

Primo Sprint

Introduzione a Java

Java è un linguaggio di programmazione **orientato a oggetti** (object oriented language) e a **tipizzazione statica**.

Java è un linguaggio **pseudo-compilato**, ciò ne garantisce la **portabilità**, in quanto l'output restituito dal compilatore Java è in formato bytecode, il quale può essere poi convertito in assembly dalla **Java Virtual Machine (JVM)**, indipendentemente dalla macchina su cui il codice è stato scritto inizialmente.

JRE e JDK:

JRE sta per **Java Runtime Environment**. Come dice il nome, esso è un ambiente di runtime, include quindi tutto il necessario per l'esecuzione di un programma Java già compilato, come la JVM, la Java Class Library e altre infrastrutture. Non può essere usato per la creazione di nuovi programmi!

JDK sta per **Java Development Kit**. E' quindi un kit di sviluppo software completo di tutte le funzioni: include di suo tutte le funzionalità della JRE, aggiungendo ad essa il compilatore (javac da linea di comando), il debugger e altri tools, come javadoc e jdb.

NOTA BENE: la JDK `e necessaria non solamente quando si sviluppa; ad esempio è necessaria anche se si deploya una web-app che utilizza le JSP, in quanto l' application server deve convertire le JSP in Java Servlets e ha bisogno del compilatore per farlo!



Principi dell'OOP

I quattro principi fondamentali della programmazione orientata a oggetti sono:

- 1. Incapsulamento**
- 2. Ereditarietà**
- 3. Polimorfismo**
- 4. Astrazione**

Seguire questi principi garantisce la stesura di un codice flessibile, facilmente testabile e coerente con gli standard. Vediamo ora tali principi nel dettaglio e come rispettarli quando scriviamo in Java.

Incapsulamento:

L'idea dietro l'incapsulamento è quella di **nascondere i dettagli** interni di come un oggetto funziona e di esporre solo ciò che è **necessario** per interagire con l'oggetto.

In termini pratici, l'incapsulamento in Java viene solitamente attuato utilizzando variabili di istanza **private** (o attributi) e **metodi public** per accedere e modificare tali variabili, conosciuti come **getter** e **setter**. Questo approccio è spesso chiamato "**data hiding**".

Ecco un esempio di come l'incapsulamento può essere implementato in Java:

```
public class Circle {  
    private double radius; // attributo privato  
  
    // Costruttore  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
  
    // Getter  
    public double getRadius() {  
        return radius;  
    }  
  
    // Setter  
    public void setRadius(double radius) {  
        if (radius >= 0) {
```

```

        this.radius = radius;
    } else {
        System.out.println("Errore: il raggio non può essere
negativo");
    }
}

public double getArea() {
    return Math.PI * radius * radius;
}
}

```

In questo esempio, l'attributo **radius è nascosto** all'esterno della classe Circle tramite la keyword **private**. Tuttavia, si può ancora accedere e modificare il raggio attraverso i metodi `getRadius()` e `setRadius()`. Questo significa che possiamo controllare come il raggio viene modificato (ad esempio, possiamo assicurarci che il raggio non sia mai negativo).

Ecco come si potrebbe utilizzare questa classe Circle:

```

public static void main(String[] args) {
    Circle circle = new Circle(5.0);

    // Accesso al raggio attraverso il getter
    System.out.println(circle.getRadius()); // Output: 5.0

    // Modifica del raggio attraverso il setter
    circle.setRadius(10.0);
    System.out.println(circle.getRadius()); // Output: 10.0

    // Tentativo di impostare un raggio negativo
    circle.setRadius(-5.0); // Output: Errore: il raggio non può
essere negativo

    // Calcolo dell'area
    System.out.println(circle.getArea()); // Output:
314.1592653589793
}

```

In questo esempio, non possiamo accedere **direttamente** alla variabile `radius` di `circle` perché è privata. Tuttavia, possiamo ancora ottenere e impostare il suo valore tramite i metodi `getRadius()` e `setRadius()`. Questo è un esempio di encapsulamento.

