# Introduzione

---

**Informazioni Corso:**

---

**Modalità ibrida:** Mercoledì e giovedì → Sesto san Giovanni ufficio.

**A valle del corso:** tirocinio + inserimento per un cliente o per l’azienda.

---

**Introduzione Corso (ripasso sulla programmazione OOP)**

---

Motivazione del paradigma ad oggetti:

- Creare un’interfaccia standard
- Separare la logica in classi ed oggetti

---

**3 Pilastri della programmazione OOP:**

---

1. **Incapsulamento:**

   > Mettere attributi e metodi all’interno della definizione di classi, con dei **modificatori di accesso**, in modo tale da potervi accedere in maniera **controllata e tramite permessi.**

Questo per:

- Aggiungere un livello di sicurezza
- Avere una coerenza logica rispetto alle informazioni che abbiamo.

2. **Ereditarietà:**

   > Risolve il problema del riutilizzo di codice, permette ad una classe **figlia di ereditare informazioni da una classe madre.**

   Questo per:

   - Implementare un primo strato di polimorfismo (override nella classe figlia, delle funzioni ereditate dalla classe madre).

3. **Polimorfismo:**

   > Capacità di un oggetto di assumere **un comportamento diverso d’accordo al contesto in cui viene utilizzato (esempio pezzi di scacchi).**

   Si implementa tramite overload e override.

---

## IDE e ambienti di lavoro

---

**Abbiamo:**

1. Compilatore/ Interprete (ripasso sulle differenze).
2. Editor di testo.
3. Debugger.

Noi utilizzeremo il vecchio e buon **Eclipse.**

> 🤔 **Vantaggi/Svantaggi Linguaggi compilati/Interpretati**
>
> I linguaggi Compilati creano un file oggetto “intermedio” nel processo di traduzione da Codice Sorgente ad eseguibile.
>
> La Compilazione è più veloce in termini di tempi di esecuzione, ma creando un file intermedio che occupa più memoria.
>
> Purtroppo però non è portabile, in quanto il risultato di un processo di compilazione dipende dalle configurazioni architetturali del calcolatore in cui si sta avviando il processo. Se compilo un programma sul mio PC non è garantito che funzioni anche su un altro.
>
> L’ interprete invece è più lento, ma interpreta riga per riga il codice → quindi è portabile.

**In Java:**

Abbiamo la **JVM (Java Virtual Machine)**, ossia un “calcolatore virtuale” che permette di leggere **sorgenti java e compilarli in bytecode** (un linguaggio oggetto creato apposta per Java).

In questo modo possiamo **compilare senza perdere la portabilità del codice.**

Da byteCode in poi invece il codice **è interpretato.**

---

## Esercizio

---

Scrivere un programma che implementi tre classi: CartaDiCredito, Postepay e N26.

La prima è padre delle restanti.

Una CartaDiCredito ha al suo interno: nome e cognome del proprietario, numero carta, CVV,scadenza e saldo.

I metodi per questa classe sono quelli base (costruttore + set/get) e uno speciale metodo di stampa (stampaDati()) da ridefinire per i figli.

Le classe Postpay aggiunge l'IBAN come attributo; la classe N26 aggiunge IBAN e codiceCliente.

---

## Gestione delle eccezioni

---

Il codice che potrebbe sollevare un’eccezione è il **try_catch.**

L’ eccezione è un oggetto vero e proprio → che viene creato nel blocco try, al momento in cui si verifica un errore.

Una volta generato l’errore → si esce dal blocco try → si lancia un’eccezione (oggetto) → che viene catturata dal primo metodo catch che può gestirlo.

```
import java.util.Scanner;

public class Main {
  public static void main( String[]  args) {

    Scanner scan = new Scanner(System.in);

    System.out.println("Dividendo:");
    int dividendo = scan.nextInt();

    System.out.println("Divisore:");
    int divisore = scan.nextInt();

    // Codice "sospetto"
    try{

      // Condizione che solleva l'eccezione
      if(divisore == 0) {
        throw new Exception("Non puoi dividere per 0!");
      }

    // Con il catch dobbiamo "matchare" il tipo di eccezione lanciata
    // Poi gestisco il catch
    }catch(Exception e) {

      // prendo il messaggio contenuto nella Exception
      System.out.println(e.getMessage());

      boolean Finito = false;

      while(!Finito) {

        System.out.println("Correggi il divisore:");
        divisore = scan.nextInt();

        if(divisore != 0) {
          Finito = true;
        }

      }

    }finally {
      // Viene eseguito SEMPRE e COMUNQUE, che si verifichi oppure no la Catch
      System.out.println(dividendo/divisore);
    }

  }
}
```

🤔 Curiosità: Scanner → manipola un file con uno stream → che poi viene letto dalla console ed eseguito.

---

## Errore comune nel try_catch

Il blocco try, catch e finally hanno ognuno **uno scope indipendente.**

Se dichiaro una variabile nel try → non posso usarla nel catch o nel finally **perchè sono in un diverso scope.**

Se voglio utilizzare delle variabili in try,catch e finally **devo dichiararle nello primo scope di livello più alto che sia comune a tutti e 3:**

```
int i = 0;

try{
  // Qui posso usare i
  int j = 10;
}catch{
  // Qui posso usare i
  // Qui NON posso usare j
}finally{
  // Qui posso usare i
  // Qui NON posso usare j
}
```

---

## Metodi static

I metodi statici (membri di una classe), permettono di avere una variabile di **visibilità globale**, senza dover creare un’istanza della classe in cui vive.

