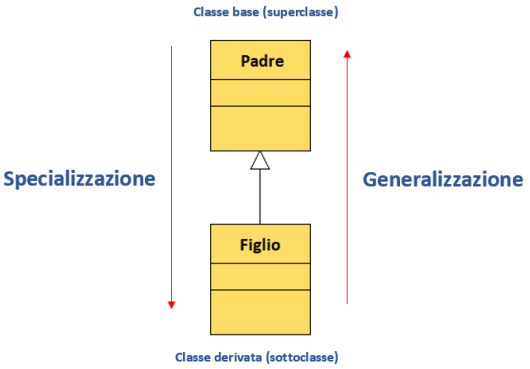
Ereditarietá

Consiste in una **relazione** tra due classi: se la classe B (classe figlia o derivata) **eredita** dalla classe A (classe padre o base), si dice che B è una **sottoclasse** di A e che A è una **superclasse** di B. L'ereditarietà è una relazione di **generalizzazione/specializzazione**: la superclasse definisce un **concetto generale** e la **sottoclasse ne è una variante specifica**. Una classe B dichiarata come sottoclasse di una superclasse A:

- eredita (ha implicitamente) tutte le variabili di istanza e tutti i metodi di A;
- può avere (non necessariamente) variabili o metodi aggiuntivi;
- può ridefinire i metodi ereditati da A attraverso l'overriding.

In generale, l'uso dell'ereditarietà dà luogo a una **gerarchia** di classi. In java (linguaggio con **ereditarietà singola**), si ha un albero se esiste una superclasse "radice" di cui tutte le altre classi sono direttamente o indirettamente sottoclassi (come la classe Object nel caso di Java) o a una foresta altrimenti;





Usi e vantaggi dell'ereditarietá

Grazie all'ereditarietá si ottengono diversi vantaggi:

Specializzazione (sottotipi)

Consiste nella possibilità di creare versioni "specializzate" (**sottotipi**) di classi già esistenti. Quando si definisce una sottoclasse bisogna sempre fare attenzione a rispettare il principio di sostituibilità di Liskov.

Ridefinizione (overriding)

Consiste nel modificare il modo in cui è implementata una funzionalità (metodo) ereditata da un'altra classe. A fronte di overriding, lo stesso metodo avrà un **comportamento diverso** se invocato sugli oggetti della superclasse o in quelli della sottoclasse. Ad esempio, data una classe Quadrilatero che definisce alcuni comportamenti generali, la sottoclasse Rettangolo potrebbe ridefinire quei metodi di Quadrilatero che possono essere reimplementati in maniera più specifica tenendo conto delle specificità dei rettangoli (per esempio, il calcolo del perimetro potrebbe essere riscritto nella classe Rettangolo come doppio della somma della lunghezza delle due basi, anziché come semplice somma delle lunghezze dei lati).

Estensione

Consiste nel fornire ad una classe dati o funzionalità **aggiuntive**. A differenza del caso della specializzazione prima esposto, con l'estensione nuovi dati o funzionalità sono aggiunti alla classe ereditata, accessibili ed utilizzabili da tutte le istanze della classe. L'estensione viene usata spesso quando non è possibile o conveniente aggiungere nuove funzionalità alla classe base. La stessa operazione può essere eseguita anche a livello di oggetto (anziché di classe) usando i decorator pattern.

Riutilizzo del codice

Implica che tutto il codice usato per manipolare oggetti della superclasse è anche in grado di **manipolare** oggetti della **sottoclasse**. Per esempio, un programma che rappresenta graficamente oggetti di classe Quadrilatero non avrebbe bisogno di alcuna modifica per trattare analogamente anche oggetti di una eventuale classe Rettangolo.

Vincoli posti dalla programmazione basata sull' ereditarietà

Un uso massiccio della tecnica dell'ereditarietà può avere qualche controindicazione e porre alcuni vincoli. Supponiamo di avere una classe Persona che contiene come dati nome, indirizzo, numero di telefono, età e sesso. Possiamo definire una sottoclasse di Persona, chiamata Studente, che contiene le medie dei voti ed i corsi frequentati, ed un'altra sottoclasse di Persona, chiamata Impiegato, che contiene il titolo di studio, la mansione svolta ed il salario.

Unicitá

Una classe può ereditare soltanto da **una** classe base. Quindi, nell'esempio sopra riportato, un'istanza di Persona può essere o Studente o Impiegato, non entrambi contemporaneamente.

Staticitá

La gerarchia dell'ereditarietà di un oggetto viene "congelata" nel momento in cui l'oggetto viene istanziato e **non può più essere modificata** successivamente. Per esempio, un oggetto della classe Studente non può diventare un oggetto Impiegato mantenendo le caratteristiche della sua classe base Persona

Visibilità

Quando un programma "client" ha accesso ad un oggetto, di solito ha accesso anche a tutti i dati di un oggetto appartenente alla classe base. Anche se la classe base non è di tipo "pubblico", il programma client può creare oggetti sul suo tipo. Per fare in modo che una funzione possa leggere il valore della media di uno Studente bisogna dare a questa funzione la possibilità di accedere anche a tutti i dati personali memorizzati nella classe base Persona.

Polimorfismo

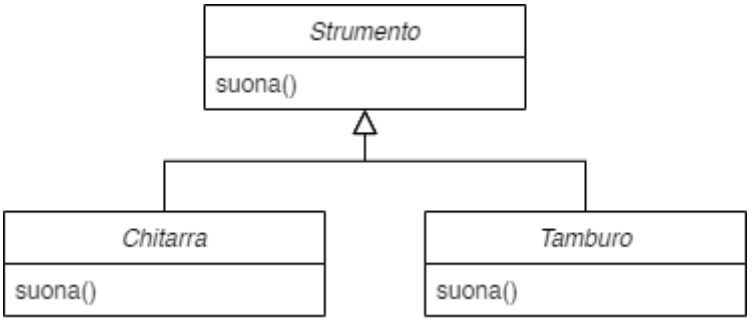
Il termine polimorfismo si riferisce al fatto che un'espressione il cui tipo sia descritto da una classe A può assumere valori di un qualunque tipo descritto da una classe B sottoclasse di A (polimorfismo per inclusione);

Polimorfismo per inclusione (method override)

Questa forma di polimorfismo è strettamente **legata** al concetto di **ereditarietà** e consiste nella possibilità che una sottoclasse A di una classe base B **ridefinisca** uno dei metodi della classe padre e che quindi quando verrà utilizzata un'istanza della classe A le invocazioni al metodo ridefinito (spesso detto overridden) eseguiranno il codice definito nella classe figlia.

Ad esempio, si suppone di avere una gerarchia in cui le classi Cane e Gatto discendono dalla superclasse Animale. Quest'ultima definisce un metodo cosaMangia(), le cui specifiche sono: Restituisce una stringa che identifica il nome dell'alimento tipico dell'animale. I due metodi cosaMangia() definiti nelle classi Cane e Gatto si sostituiscono a quello che ereditano da Animale e, rispettivamente, restituiscono due risultati diversi a seconda del tipo effettivo dell'oggetto su cui viene richiamato il metodo.

Ricordiamo che per il principio di sostituzione di Liskov le istanze della classe A dovranno poter essere utilizzate in **ogni** espressione che preveda l'utilizzo di una istanza della classe B e a queste espressioni si applica dunque la definizione di espressione polimorfa data all'inizio di questa pagina.