Ereditarietà:

L'ereditarietà ci consente di strutturare una gerarchia tra le classi. Una classe figlia eredita tutte le funzionalità della classe madre, ciò consente di strutturare gerarchicamente il nostro programma, evitando ripetizioni di codice, consentendo maggior chiarezza e specializzazione. In java, per creare una classe figlia di un'altra si utilizza la keyword **extends** nella dichiarazione. Diamo innanzitutto un'occhiata al codice:

```
import java.util.Date;

public class Persona {
    private String nome;
    private Date dataNascita;
    protected final String CONST = "di Persona";
    public Persona(String nome, String dataNascita) {
        this.nome = nome;
        this.dataNascita = dataNascita;
    }
    public void greetings() {
        System.out.println("Ciao!");
    }
}

public class Impiegato extends Persona {
    private Long id;
    private final String CONST = "di Impiegato";
    public Impiegato(String nome, Date dataNascita, Long id) {
        super(nome, dataNascita); this.id = id;
    }
    public void demoSuper() {
        System.out.println(this.CONST + " / " + super.CONST);
    }
}

public class App() {
    public static void main(String[] args) {
        Impiegato myImp = new Impiegato("Luca", new
            Date(1999,09,03), 1254563);
        myImp.greetings(); myImp.demoSuper();
    }
}
```

Andiamo con ordine:

- **Persona** è la classe **madre**, essa ha come valori un nome e una data di nascita. Il costruttore assegna questi valori.
Nota: abbiamo volutamente chiamato i parametri del costruttore con lo stesso nome delle variabili locali; questo non è un problema in quanto grazie alla keyword this siamo in grado di specificare quando ci stiamo riferendo alle variabili e quando ai parametri.
- **Impiegato** è la classe **figlia**: eredita tutti gli attributi e i metodi di **Persona**, senza alcun bisogno di dichiararli nuovamente, e introduce l'attributo id, specializzandosi in questo rispetto alla classe madre. Osserviamo ora bene il costruttore: la keyword **super** invoca il costruttore della classe madre, inizializzando quindi le variabili ereditate.
Nota: super ha anche un altro utilizzo, simile a quello di this: serve a riferirsi a una variabile della classe madre qualora essa venga nascosta nella classe figlia, come nel caso di CONST nell'esempio di qui sopra.

E` importante ricordare che una classe può estendere ` una e una sola classe madre. Inoltre è possibile limitare la disponibilità all'ereditarietà tramite la keyword **final**: una classe o un metodo dichiarato come final non può essere esteso/a (mentre una variabile, al solito, non può essere modificata).

Polimorfismo:

Il **polimorfismo** è uno dei quattro concetti principali della programmazione orientata agli oggetti. Il termine polimorfismo deriva dal greco "poli" che significa "molti" e "morphos" che significa "forme". Quindi il polimorfismo significa "**molte forme**".

In Java, il polimorfismo permette ai programmati di trattare una classe come la sua **superclasse o interfaccia**, il che può essere molto utile quando si lavora con gruppi di classi correlate.

Esistono due tipi principali di polimorfismo in Java: **il polimorfismo statico (o overloading)** e **il polimorfismo dinamico (o overriding)**.

1. Polimorfismo statico (Overloading)

L'overloading si verifica quando due o più metodi nella stessa classe hanno lo stesso nome ma parametri diversi.

Ad esempio:

```
public class Mathematics {
    public int add(int a, int b) {
        return a + b;
    }
}
```

```

    }

    public double add(double a, double b) {
        return a + b;
    }
}

```

In questo esempio, il metodo add è un **Overload**. C'è una versione che accetta due interi e una che accetta due double. Java determina quale metodo chiamare basandosi sui tipi di dati dei parametri al momento della compilazione.