**Svantaggio:** Non ha visibilità degli attributi della classe di cui fa parte.

I metodi static possono utilizzare solo **altri metodi static.**

---

## Propagazione degli errori di funzioni

---

Aggiungendo un **throws** nella firma della funzione → sto dicendo che la funzione **potrebbe lanciare un’eccezione.**

In altre parole → sto delegando la gestione dell’eccezione esternamente al chiamante.

```
public static void verificaDivisore(int divisore) throws Exception {
    if(divisore == 0) {
      throw new Exception();
    }
  }
```

```
import java.util.Scanner;

public class Main {
  public static void main( String[]  args) {

    Scanner scan = new Scanner(System.in);

    System.out.println("Dividendo:");
    int dividendo = scan.nextInt();

    System.out.println("Divisore:");
```

```
    int divisore = scan.nextInt();

    // Codice "sospetto"
    try{
      // Funzione di una funzione che può lanciare un'eccezione
      verificaDivisore(divisore);

    }catch(Exception e) {

      // prendo il messaggio contenuto nella Exception
      System.out.println(e.getMessage());

      boolean Finito = false;

      while(!Finito) {

        System.out.println("Correggi il divisore:");
        divisore = scan.nextInt();

        if(divisore != 0) {
          Finito = true;
        }

      }

    }finally {
      // Viene eseguito SEMPRE e COMUNQUE, che si verifichi oppure no la Catch
      System.out.println(dividendo/divisore);

    }
  }


  public static void verificaDivisore(int divisore) throws Exception {
    if(divisore == 0) {
      throw new Exception("Non puoi dividere per 0!");
    }
  }
```

🤔 Posso rilanciare l’eccezione al “livello di scope” successivo nella gerarchia e delegarlo ad un altro chiamante più altro (in questo caso siamo nel main quindi non sarabbe corretto farlo).

---

## Come creare delle Eccezioni Custom

I catch **vengono catturati in modo sequenziale.**

Per questo è utile, se voglio **implementare più catch di errore per lo stesso try**, creare **delle eccezioni custom.**

Creiamo una nuova classe DivisionePerZeroException

```
// Bisogna creare una classe
public class DivisionePerZeroException extends Exception{

  // Costruttore
  public DivisionePerZeroException() {
    super();
  }

  public void msqErrore() {
    System.out.println("Impossibile eseguire una divisione per zero!");
    System.exit(0);
  }

}

// Che diventa il tipo di eccezione da catturare
```

## Esercizio

Modificare il codice di prima in modo tale che se un utente cerca di prelevare più del saldo disponibile, viene lanciata un’eccezione custom.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/a0a79d13-f2ac-49cb-b4c4-fab11d4455de/Soluzione_Eccezioni.zip

## Utilizzo pratico delle variabili / funzioni static

Le funzioni statiche sono utili per scrivere funzioni **che servono ovunque in giro per il codice.**

Le separo in una classe di funzioni che fanno da “coltellino svizzero” per tutte le altre.

Esempio:

```
// Ecco la mia classe "coltellino svizzero"
public class FunzioniUtili {

  // Qui ho un array di parametri (non so quanti) di interi salvati in is
  public static int somma(int...is ) {

    int risultato = 0;

    for( int i = 0; i < is.length; i++) {
      risultato += is[i];
    }

    return risultato;
  }
}
```

```
public class Main {

  public static void main( String[]  args) {
```

```
    // Uso la funzione somma della classe "coltellino svizzero"
    // FunzioniUtili
    System.out.println(FunzioniUtili.somma(1,2,3));

  }

}
```

🤔 Il parametro di funzione (int…is) è un **parametro che prende un numero indefinito di interi e li incapsula in un array.**

Cioè → se passo alla funzione 100 interi → verranno incapsulati in un array di 100 interi a cui posso accedere localmente alla funzione.

🤔 **Curiosità**

In questo esercizio abbiamo anche visto:

```
int x = 1;
// Utilizzo dei placeholder + printf
System.out.printf("Numero positivo e negativo: %d %d", x, -x);
```

## Differenza tra funzione static e variabile static

C’è una differenza tra funzione static e variabile static.

**Una funzione static** → membro di una classe, ma non deve essere istanziata per essere usata esternamente. Non ha visibilità

della parte privata della classe in cui vive.

**Una variabile static** → è una variabile **condivisa tra tutte *le istanze della classe* in cui vive.**

---

**Esercizio**

---

**Premessa:** Abbiamo implementato insieme l’algoritmo di ordinamento Bubblesort.

“Data una stringa di caratteri, ordinarli alfa-numericamente.”

“Per ogni lettera, contare il numero di occorrenze nella stringa ottenuta dal punto precedente.”

> https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b70e30d4-6e09-4cb2-be99-dc70fc4d4b49/Main.java

---