Polimorfismo ad hoc (method overloading)

È una funzionalità di java che consente a **metodi diversi** di avere lo **stesso nome**, ma **firme diverse** in cui la firma può differire per il **numero di parametri** di input o il **tipo di parametri** di input o una **combinazione** di entrambi. Java supporta l'overloading attraverso due meccanismi:

1. **Stesso nome** per più metodi **all'interno di una classe**;
2. **Stesso nome** per più metodi in **classi diverse**.

```
public class C {
   public double power (int i, long c)  { … } // 1
   public double power (long x, int c)  { … } // 2
   public double power (long x, long y) { … } // 3
} …

C cc = new C();
int x; long y; double z;

cc.power (x,y) // x intero, y long → esegue il numero 1;
cc.power (x,x) // errore perchè ci potrebbe essere più di una assegnazione ammissibile
```

Per scegliere il metodo **all'interno** di una classe, si guardano i **tipi apparenti** mentre per **scegliere la classe** dove andare a prendere il metodo si guardano i **tipi reali**.

```
public class C {
   public double power (int i, long c)  { … } // 1
   public double power (long x, int c)  { … } // 2
   public double power (long x, long y) { … } // 3
}

public class D extends C {
   public double power (int i, long c)  { … } // 4
   public double power (long x, int c)  { … } // 5
   public double power (long x, long y) { … } // 6
} …

C cc = new C();
D dd = new D();
C cd = new D();

int x; long y; double z;

cc.power (x,y); // esegue il metodo 1
dd.power (x,y); // esegue il metodo 4
cd.power (x,y); // sceglie la classe che corrisponde al tipo reale → cd: tipo apparente C,
reale D. Di conseguenza si prende i metodi del tipo reale (quelli in D)
```

L'astrazione e i suoi livelli

L'astrazione, è l'applicazione del **metodo logico di astrazione** nella strutturazione della descrizione dei sistemi informatici complessi, per facilitarne la progettazione e manutenzione o la stessa comprensione. La pratica consiste nel presentare il sistema in maniera **ridotta ai soli dettagli** considerati essenziali all'interesse specifico, ad esempio **raggruppando** il codice in una funzione, in una classe o nell'intera applicazione. Ci sono tre livelli: funzionale, dei dati e di sistema:

- **Astrazione funzionale:**

  Viene usata ogni volta che si **definisce un metodo** (attraverso un metodo si porta all'interno di un "applicazione" un concetto dinamico, sinonimo di azione, funzione). Definendo un metodo, esso potrà venir richiamato senza badare all'implementazione dello stesso;

- **Astrazione dei dati:**

  Viene usata ogni volta quando si **definisce una classe**: in essa si raccolgono solo caratteristiche e funzionalità essenziali degli oggetti che essa deve definire nel contesto in cui si trova;

- **Astrazione del sistema:**

  Viene usata ogni volta che si **definisce un'applicazione nei termini delle classi essenziali** che devono soddisfare gli scopi dell'applicazione stessa.

Astrazione per specifica

È molto comoda perchè permette una **modifica più semplice** e rende l'**implementazione dei metodi/classi più intuitiva** dato che nella mission/contratto c'è scritto tutto ciò che serve per poter sviluppare il body. Questo permette di:

1. **Isolare il problema** alla sola definizione di quel body;
2. **Delegare l'implementazione** a qualcun altro (dato che tutto ciò che deve sapere è scritto nella specifica);
3. **Rimandare il problema** in futuro e occuparsi di altro.

Astrazione via specifica per le classi

Viene fatta usando una **mission**, ossia una descrizione a parole (astrazione) che consente di capire **cosa fa** un una classe senza essere costretti a capirne l'implementazione.

```
public class Arrays {
   /**
   * The mission of this class is to prove of standalones procedures that can
   * be useful for manipulating arrays of integers.
   **/
```

Astrazione via specifica per i metodi (procedurale)

Viene fatta usando i **contratti**, ossia una descrizione a parole (astrazione) che consente di capire **cosa fa** un metodo senza essere costretti a capirne l'implementazione. Un contratto è formato da:

- **Pre-condizione**: dichiara i vincoli che devono valere prima della invocazione del metodo (se i vincoli non sono soddisfatti allora il contratto non vale);
- **Post-condizione** dichiara quali sono le proprietà che devono valere al termine dell'esecuzione del metodo (nell'ipotesi che la pre-condizione sia valida).

```
   /**
   * Replace each element of the array that is greater than m with m
   * PRE: a ≠ null, a.length ≥ 1, a è ordinato in ordine crescente
   * POST: @modifies a ∀i indice di a, se a[i]>m allora a'[i]=m
   **/
private static void boundArray (int[] a, int m) { … }
```

Come Java usa la memoria?

In Java, le variabili sono memorizzate in questo modo:

- Le variabili **locali** come primitive e riferimenti a oggetti vengono create nello **stack**.
- Gli **oggetti** sono nella **heap** (mucchio).

Uguaglianza per valori o per identità

Quando si confronta un elemento a con un elemento b, si utilizza la sintassi a == b. Viene interpretato in due modi diversi:

- se a e b sono **tipi base**: ritorna vero se sono lo stesso valore (**uguaglianza per valore**);
- se a e b sono **oggetti**: ritorna vero se puntano alla stessa cella di memoria nello heap (**uguaglianza per identità**);

La stessa cosa vale anche per il metodo a.equals(b): tale metodo verifica un **uguaglianza per identità**.

Passaggio di parametri ai metodi

Ci sono due modi per passare i parametri:

1. Per **valore** (nel caso di tipi base): il metodo chiamato crea una propria **copia** dei valori degli argomenti e quindi la utilizza. Poiché stiamo lavorando con una copia, ciò non influisce in alcun modo sul parametro originale.
2. Per **riferimento** (nel caso di oggetti): il metodo chiamato fa riferimento al **valore originale**. Pertanto, le modifiche apportate al metodo chiamato si rifletteranno anche nel valore originale.

Java **passa sempre i parametri per valore.** Ad esempio:

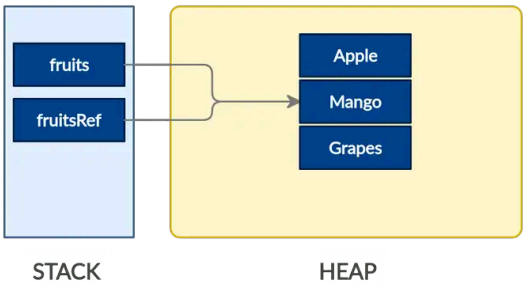
```
public class ParamPassing {

   public static void main(String args) {
       int data = 10;

       print("Data before calling method: " + data);  // Stampa 10
       processData(data);
       print("Data after calling method: " + data);    // Stampa 10
   }

   private static void processData(int data) {
       data = data + 1;
   }
}
```

Il metodo processData funziona con una **copia** di data. Pertanto, nei dati iniziali non ci sono stati cambiamenti. Invece:

```
public class ObjectPassing {

   public static void main(String args) {
       List<String> fruits = new ArrayList<>(Arrays.asList("Apple", "Mango", "Grapes"));
       print((Arrays.toString(fruits.toArray()))); // [Apple, Mango, Grapes]
       processData(friuts);
       print((Arrays.toString(fruits.toArray()))); // [Apple, Mango, Grapes, Orange]
   }

   private static void processData(List<String> fruitsRef) {
       fruitsRef.add("Orange");
   }
}
```

Cosa sta succedendo qui? Se Java sta passando i parametri per valore, perché l'elenco originale è stato modificato? Sembra che Java non passi i parametri per valore dopo tutto?