2. Polimorfismo dinamico (Overriding)

L'overriding si verifica quando una sottoclasse fornisce un'implementazione specifica di un metodo che è già fornito dalla sua superclasse.

Ad esempio:

```

public class Animal {
    public void sound() {
        System.out.println("L'animale fa un suono");
    }
}

public class Dog extends Animal {

    @Override
    public void sound() {
        System.out.println("Il cane abbaia");
    }
}

public class Cat extends Animal {

    @Override
    public void sound() {
        System.out.println("Il gatto miagola");
    }
}

```

In questo esempio, il metodo sound è **sovrascritto** nelle classi Dog e Cat. Java determina quale versione del metodo chiamare basandosi sull'oggetto che viene usato per chiamarlo **al momento dell'esecuzione**.

Vediamo un esempio di come si può usare il polimorfismo in pratica. Creiamo un metodo in un'altra classe che accetta un'interfaccia Animal come parametro:

```
public class AnimalCare {  
  
    public void checkSound(Animal animal) {  
        animal.sound();  
    }  
}
```

Il metodo `checkSound()` può accettare qualsiasi oggetto che implementa l'interfaccia **Animal**. In questo modo, possiamo utilizzare il polimorfismo per lavorare con diversi tipi di animali **senza dover scrivere un metodo specifico per ogni tipo di animale**.

Ecco un esempio di come si può usare questo metodo:

```
public static void main(String[] args) {  
    AnimalCare animalCare = new AnimalCare();  
  
    Dog myDog = new Dog();  
    Cat myCat = new Cat();  
  
    animalCare.sound(myDog); // Output: Il cane abbaia  
    animalCare.sound(myCat); // Output: Il gatto miagola  
}
```

In questo esempio, sia `myDog` che `myCat` sono passati al metodo `checkSound()`. Nonostante `myDog` e `myCat` siano di **tipi diversi**, entrambi implementano l'interfaccia **Animal**, quindi entrambi possono essere passati al metodo **checkSound()**. Quando il metodo `sound()` viene chiamato su **animal** all'interno del metodo `checkSound()`, la versione appropriata del metodo viene chiamata a seconda del **tipo reale dell'oggetto**. Questo è un esempio di polimorfismo **dinamico**.

Un altro esempio di Polimorfismo in Java è fornito dai **Generics**. Andiamo a vedere di cosa si tratta.

I Generics sono un concetto in Java che permette la **tipizzazione dei parametri**. Questo significa che si possono definire classi, interfacce o metodi che possono operare su **diversi tipi di dati** mantenendo al contempo la sicurezza del tipo.

Uno dei principali vantaggi dell'utilizzo dei Generics è che eliminano la necessità di casting. Senza Generics, se si desidera avere un codice che può operare su diversi tipi di dati, si dovrebbe utilizzare il tipo di dato più generale disponibile, cioè la classe Object. Tuttavia, ciò richiederebbe il casting ogni volta che si desidera operare su un tipo di dato specifico.

Ecco un esempio di come i Generics possono essere utilizzati in Java:

```
public class Box<T> {
```

```

private T t;

public void add(T t) {
    this.t = t;
}

public T get() {
    return t;
}
}

```

In questo esempio, **T** è un parametro di tipo che sarà sostituito con un tipo reale quando un oggetto di **Box** sarà creato.

Ecco come si può utilizzare la classe Box:

```

Box<Integer> integerBox = new Box<Integer>();
Box<String> stringBox = new Box<String>();

integerBox.add(new Integer(10));
stringBox.add(new String("Hello World"));

System.out.printf("Integer Value :%d\n", integerBox.get());
System.out.printf("String Value :%s\n", stringBox.get());

```

In questo esempio, creiamo due box: uno per contenere un **intero** e uno per contenere una **stringa**. Quando chiamiamo il metodo **add()**, non abbiamo bisogno di fare il casting perché sappiamo che il tipo di dato è corretto. Analogamente, quando chiamiamo il metodo **get()**, otteniamo il tipo di dato corretto senza la necessità di fare il casting.