No, è sbagliato. Ripeti dopo di me: "**Java passa sempre i parametri per valore**". Per capirlo, diamo un'occhiata al diagramma seguente:

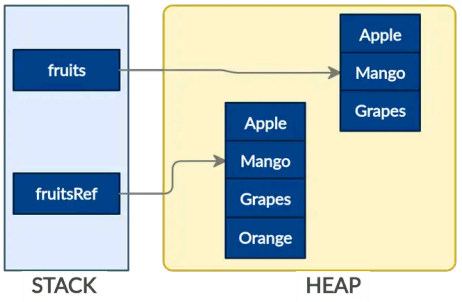


Nel programma sopra, la lista fruits passata al metodo processData (fruitsRef) è una copia del parametro fruits e di conseguenza fruits e fruitsRef sono due **link diversi** messi nello **stack** che **puntano alla stessa cosa** nella **heap**. Di conseguenza, qualsiasi modifica apportata utilizzando uno di questi collegamenti influisce sull'oggetto.

Vediamo ora un controesempio:

```
public class ObjectPassing {

   public static void main(String args) {
       List<String> fruits = new ArrayList<>(Arrays.asList("Apple", "Mango", "Grapes"));
       System.out.println((Arrays.toString(fruits.toArray()))); [Apple, Mango, Grapes]
       processData(friuts);
       System.out.println((Arrays.toString(fruits.toArray()))); // [Apple, Mango, Grapes]
   }

   private static void processData(List<String> fruitsRef) {
       List<String> fruits = new ArrayList<>(List<String> fruitsRef);
       fruitsRef.add("Orange");
   }
}
```

In questo caso, per cambiare il collegamento fruitRef abbiamo usato l'operatore new. Adesso fruitRef **punta a un nuovo oggetto** e quindi qualsiasi modifica apportata ad esso **non influirà** sull'oggetto elenco di frutta originale.



Quindi Java passa sempre i parametri per valore. Tuttavia, dobbiamo fare attenzione quando passiamo i riferimenti agli oggetti.

Test di unità

L'**Unità** è la parte più **piccola** del software che si intende testare (funzioni, metodi, classi, package e interi sistemi). Il testing di unità può essere svolto in modalità:

- **Black box**, se è nota solo la specifica dell'unità (ingressi/uscite/funzione realizzata);
- **White box** se si conosce anche come l'unità è realizzata e si vuole sfruttare tale conoscenza per valutare anche la correttezza parziale dell'unità.

Junit e i suoi componenti

JUnit è un **framework** (archivio .jar contenente una collezione di classi) che permette la scrittura di test in maniera ripetibile. Un test junit è composto da:

- **Classi di test**: una classe di test Junit deve estendere (ereditare da) una classe denominata TestCase della libreria. Una classe di test, contiene anche:
  - un metodo **setup()**, eseguito prima di ogni test (utile per settare precondizioni comuni a più casi di test);
  - un metodo **teardown()**, eseguito dopo ogni caso di test (utile per resettare le postcondizioni);
- **Metodi di test**: ogni metodo di test deve essere preceduto da @test. Sono potenzialmente infiniti. Struttura:
  - **Inizializzazione precondizioni**;
  - **Inserimento valori di input**;
  - **Codice di test**;
  - **Controllo** di espressioni booleani che devono risultare vere se il test dà esito positivo, ovvero se i dati di uscita e/o le postcondizioni riscontrati sono diversi da quelli attesi

Esecuzione del test

La **traduzione** del codice java di un test in codice junit viene fatta secondo 5 step:

1. **Precondizioni**: Tramite il metodo setup vengono eseguite delle operazioni mirante a far diventare vere le precondizioni. Al termine del metodo vengono poste delle asserzioni che valutano le precondizioni: se non fossero verificate, i test non vengono eseguiti;
2. **Input**: tramite settaggi diretti di attributi oppure chiamate a metodi set;
3. **Test**: esecuzione del metodo da testare con gli eventuali ulteriori parametri di input relativi a quel caso di test;
4. **Output**: controllo di espressioni booleane sulla coincidenza dei valori di output ottenuti con quelli osservati;
5. **Postcondizioni**: valutazione di espressioni booleane (asserzioni) relative alle postcondizioni.

Annotazioni precedute da @

- Metodi preceduti da **@BeforeEach** vengono eseguiti prima di ogni test;
- Metodi preceduti da **@Test** rappresentano metodi di test;
- Metodi preceduti da **@DisplayName** vengono usati per definire il nome del test mostrato all'utente;
- Metodi preceduti da **@RepeatedTest** vengono eseguiti più volte.

Dove dovrebbero stare i test nel sorgente java?

Tipicamente, gli unit test vengono creati in una **cartella di origine separata** per mantenere il codice di test separato dal codice reale. La convenzione standard degli strumenti di compilazione Maven e Gradle prevede l'utilizzo di:

- src/main/java per le classi;
- src/test/java per i test;

Esempio

Esempio di test per il metodo swap contenuto nella classe main:

```
public class Calculator {
    public int multiply(int a, int b) {
        return a * b;
    }
}
```

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.Test;

class CalculatorTest {

    Calculator calculator;

    @BeforeEach
    void setUp() {
        calculator = new Calculator();
    }

    @Test
    @DisplayName("Simple multiplication should work")
    void testMultiply() {
        assertEquals(20, calculator.multiply(4, 5),
                "Regular multiplication should work");
    }

    @RepeatedTest(5)
    @DisplayName("Ensure correct handling of zero")
    void testMultiplyWithZero() {
        assertEquals(0, calculator.multiply(0, 5), "Multiple with zero should be zero");
        assertEquals(0, calculator.multiply(5, 0), "Multiple with zero should be zero");
    }
}
```

Gestione degli errori

Ogni programma java, durante la sua esecuzione, è caratterizzato da uno **stato** (normale o errato) che comprende i valori delle variabili in quel particolare istante e un **flusso di controllo**. Durante un ciclo di vita di un programma, possono verificarsi degli **errori** (difetti, computi dal programmatore). Tali difetti però, **non sempre emergono** cambiando lo stato di esecuzione del programma (da normale a errato). Una manifestazione di un errore viene definita come **failure** (malfunzionamento). Tutti i tipi di eccezioni ed errori sono sottoclassi della classe **Throwable**, che è la classe base della gerarchia (a sua volta sottoclasse di Object). Essa si divide in due diramazioni:

- **Exception**: classe viene utilizzata per condizioni eccezionali che i programmi utente dovrebbero rilevare. NullPointerException è un esempio di tale eccezione.
- **Error**: viene utilizzato dal sistema di runtime Java (JVM) per indicare errori che hanno a che fare con l'ambiente di runtime stesso (JRE). StackOverflowError è un esempio di tale errore.

Le exceptions possono essere **user-defined exception** oppure **built-in exception**. Queste ultime vengono categorizzate in:

- **Checked exceptions**: chiamate durante la compilazione, ogni metodo che le usa deve includere nella definizione;
- **Unchecked exceptions**: anche se non dichiarate, il programma non darebbe un errore di compilazione.

Affidabilità vs robustezza

L'obiettivo di un bravo programmatore è quello di scrivere del codice **robusto**, ossia un codice che, nonostante la presenza di errori (a.e. input non validi), funzioni bene in ogni circostanza. In certe circostanze, è necessario garantire che il codice prodotto sia **affidabile** e che quindi garantisca la totale assenza di **failure**.

Funzioni parziali

Una funzione viene definita **parziale** quando **per alcuni input non è definita**. Si pensi ad esempio alla funzione fattoriale: essa ha come dominio i numeri naturali (come si può leggere dal contratto del metodo qui sotto) e di conseguenza è una funzione parziale perché non è definita per i numeri negativi:

```
/**
* Compute factorial of n
* @param n: require to be ⩾ 0
* @return n!
**/
public static int factorial (int n) {...}
```

Trasformare funzioni parziali in totali

Programmi che calcolano **funzioni parziali** sono quindi sempre **poco robusti**. Per trasformare funzioni parziali in totali:

1. Aggiungendo nuovi output al programma

Si aggiunge una condizione nel metodo chiamante che notifica se l'input non soddisfa le precondizioni del metodo chiamato

2. Usando try, catch, throw

```
/** Compute factorial of n
* @param n
* @return n!
* @throw NonPositiveArgumentException if n < 0
**/
public static int factorial (int number)  {
   if (number < 0) { throw new IllegalArgumentException("Only Positive Numbers!");
   } else { // Do factorial (number) }
}
```

```
// Metodo chiamante
public static void printFactorial (int n) {

   try {
      System.out.println(factorial.factorial(n));
   } catch (IllegalArgumentException iae) {
      System.out.println("Input < 0");
   }
}
```

3. Asserzioni

Una asserzione è un'espressione booleana che deve essere "true" se e solo se il codice sta **funzionando correttamente**. Se l'asserzione risulta falsa, viene **segnalato l'errore** (opzionalmente stampando un messaggio o anche invocando un metodo (attenzione!)) e bisogna **correggere** il codice. Il controllo delle asserzioni può essere **attivato o disattivato** a runtime. Per default il controllo delle asserzioni è **disabilitato**.

```
assert condizione [: messaggio];
```

Uso appropriato della asserzioni

1. Le asserzioni possono essere utilizzate come **precondition** nei **metodi privati** (i metodi pubblici devono fare un controllo esplicito e sollevare un **eccezione**);
2. Le asserzioni **non** vanno usate per **controllare l'input** dell'utente, i **parametri** a linea di comando, etc.;
3. Le asserzioni **possono** sempre essere utilizzate come **postconditions** e **loop invariants**;

Differenza tra asserzioni ed eccezioni

Le **asserzioni** permettono di inserire un meccanismo di "sanity check" utilizzato nella **fase di test** e sviluppo mentre le **eccezioni** vengono usate per gestire malfunzionamenti quando il codice è finito e completato.

- L'**eccezione** dice all'utente del programma che **qualcosa è andato storto**. Si creano eccezioni quando si ha a che fare con **problemi** che **possono insorgere**.
- Una **asserzione violata**, invece, rappresenta un **bug**. Infatti, si scrivono asserzioni per **salvaguardare** cose che si conoscono relativamente al comportamento del programma.

ADT Mutabili e immutabili

Un tipo di dato astratto viene definito **dal suo comportamento**, non dalla sua implementazione interna. La sua specifica dovrebbe contenere la mission dove viene enunciato **cosa la classe sa** e **cosa la classe sa fare**.

Classi immutabili

L'aggettivo "immutabile" indica che quando una classe viene istanziata, se immutabile, non può cambiare il suo stato interno, di conseguenza, essa **non ha uno stato**. Per definire un classe immutabile esistono delle **linee guida**:

1. Tutti gli **attributi** devono essere **private** e **final**;
2. La classe **non** deve avere **metodi setter** o che ne modificano le variabili d'istanza;
3. **Impedire** alle sottoclassi di fare **override dei metodi** (dichiarare final la classe e i metodi);
4. Se le variabili devono essere passate in **input** a **metodi esterni**, bisogna **creare** delle **copie** da restituire all'esterno;
5. Se si vuole usare dei **getter**, bisogna preoccuparsi di **restituire** delle **copie del dato**.

```
public class Poly {
/** This class provides an ADT for polynomials with integer coefficients. Poly is
immutable, unbounded, with non negative exponents. */

   /**
   @return: a new zero poly.
   */
   public Poly(){}

   /**
    @param c: the coefficient
    @param n: the exponent
    @return: a new Poly c*x^n.
    @Throw NegativeExponentException when n<0.
   */
    public Poly(int c, int n) throw NegativeExponentException {}

    /**
    @param p: the poly to be added to this; REQUIRE not null.
    @return a new poly that is this+p
    */
    public Poly add(Poly p){}

    /**
    @param p: the poly to be multiplied to this; REQUIRE not null.
    @return a new poly that is this*p
    */
    public Poly mul(Poly p){}

    /**
    @return the largest exponent in this with non zero coeff
    */
    public int getDegree(){}

    /**
    @param n: an exponent.
    @return the coefficient of the term in this that has exponent n; possibly 0
    */
    public int getCoeff(int n){}
```

Immutabili	Mutabili

class TypeName {
1. private final fields
2. costruttore
3. getters
4. producers
}
	class TypeName {
	1. overview & abstract fields
	2. costruttore
	3. getters
	4. mutators
	}

Costruttore

Un metodo costruttore **inizializza** un **nuovo oggetto istanza della classe**. Ogni classe ha **almeno un** costruttore che viene eseguito quando si crea un oggetto istanza. Una classe può comunque avere **più costruttori**, distinti tra loro da una diversa **combinazione di argomenti**. Il **nome** è invece sempre lo **stesso**. Per invocare il costruttore si usa l'operatore **new** che crea l'area di memoria dove sarà collocato lo stato dell'oggetto mentre il **costruttore** inizializza lo **stato iniziale** dell'oggetto. Esempio:

```
class Persone {
   private String nome;
   private String cognome;

   // Metodo costruttore
   public Persone(String nome, String cognome) {
      this.nome = nome;
      this.cognome = cognome;
   }
   // Per creare un nuovo oggetto, invoco il costruttore usando l'operatore new:
   Persone andrea = new Persone("Andrea", "Minini");
}
```

Getters

Metodi usati per **ottenere delle informazioni** su oggetti. Non modificano mai il valore astratto degli oggetti. Quando si usa il termine **this** ci si riferisce al particolare oggetto su cui si sta lavorando (conosciuto come il ricevitore).

Producers (per ADT immutabili)

Metodi usati per **fare delle operazioni** su degli oggetti che creano altri oggetti di quel tipo **lasciando inalterato** l'originale.

Mutators (per ADT mutabili)

Metodi che **modificano** gli oggetti. Raramente modificano qualcosa di diverso da **this**. Tipicamente non ritornano niente.

Iteratori in Java

È un oggetto che permette di **scorrere** una collezione di dati.

Metodi supportati

- **next()**: restituisce l'elemento successivo. Se la raccolta non ha elementi, genera NoSuchElementException;
- **hasNext()**: restituisce true o false a seconda se ci sono elementi nella raccolta;
- **remove()**: rimuove l'ultimo elemento restituito dall'iteratore;
- **forEachRemaining()**: esegue l'azione specificata su ciascuno degli elementi rimanenti della raccolta.

Inizializzazione

```
Iterator<tipo_oggetti_itera> nome = oggetto_su_cui_iterare.iterator()
```

Esempio

```
ArrayList<String> persone = new ArrayList<String>();

persone.add(“Luca”);
persone.add(“Marco”);
persone.add(“Anna”);
persone.add(“Paolo”);

Iterator<String> it = persone.iterator();
```

Stampare gli elementi puntati dall’iteratore:

```
print(it.next());   // Stampa Luca
println(it.next()); // Stampa Marco
println(it.next()); // Stampa Anna
println(it.next()); // Stampa Paolo
```

```
while(it.hasNext()) {
   print(it.next());
}
// Stampa [Luca, Marco, Anna, Paolo]
```

Rimuovere un elemento da una struttura dati rimuovendolo dall’iteratore:

```
while(it.hasNext()) {
   if (it.next() == “Anna”) {
      it.remove();
   }
}
```

Interfacce, classi astratte e metodi astratti

Interfacce

Un'interfaccia è un'entità che funge da **schema** (scheletro, prototipo) di una classe. Un'interfaccia può contenere **SOLO:**

- **Metodi public** e **abstract**;
- **Variabili static** e **final**.

Non può quindi contenere costruttori, variabili statiche o di istanza e metodi statici.

Classi astratte

Una classe che ha **almeno un metodo astratto** (senza body e dichiarato con la keyword **abstract**):