I Generics in Java sono utilizzati in modo estensivo nelle librerie standard, in particolare nelle collezioni come List, Set, Map, ecc. Ad esempio, possiamo creare una List di String:

```

List<String> list = new ArrayList<String>();
list.add("Hello");
String s = list.get(0); // Nessun casting necessario

```

In questo caso, non solo evitiamo il casting, ma otteniamo anche un errore di compilazione se proviamo ad aggiungere un tipo di dato non corretto alla lista, rendendo il nostro codice **più sicuro**.

Astrazione:

L'astrazione si riferisce al processo di **nascondere** i dettagli complessi di come un oggetto funziona, presentando all'utente o al programmatore solo le informazioni e le funzionalità necessarie.

L'astrazione è un concetto potente che può essere applicato a vari livelli nell'ambito dello sviluppo software. Ad esempio, un metodo o una funzione forniscono un certo grado di astrazione: eseguono un insieme di operazioni, ma il codice che chiama il metodo non ha bisogno di conoscere **i dettagli specifici** di come quelle operazioni vengono eseguite.

In termini più specifici dell'OOP, le classi forniscono un altro livello di astrazione. Una classe definisce una serie di metodi e variabili, ma il codice che utilizza un'istanza della classe (un oggetto) non ha bisogno di conoscere come la classe è implementata. Può interagire con l'oggetto solo attraverso i suoi metodi pubblici, e non ha accesso diretto alle sue variabili interne (a meno che non siano dichiarate pubbliche).

Infine, in Java e in alcuni altri linguaggi OOP, è possibile definire **interfacce** o **classi astratte**, che rappresentano un ulteriore livello di astrazione. Un'interfaccia o una classe astratta definisce un **contratto** - un insieme di metodi che le classi che implementano l'interfaccia o estendono la classe astratta devono fornire - ma non fornisce **un'implementazione** di quei metodi.

Ecco un esempio di come funziona l'astrazione in Java utilizzando le classi astratte e le interfacce:

Classe astratta:

Una classe astratta è una classe che **non può essere istanziata**. Può contenere sia metodi **astratti** (metodi senza corpo) che **non astratti** (metodi con corpo).

```
public abstract class Animal {  
    public abstract void makeSound();  
  
    public void eat() {  
        System.out.println("L'animale mangia");  
    }  
}
```

Nell'esempio sopra, Animal è una classe astratta con un metodo astratto makeSound(). Questo metodo non ha un'implementazione nella classe Animal. Ogni classe che estende Animal **dovrà fornire un'implementazione** per makeSound().

```
public class Dog extends Animal {
```

```

        public void makeSound() {
            System.out.println("Il cane abbaia");
        }
    }

public class Cat extends Animal {
    public void makeSound() {
        System.out.println("Il gatto miagola");
    }
}

```

In questo caso, sia Dog che Cat estendono Animal e forniscono la propria implementazione del metodo makeSound().

Interfaccia

Un'interfaccia è simile a una classe astratta in quanto non si può istanziare. A differenza di una classe astratta, tuttavia, un'interfaccia **non può contenere metodi non astratti** (cioè metodi con corpo) e le variabili sono implicitamente **final** e **static**.

```

public interface Vehicle {
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}

```

Nell'esempio sopra, Vehicle è un'interfaccia con tre metodi astratti. Ogni classe che implementa Vehicle deve fornire un'implementazione per questi metodi.

```

public class Bike implements Vehicle {
    int speed;
    int gear;

    public void changeGear(int newGear) {
        gear = newGear;
    }

    public void speedUp(int increment) {
        speed = speed + increment;
    }

    public void applyBrakes(int decrement) {
        speed = speed - decrement;
    }
}

```

In questo caso, la classe Bike implementa l'interfaccia Vehicle e fornisce la propria implementazione per tutti i metodi dell'interfaccia.

In entrambi gli esempi, l'astrazione consente di definire un contratto (sia attraverso classi astratte che interfacce) che **le classi concrete devono rispettare**, nascondendo i dettagli di implementazione all'utilizzatore della classe o dell'interfaccia.

Usando classi astratte e interfacce, si possono raggiungere molti benefici, tra cui il **riutilizzo** del codice, **l'estendibilità** e la **modularità**.

1. Riutilizzo del codice

Le classi astratte permettono di definire metodi concreti (implementati) che possono essere ereditati dalle sottoclassi, riducendo la duplicazione del codice.

Ad esempio, considera la seguente classe astratta Animal e le sue sottoclassi Dog e Cat:

```
public abstract class Animal {  
    public void eat() {  
        System.out.println("L'animale mangia");  
    }  
  
    public abstract void makeSound();  
}  
  
public class Dog extends Animal {  
    public void makeSound() {  
        System.out.println("Il cane abbaia");  
    }  
}  
  
public class Cat extends Animal {  
    public void makeSound() {  
        System.out.println("Il gatto miagola");  
    }  
}
```

In questo caso, il metodo eat() è condiviso sia dalla classe Dog che dalla classe Cat, il che riduce la duplicazione del codice.

2. Estendibilità

Le interfacce consentono di estendere le funzionalità di una classe senza modificare il codice esistente (**principio del Open/Closed**). Ciò rende il codice più flessibile e adattabile ai cambiamenti.

Considera la seguente interfaccia Flyable e la classe Bird che la implementa:

```
public interface Flyable {  
    void fly();  
}  
  
public class Bird implements Flyable {  
    public void fly() {  
        System.out.println("L'uccello vola");  
    }  
}
```

Se, in futuro, dovessi aggiungere una nuova classe Airplane che può volare, puoi facilmente farla implementare l'interfaccia Flyable **senza dover modificare il codice esistente**.

3. Modularità

Le interfacce e le classi astratte permettono di definire **contratti chiari tra differenti parti del tuo sistema**. Questo rende il tuo codice più modulare e più facile da **testare e mantenere**.

Considera l'esempio del Vehicle di prima. Se hai un metodo che prende un oggetto Vehicle come parametro, non ti importa quale sia la specifica implementazione (potrebbe essere Car, Bike, Boat, ecc.). Questo ti permette di cambiare le implementazioni specifiche senza dover cambiare il codice che usa l'interfaccia o la classe astratta.

Eccezioni

Un'eccezione è un evento inaspettato che interferisce con la normale esecuzione di un programma.

Java offre al programmatore la possibilità di catturare (**catch**) tali eccezioni: quando una di queste

avviene durante l'esecuzione di un metodo, viene creata un'istanza della classe

Exception. Tale

oggetto contiene informazioni riguardanti l'eccezione, come il nome, la descrizione di cosa è andato storto e lo stato del programma al momento del lancio di tale eccezione.

Ci sono due tipi principali di eccezioni in Java:

1. **Eccezioni controllate (Checked Exceptions):** Sono eccezioni che devono essere gestite esplicitamente dal codice, altrimenti il compilatore segnalera un errore e che avvengono dunque a **tempo di compilazione**. Queste sono di solito legate a problemi recuperabili come errori di I/O, problemi di connessione alla rete, ecc.
2. **Eccezioni non controllate (Unchecked Exceptions):** Sono eccezioni che il compilatore non obbliga a gestire e che avvengono a **runtime**. Sono spesso il risultato di problemi nel codice come divisione per zero, accesso a un indice fuori da un array, ecc.