```
public abstract sum (int a, int b);
```

è una **classe astratta**.

Le classi astratte non possono mai essere istanziate. Ad esempio, se Animal é una classe astratta estesa dalla classe Cat e dalla classe Dog:

```
Animal animal = new Cat();     // Valido
Animal animal = new Dog();     // Valido
Animal animal = new Animal();  // Non valido
```

Una classe astratta deve avere sempre **almeno un costruttore**.

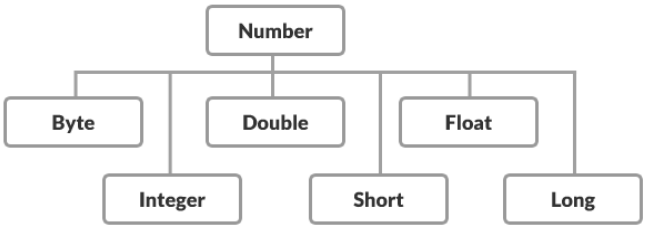
Classi astratte vs interfacce

Presentano due differenze:

- Una **classe astratta** combina **progetto** e **implementazione**, un' **interfaccia** solo **progetto**;
- Una **classe astratta** gode di ereditarietà **singola** (tale classe eredita proprietà da un unico genitore), un'**interfaccia** permette l'ereditarietà **multipla** (una classe può ereditare proprietà da più di un genitore).

Le classi wrapper

Sono dette **classi wrapper** (involucro) le classi che fanno da contenitore ad un tipo di dato primitivo. Un esempio sono le sottoclassi della classe Number (Byte, Short, Integer, Long, Float, Double), Boolean e la classe Character ognuna delle quali può contenere il relativo tipo primitivo.



Sono classi **immutabili** (una volta istanziate non potranno mai cambiare il proprio valore) e sono utili nei casi in cui bisogna usare un tipo di dato **primitivo** laddove è **richiesto un oggetto**.

Boxing e unboxing

Le classi wrapper permettono di fare **boxing** (passaggio da tipo primitivo ad oggetto) e **unboxing** (passaggio da oggetto a tipo primitivo) per poter utilizzare le variabili con i metodi.

Boxing	Unboxing

`int i = 23;`
`Integer x = new Integer(i); //oppure`
`Integer x = Numbers.valueOf(i);`
	`int i = 23;`
	`Integer x = new Integer(i);`
	`int j = x.intValue;`

Autoboxing

In certi casi il compilatore Java fa **autoboxing**, ma bisogna prestare attenzione perché spesso causa problemi.

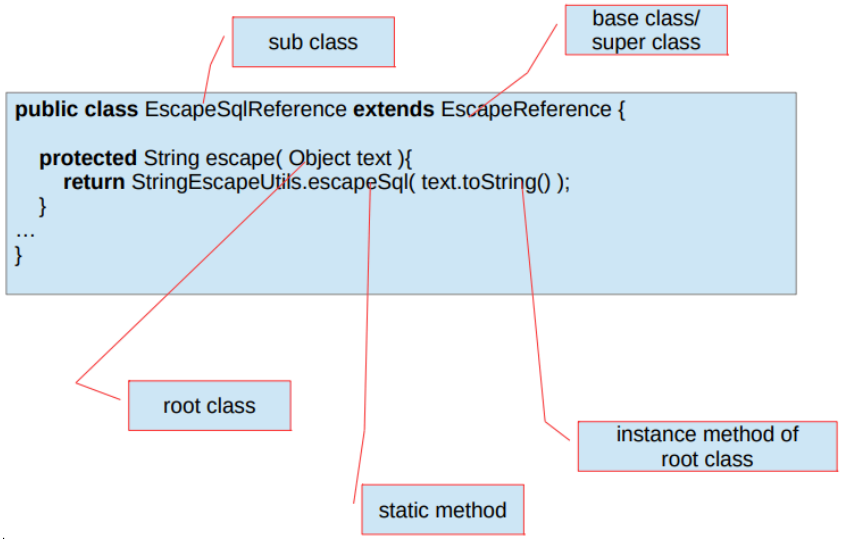
Autoboxing	Boxing

`List<Integer> li = new ArrayList<>();`
 	
`for (int i = 1; i < 50; i++){`
`  li.add(i);`
`}`
	`List<Integer> li = new ArrayList<>();`
	
	`for (int i = 1; i < 50; i++){`
	`  li.add(Integer.valueOf(i));`
	`}`

Tipi reali e apparenti

Il **compilatore** Java deduce il **tipo apparente** momento della compilazione mentre la **JVM** manipola i **tipi reali** al momento dell'**esecuzione.** Ad esempio:

```
List<Student> theBestOnes = new ArrayList<Student>();  // Dove:

// List<Student>:                      apparent type of theBestOnes
// ArrayList<Student>:        real type of theBestOnes
```



Il tipo reale di una espressione è un sottotipo del tipo apparente:

```
int [] a = new int[3];
Object x = a;

// Dove:
// Tipo apparente di a: int[], Tipo apparente di x: Object
// Tipo reale di a: int[], Tipo reale di x: int[]
```

Enumerazioni

Il tipo **enumerato** può essere pensato come un modo per **vincolare** una variabile a poter assumere solo un determinato (dallo sviluppatore) set di valori. Non ha senso utilizzarlo nel caso in cui si debba ricorrere a molti simboli (dato che, oltretutto, porterebbe a usare nomi poco significativi).

Esempio

```
public enum StatoDellaMerce {
   inEsposizione,
   inMagazzino,
   inArrivo
}
```

```
// Usage
StatoDellaMerce stato = StatoDellaMerce.inEsposizione;
```

Linee guida per lo sviluppo orientato agli oggetti: principi Solid

I principi SOLID sono intesi come **linee guida** per lo sviluppo di software **leggibile**, **estendibile** e **manutenibile**, in particolare nel contesto di pratiche di sviluppo agili e fondate sull'identificazione di code smell e sul refactoring.

S - Single Responsibility

Ogni classe dovrebbe avere un unico scopo e non dovrebbe essere composta da funzioni aggiuntive.

Ad esempio, supponiamo di avere una classe AreaCalculator, che contiene dei metodi per calcolare l'area delle figure geometriche (cerchio e quadrato) e ne ritorna la somma. Se volessimo aggiungere la funzione di stampare tale risultato, magari scegliendo se ottenerlo in json o in csv, dovremmo definire due nuovi metodi. Tuttavia l'aggiunta di questi due metodi viola il principio di singola responsabilitá dato che ora la classe oltre a calcolare l'area delle figure, ne stampa anche il risultato.

Per risolvere questo problema, inseriamo i due metodi di stampa in una nuova classe ShapesPrinter, che avrà la singola responsabilitá di stampare il valore dell'area a seconda del formato che si desidera.