La gestione delle eccezioni in Java è basata sul concetto di "**stack di chiamate**". Quando un metodo incontra una situazione anomala (cioè un'eccezione), l'esecuzione del metodo si interrompe e l'eccezione viene "lanciata" o sollevata. Questa eccezione si propaga **retroattivamente** attraverso lo stack delle chiamate fino a quando non viene catturata e gestita da un blocco di codice **catch** appropriato.

Ecco un esempio di come gestire un'eccezione in Java:

```
try {  
    int[] array = new int[5];  
    System.out.println(array[10]); // Accesso fuori dai limiti  
dell'array  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("Errore: " + e.getMessage());  
} finally {  
    System.out.println("Questo blocco verrà eseguito  
indipendentemente dal fatto che si verifichi un'eccezione o meno.");  
}
```

In questo esempio, stiamo cercando di accedere a un indice dell'array che non esiste. Questo solleva un'eccezione **ArrayIndexOutOfBoundsException**, che è un tipo di RuntimeException (una eccezione **non controllata**).

Il blocco catch cattura l'eccezione e la gestisce stampando un messaggio di errore.

Il blocco finally viene eseguito indipendentemente dal fatto che si sia verificata un'eccezione o meno. È un buon posto per mettere il codice di pulizia, come la chiusura di un file o il rilascio di risorse.

Per le eccezioni controllate, devi o gestire l'eccezione con un blocco try/catch, o dichiarare che il tuo metodo può sollevare l'eccezione con la clausola throws. Ad esempio:

```
import java.io.BufferedReader;
```

```

import java.io.FileReader;
import java.io.IOException;

public void readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
        new BufferedReader(new FileReader(path))) {
        System.out.println(br.readLine());
    }
}

```

In questo esempio, il metodo `readFirstLineFromFile` può sollevare un'IOException, che è un tipo di eccezione controllata. Dal momento che non gestiamo l'eccezione all'interno del metodo, dobbiamo dichiarare che il metodo può sollevare l'eccezione con la clausola `throws IOException`.

Gestione eccezioni della JVM:

Quando un'eccezione viene sollevata, la **JVM** cerca nel metodo corrente un blocco `catch` che corrisponda **al tipo dell'eccezione**. Se un tale blocco viene trovato, il controllo viene trasferito a quel blocco `catch`. Altrimenti, il metodo corrente viene terminato e l'eccezione viene riportata al metodo che ha chiamato il metodo corrente. Questo processo continua fino a quando l'eccezione non viene catturata o fino a quando non raggiunge il metodo `main()`, che è il primo metodo chiamato nel programma. Se l'eccezione **non viene catturata** da nessun blocco `catch` e raggiunge il metodo `main()`, il programma termina e la JVM stampa un messaggio di errore sulla console che indica la natura dell'eccezione e lo **stack delle chiamate**, che mostra la sequenza di chiamate di metodo che hanno portato all'eccezione.

Ecco un semplice esempio per illustrare questo processo:

```

public class Main {
    public static void main(String[] args) {
        try {
            method1();
        } catch (ArithmetricException e) {
            System.out.println("Eccezione catturata: " + e);
        }
    }

    public static void method1() {
        method2();
    }

    public static void method2() {
        int x = 1 / 0; // Questo solleva un'ArithmetricException
    }
}

```

```
}
```

In questo esempio, il metodo **method2()** solleva un `ArithmeticException` a causa della divisione per zero. Poiché `method2()` non gestisce l'eccezione, termina e l'eccezione viene riportata a **method1()**. Poiché anche `method1()` non gestisce l'eccezione, termina e l'eccezione viene riportata al metodo **main()**. Il metodo `main()` ha un blocco `catch` che può gestire un `ArithmeticException`, quindi l'eccezione **viene catturata e gestita lì**. Se il blocco `catch` non fosse stato presente nel metodo `main()`, il programma sarebbe terminato e la **JVM avrebbe stampato un messaggio di errore sulla console**.