```
public class AreaCalculator {
   public int sum(List<Object> shapes) {

      int sum = 0;

      for (int i = 0; i < shapes.size(); i++) {
         Object shape = shapes.get(i);
         if (shape instanceof Square) {
            sum += Math.pow(((Square) shape).getLength(), 2);
         }
         if (shape instanceof Circle) {
            sum += Math.PI * Math.pow(((Circle) shape).getRadius(), 2);
         }
         return sum;
   }
}
```

```
public class ShapesPrinter {

   public String json(int sum) {
      return “{shapesSum: %s}”.formatted(sum);
   }

   public String csv(int sum) {
      return “shapes_sum,%s”.formatted(sum);
   }
}
```

O - Open/Closed

Le classi dovrebbero essere aperte alle estensioni, ma chiuse alle modifiche. In altre parole, per implementare una nuova funzione, non dovrebbe essere necessario scrivere all'interno della classe.

Supponiamo che al programma di prima si voglia aggiungere delle nuove figure, per poterlo fare, bisognerebbe modificare la classe AreaCalculator aggiungendo tante condizioni quante sono il numero di nuove figure. Così facendo, stiamo infrangendo il principio open/close. Per risolvere questo problema, la soluzione é astrarre: si crea una nuova interfaccia chiamata Shape, con, in questo caso, un unico metodo che calcola l'area delle figure. Dopodiché si fa implementare tale interfaccia ad ogni figura geometrica che definirà tale metodo con la propria funzione.

```
public class AreaCalculator {
   public int sum(List<Shape> shapes) {
      int sum = 0;
      for (int i = 0; i < shapes.size(); i++) {
         sum += shapes.get(i).area();
      }
      return sum;
   }
}
```

```
public interface Shape {
   double area();
}
```

```
public class Square implements Shape {

   private final int length;
   public Square(int length) {
      this.length = length;
   }
   public int getLength() {
      return length;
   }
   @Override
   public double area() {
      return Math.pow(getLength(), 2);
   }
}
```

```
public class Circle implements Shape {

   private final int length;
   public Square(int len) {
      this.length = len;
   }
   public int getLength() {
      return length;
   }
   @Override
   public double area() {
      return Math.PI*Math.pow(getRadius(), 2);
   }
}
```

L - Liskov Substitution

Ogni classe derivata dovrebbe essere sostituibile dalla classe genitrice.

Immaginiamo di voler aggiungere una classe NoShape che implementa Shape e ritorna un'exception quando viene chiamato il suo metodo area. Facendo ciò stiamo violando Liskov, dato che NoShape non è sostituibile da Shape.

```
public class NoShape implements Shape {
   @Override
   public double area() {
      throw new IllegalStateException("Cannot calculate area");
   }
}
```

I - Interface Segregation

Le interfacce non dovrebbero forzare le classi ad implementare quello che non possono fare. Le interfacce che contengono molti metodi dovrebbero essere divise in interfacce minori.

Se per esempio volessimo aggiungere una classe per rappresentare una figura tridimensionale, ad esempio la classe Cube, allora per essa potremmo volerne calcolare il volume. Aggiungiamo tale funzione all'interfaccia per mantenere valido il principio open/close. Così facendo peró, infrangiamo il principio di segregazione delle interfacce dato che stiamo forzando anche Square e Circle ad implementare il calcolo del volume (che ovviamente non hanno). La soluzione a questo problema è creare una seconda interfaccia, chiamata ThreeDimensionalShape ed equipaggiarla con un metodo per il calcolo del volume. A questo punto la classe Cube implementerá ThreeDimensionalShape.

D - Dependency Inversion

I moduli di alto livello non devono dipendere da quelli di basso livello, entrambi devono dipendere da astrazioni.

Immaginiamo di voler modificare il codice usando AreaCalculator come nel codice qui sotto. Cosí facendo stiamo violando in primis il principio open/close (una modifica ad AreaCalculator implica di dover modificare anche ShapesPrinter) ma anche il principio di inversione delle dipendenze dato che ora ShapesPrinter dipende dall'istanza concreta areaCalculator.

```
public class ShapesPrinter {

   private AreaCalculator areaCalculator = new AreaCalculator();
   public String json(List<Shape> shapes) {
      return "{shapesSum: %s}".formatted(areaCalculator.sum(shapes));
   }
   public String csv(List<Shape> shapes) {...}
}
```

Per risolvere questo problema, bisogna fare un refactoring del codice di AreaCalculator facendolo implementare un'interfaccia IAreaCalculator che definisca il metodo sum.

```
public interface IAreaCalculator  {
   public int sum(List<Shape> shapes);
}
```

```
public class AreaCalculator implements IAreaCalculator  {

   @Override
   public int sum(List<Shape> shapes) {
      int sum = 0;
      for (int i = 0; i < shapes.size(); i++) {
         sum += shapes.get(i).area();
      }
      return sum;
   }
}
```

E poi utilizzare IAreaCalculator passandola nel costruttore di ShapesPrinter. In questo modo, anziché dipendere dalla classe concreta AreaCalculator, ShapesPrinter dipende dall'interfaccia IAreaCalculator e quindi dall'astrazione di AreaCalculator:

```
public class ShapesPrinter {
   private final IAreaCalculator areaCalculator;
   public ShapesPrinter (IAreaCalculator areaCalculator ) {
      this.areaCalculator = areaCalculator ;
   }
   public String json(List<Shape> shapes) {...}
   public String csv(List<Shape> shapes) {...}
}
```

Questo è utile perché: immaginiamo di voler aggiungere una classe, AreaCalculatorV2 che implementa IAreaCalculator e definisce il metodo sum diversamente da AreaCalculator. Per utilizzare AreaCalculatorV2 basta cambiare il tipo nel main.

```
IAreaCalculator areaCalculator = new AreaCalculator;
IAreaCalculator areaCalculator = new AreaCalculatorV2;
```

Factory method Design Pattern

Una Factory è semplicemente una **funzione wrapper** attorno a un costruttore (possibilmente uno in una classe diversa). La differenza fondamentale è che un modello di metodo factory richiede che l'intero oggetto venga compilato in una singola chiamata di metodo, con tutti i parametri passati su una singola riga. L'oggetto finale verrà restituito.

Funzionamento

Sostituire le chiamate all'operatore new con chiamate ad un factory method che restituiranno degli oggetti chiamati products. Tali oggetti devono implementare la stessa interfaccia o classe base.

Vantaggi

Nel Factory pattern, creiamo l'oggetto senza esporre la logica di creazione al client e ci riferiamo all'oggetto appena creato utilizzando un'interfaccia comune.

Esempio

```
// Factory
static class FruitFactory {
    static Fruit create(name, color, firmness) {
        // Additional logic
        return new Fruit(name, color, firmness);
    }
}

// Usage
Fruit fruit = FruitFactory.create("apple", "red", "crunchy");
```

Builder Design Pattern

Un pattern builder, è un oggetto wrapper attorno a tutti i possibili parametri che potresti voler passare in una chiamata al costruttore. Ciò consente di **utilizzare i metodi setter per creare lentamente** l'elenco dei parametri. Un metodo aggiuntivo su una classe builder è un metodo **build()**, che passa semplicemente l'oggetto builder nel costruttore desiderato e **restituisce il risultato**. IL BUILDER LO USI QUANDO HAI PARAMETRI OPZIONALI

Vantaggi

- Consente di variare la rappresentazione interna di un prodotto.
- Incapsula il codice per la costruzione e la rappresentazione.
- Fornisce il controllo sulle fasi del processo di costruzione.

Procedimento di uso

1. Creare una classe nidificata statica;
2. Copiare tutti gli argomenti della classe esterna *Nome* alla classe *NomeBuilder*;
3. *NomeBuilder* deve avere un costruttore pubblico con tutti gli attributi richiesti come parametri;
4. *NomeBuilder* deve avere metodi per impostare i parametri facoltativi e restituire lo stesso oggetto *NomeBuilder* dopo aver impostato l'attributo facoltativo.
5. Il passaggio finale consiste nel fornire un metodo *build()* nella classe *NomeBuilder* che restituirà l'oggetto necessario al programma client. Per questo abbiamo bisogno di avere un costruttore privato nella classe con la classe *NomeBuilder* come argomento.

Esempio

```
public class Computer {

   // required parameters
   private String HDD;
   private String RAM;

   // optional parameters
   private boolean isGraphicsCardEnabled;
   private boolean isBluetoothEnabled;

   // get methods
   public String getHDD() { return HDD; }
   public String getRAM() { return RAM; }
   public boolean isGraphicsCardEnabled() { return isGraphicsCardEnabled; }
   public boolean isBluetoothEnabled() { return isBluetoothEnabled; }

   // builder method
   private Computer(ComputerBuilder builder) {
      this.HDD=builder.HDD;
      this.RAM=builder.RAM;
      this.isGraphicsCardEnabled=builder.isGraphicsCardEnabled;
      this.isBluetoothEnabled=builder.isBluetoothEnabled;
   }

   // Builder Class
   public static class ComputerBuilder{

      // required parameters
      private String HDD;
      private String RAM;

      // optional parameters
      private boolean isGraphicsCardEnabled;
      private boolean isBluetoothEnabled;

      public ComputerBuilder(String hdd, String ram){
         this.HDD=hdd;
         this.RAM=ram;
      }

      public ComputerBuilder setGraphicsCardEnabled(boolean isGraphicsCardEnabled) {
         this.isGraphicsCardEnabled = isGraphicsCardEnabled;
         return this;
      }

      public ComputerBuilder setBluetoothEnabled(boolean isBluetoothEnabled) {
         this.isBluetoothEnabled = isBluetoothEnabled;
         return this;
      }

      // method that will return the Object needed by client program
      public Computer build(){
         return new Computer(this);
      }
   }
}
```

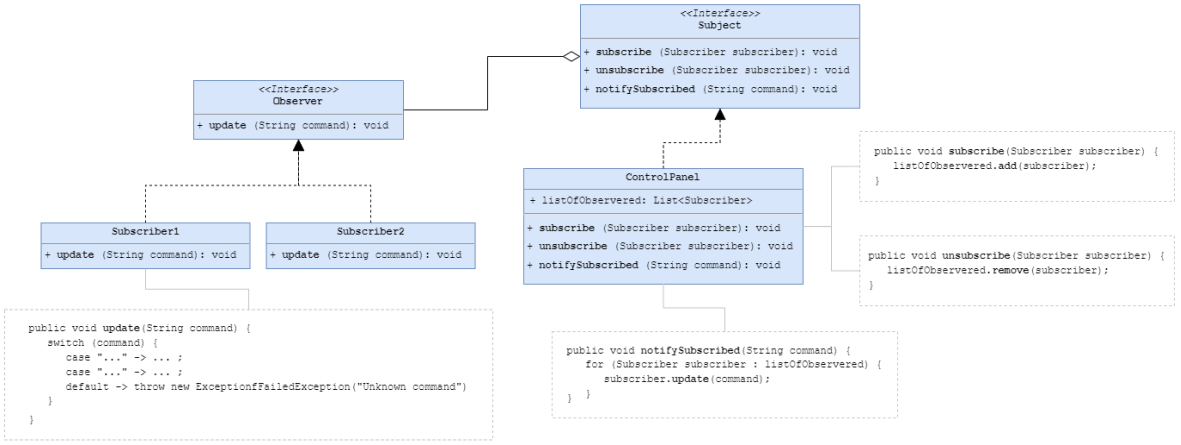
La classe Computer ha **solo metodi getter** e **nessun costruttore pubblico**. Quindi l'unico modo per ottenere un oggetto Computer è attraverso la classe ComputerBuilder. Ecco un esempio di programma di test del modello di builder che mostra come utilizzare la classe Builder per ottenere l'oggetto:

```
public class TestBuilderPattern {

   public static void main(String[] args) {

      // using builder to get the object in a single line of code and
      // without any inconsistent state or arguments management issues
      Computer comp = new Computer.ComputerBuilder(
                      "500 GB", "2 GB").setBluetoothEnabled(true)
                       .setGraphicsCardEnabled(true).build();
   }
}
```

Observer Design Pattern

Il pattern Observer mette a disposizione un **meccanismo di sottoscrizione** alla classe *Subject* in modo che i diversi oggetti *Observers* possano sottoscrivere o annullare la sottoscrizione a un flusso di eventi provenienti da quel publisher. Questo meccanismo è costituito da:

- Una classe Subject (canale youtube) che contiene:
    - Un array per memorizzare un **elenco di riferimenti a oggetti abbonato**;
    - Alcuni **metodi** che consentono di **aggiungere, rimuovere e notificare** gli abbonati da tale elenco.
- Diverse classi Observers (iscritti youtube) che contiene un metodo update che determina il messaggio

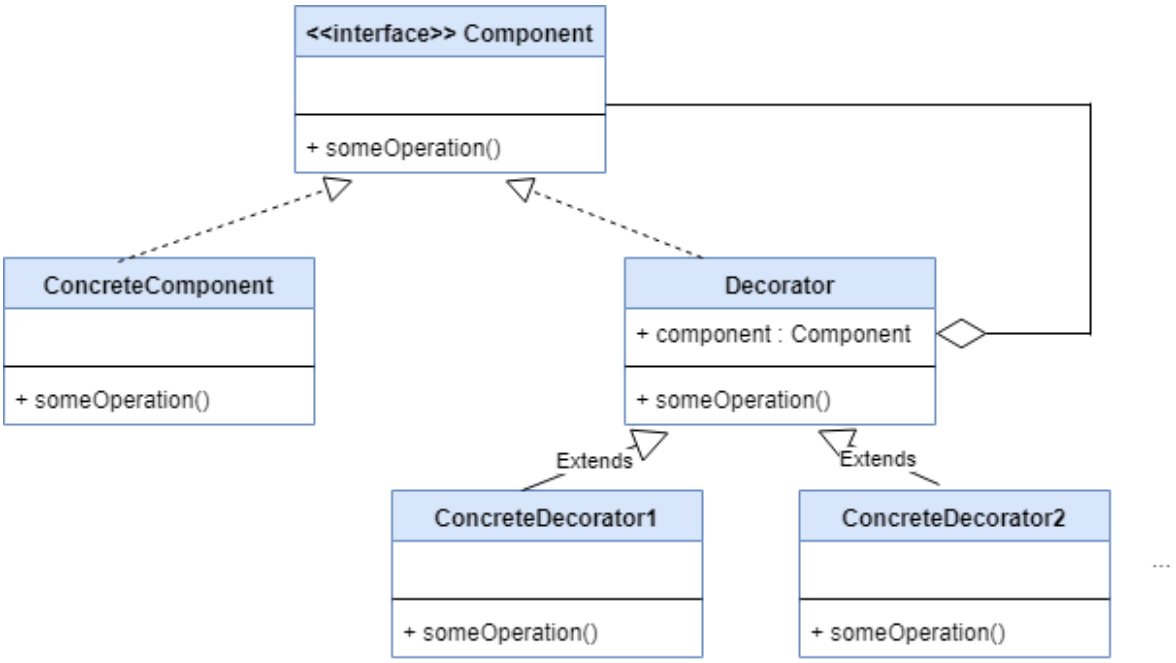


Ogni volta che si verifica un evento importante per il publisher, chiama il metodo di notifica specifico sugli oggetti dei subscribers.

È fondamentale che tutti gli abbonati implementino la **stessa interfaccia** e che il publisher **comunichi** con loro **solo** tramite **quell'interfaccia**. Questa interfaccia deve dichiarare il metodo di notifica insieme a una serie di parametri che l'editore può utilizzare per passare alcuni dati contestuali insieme alla notifica.

Decorator Design Pattern

È un pattern strutturale. Rappresenta un'alternativa dinamica alla statica eredità. Il decorator rende possibile l'aggiungere responsabilità addizionali agli oggetti durante l'esecuzione



Struttura

La struttura del decorator pattern si compone di quattro elementi principali:

- **Component**: rappresenta l'interfaccia dell'oggetto che dovrà essere decorato dinamicamente,
- **ConcreteComponent**: rappresenta l'oggetto a cui andranno aggiunte le nuove funzionalità,
- **ConcreteDecorator**: rappresentano gli oggetti che aggiungono le nuove funzionalità ai ConcreteComponent,
- **Decorator**: (può anche essere astratta) obbliga le sue sottoclassi (ConcreteComponent) ad implementare Componente e referenziare un Component con un'aggregazione.

Esempio

Supponiamo di voler realizzare un'applicazione grafica che permette di aggiungere effetti speciali (3d e trasparente) ad un'immagine. Per farlo, si usa la classe Immagine come ConcreteComponent e le classi Effetto3DDecorator e TrasparenteDecorator come ConcreteDecorator.

```
Immagine monnaLisa = new Immagine();
monnaLisa.visualizza();

Effetto3DDecorator monnaLisa3D = new Effetto3DDecorator(monnaLisa);
monnaLisa 3D.visualizza();

TrasparenteDecorator monnaLisa3DTrasparenteDecorator=new TrasparenteDecorator(monnaLisa3D);
monnaLisa3DTrasparenteDecorator.visualizza();
```

Alla riga 1 e 2 viene istanziata l'immagine monnalisa e viene poi visualizzata. Alla riga 3 viene istanziato un decoratore che viene istanziato **aggregando** l'oggetto monnaLisa appena istanziato (Nota che ogni decoratore ha un unico costruttore che prende come input obbligatoriamente un Component. Questo comportamento prende il nome di concetto di aggregazione).

MVVM (Model-View-ViewModel)

Il pattern MVVM (Model-View-ViewModel) è un pattern architetturale utilizzato nello sviluppo di software per separare la logica di business (back-end) dall'interfaccia utente (front-end), facilitando così la manutenzione e il testing del codice. Viene spesso utilizzato in applicazioni con interfacce utente complesse, come quelle sviluppate con WPF, Xamarin, o Angular, ma è applicabile in molti contesti di sviluppo software.

Componenti Principali

Model: Rappresenta i dati e la logica di business dell'applicazione. Non sa nulla dell'interfaccia utente.

View: È la rappresentazione visiva dei dati, l'interfaccia utente. Mostra i dati contenuti nel ViewModel con cui interagisce.

ViewModel: Funziona da ponte tra il Model e la View. Contiene logiche di presentazione e comandi che la View può usare per presentare dati e interagire con il Model. Il ViewModel notifica la View di eventuali cambiamenti dei dati usando pattern come il data binding.

Esempio semplice

Model

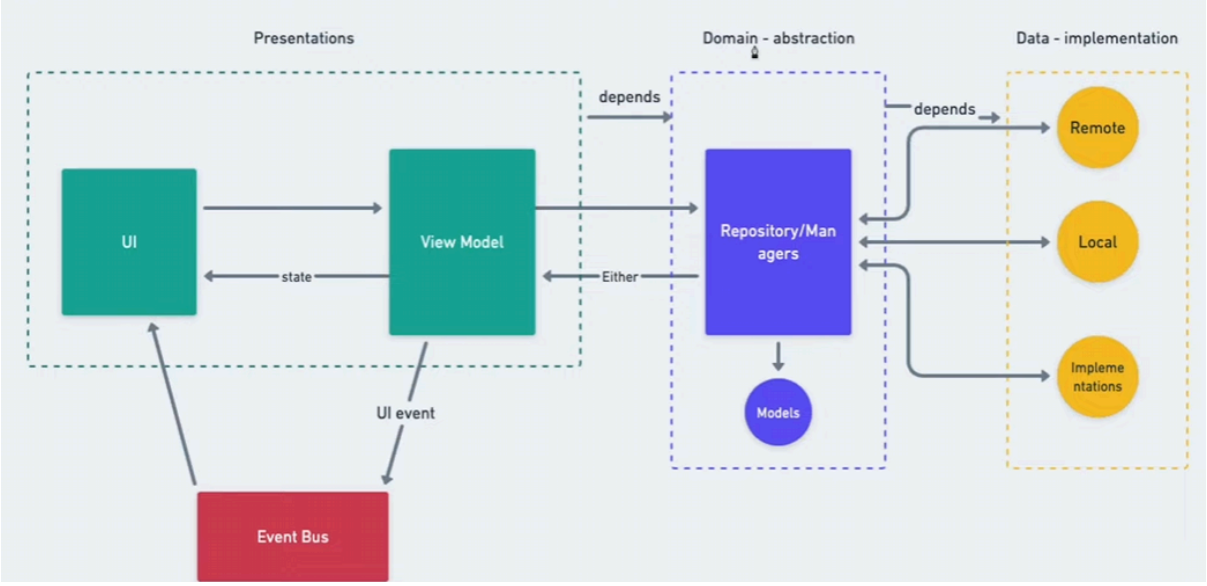
```
public class Note {
    private String title;
    private String content;
    // Costruttori, getter e setter
}
```

ViewModel

```
public class NoteViewModel extends ViewModel {
    private MutableLiveData<List<Note>> notes = new MutableLiveData<>();

    public LiveData<List<Note>> getNotes() { return notes; }

    public void loadNotes() {         // Simula il caricamento delle note
        List<Note> loadedNotes = ...; // Carica note
        notes.setValue(loadedNotes);
    }
}
```

View

```
public class NotesActivity extends AppCompatActivity {
    private NoteViewModel viewModel;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_notes);

        viewModel = new ViewModelProvider(this).get(NoteViewModel.class);
        viewModel.getNotes().observe(this, new Observer<List<Note>>() {
            @Override
            public void onChanged(List<Note> notes) {
                // Aggiorna l'UI con le note caricate
            }
        });
        viewModel.loadNotes();
    }
}
```



Presentation

Contiene UI e ViewModel e tutto ciò che c'entra con l'UI ed é visibile all'utente (activity, dialogs, bottom sheets, ...). il ViewModel viene usato per gestire i click, salvare gli stati (a.e. in una schermata per il login con un testo per l'inserimento dell'email e uno per la password, email e password vengono salvati nel ViewModel).

Ui usa ViewModel e ViewModel ritorna lo stato alla UI per mostrare all'utente.

Domain

Contiene constraints e regole dell'app. Contiene solo astrazioni (repository/manager e models ossia interfacce e classi data)

Data

Ottiene dati da un API o da un Database e/o implementa repository/manager definiti nel Domain.

Un esempio

Per fare un login da un menu di login, l'utente inserisce le credenziali (email e password) attraverso l'UI, viene chiamata una funzione per il login all'interno del ViewModel, il quale utilizza il login repository per chiamare la funzione di login dal repository, il quale fa una chiamata al database. A questo punto il DB ritorna un risultato e la repository ritorna un risultato detto Either al ViewModel, il quale modifica la UI.

Lambda calcolo...

...Come filtro

```
private final List<Automobile> automobili = new ArrayList<>();

/**
* Metodo per filtrare un elenco di veicoli
* @param marca ≠ null, rappresenta il parametro filtro
* @return lista di veicoli disponibili della marca inserito
*/
public List<Automobile> elencoVeicoliFiltrati(Marca marca) {

   return automobili
      .stream()
      .filter((s) → s.isDisponibilita())     // filtra tutte le automobili disponibili
      .filter((s) → s.getMarca() == marca)   // filtra tutte le automobili della marca
      .toList();
}
```