Custom Exception:

Le eccezioni definite dall'utente, anche conosciute come eccezioni **personalizzate** o **custom**, sono eccezioni che vengono create dall'utente per soddisfare specifici requisiti di **business**. Queste eccezioni estendono direttamente la classe **Exception** o una delle sue sottoclassi.

Le eccezioni definite dall'utente possono essere sia **controllate** che **non controllate**. Generalmente, si creano eccezioni controllate quando si desidera che l'applicazione recuperi da un'eccezione, e si creano eccezioni non controllate quando si desidera che l'applicazione termini.

Ecco un esempio di un'eccezione personalizzata:

```
// Eccezione personalizzata che estende la classe Exception
public class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

In questo esempio, abbiamo creato un'eccezione personalizzata chiamata **InvalidAgeException**. Questa eccezione è controllata perché estende direttamente la classe `Exception`.

Puoi lanciare questa eccezione nel tuo codice quando si verifica una condizione specifica. Ecco come potrebbe apparire:

```
public class Main {
    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch (InvalidAgeException e) {
```

```

        System.out.println("Eccezione catturata: " +
e.getMessage());
    }
}

// Metodo che verifica l'età di una persona
public static void checkAge(int age) throws InvalidAgeException
{
    if (age < 18) {
        throw new InvalidAgeException("Età non valida, deve
essere
                maggiore di 18 anni.");
    } else {
        System.out.println("Età valida.");
    }
}
}

```

In questo esempio, il metodo checkAge solleva un'InvalidAgeException se l'età fornita è inferiore a 18. Dal momento che InvalidAgeException è un'eccezione controllata, il metodo checkAge deve dichiarare l'eccezione con la clausola **throws**.

Nel metodo main, catturiamo l'InvalidAgeException con un blocco catch e stampiamo il suo messaggio.

Collections e Map

Una Collection è fondamentalmente un gruppo di oggetti individuali rappresentati come una **singola unità**. A partire dalla JDK 1.2, Java mette a disposizione un **framework** apposito, nel quale sono contenute tutte le classi e interfacce relative alle collections. Le due principali interfacce di questo framework sono la **java.util.Collection** e la **java.util.Map**, in base alle quali vengono implementate tutte le altri classi/interfacce del framework.

Il Java Collection Framework include le seguenti interfacce principali:

Collection: Questa è l'interfaccia radice del framework delle collezioni. Non fornisce funzionalità per la manipolazione di dati.

List: Questa interfaccia estende Collection e rappresenta una lista **ordinata** di elementi. Gli elementi possono essere inseriti o acceduti per **posizione**. Inoltre, gli elementi possono essere duplicati. Esempi di implementazioni di List includono ArrayList, LinkedList, Stack e Vector.

Set: Questa interfaccia estende Collection e rappresenta un gruppo di elementi **unici** - non contiene duplicati. Esempi di implementazioni di Set includono HashSet, LinkedHashSet e TreeSet.

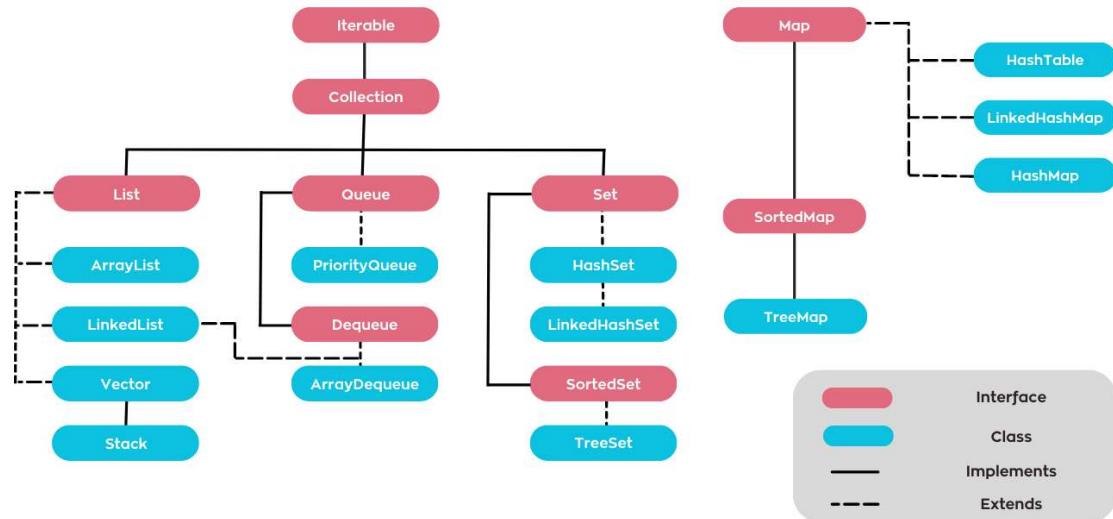
Queue: Questa interfaccia estende Collection e rappresenta una **coda** di elementi. Le code generalmente ordinano gli elementi in un ordine specifico (ad esempio, FIFO - First In, First Out). Esempi di implementazioni di Queue includono PriorityQueue, ArrayDeque, e LinkedList.

Map: Questa interfaccia **non estende Collection**, ma fa comunque parte del Java Collection Framework. Una mappa contiene valori basati su una **chiave**. Ogni chiave deve essere **unica**. Esempi di implementazioni di Map includono HashMap, LinkedHashMap, TreeMap, e Hashtable.

Esempio:

```
Map<String, Integer> map = new HashMap<>();
map.put("Apple", 1);
map.put("Banana", 2);
map.put("Cherry", 3);
System.out.println(map); // {Apple=1, Banana=2, Cherry=3}
```

Hierarchy Collection Framework



Database relazionali

I **database relazionali** sono un tipo di database che organizzano i dati in una serie di **tabelle** interconnesse tra loro. Il termine "relazionale" si riferisce al fatto che le tabelle sono collegate o "relazionate" tra loro in vari modi, permettendo ai dati di essere accessibili e organizzati in maniera efficiente.

Vediamo i termini fondamentali dei database relazionali:

- **Tabelle:** In un database relazionale, i dati vengono archiviati in tabelle. Ogni tabella rappresenta **un'entità** o un oggetto specifico all'interno del sistema, come un cliente, un prodotto o un ordine.
- **Righe e Colonne:** Ogni tabella è composta da righe e colonne. Ogni riga (chiamata anche **record** o **tupla**) rappresenta un singolo elemento all'interno dell'entità, come un particolare cliente o prodotto. Ogni colonna (o campo) rappresenta un **attributo** dell'entità, come il nome di un cliente o il prezzo di un prodotto.
- **Chiavi Primarie:** Ogni riga in una tabella ha un **identificatore univoco** chiamato chiave primaria. Questo permette di distinguere ogni record in modo univoco.
- **Chiavi Esterne:** Le chiavi esterne sono usate per stabilire **relazioni** tra le tabelle. Una chiave esterna è un campo (o un insieme di campi) in una tabella, che si riferisce alla chiave primaria di un'altra tabella.
- **Relazioni:** Le relazioni tra le tabelle possono essere di diversi tipi, tra cui **one-to-one**, **one-to-many** e **many-to-many**. Per esempio, in una relazione uno-a-molti tra i clienti e gli ordini, un cliente può avere molti ordini, ma ogni ordine è associato a un solo cliente.
- **Normalizzazione:** La normalizzazione è un processo utilizzato nella progettazione del database relazionale per **ridurre la duplicazione** dei dati. Questo processo implica la suddivisione dei dati in tabelle separate e la definizione di relazioni tra di esse.
- **SQL:** SQL (Structured Query Language) è il linguaggio standardizzato utilizzato per interrogare e manipolare i database relazionali.

I database relazionali sono molto diffusi grazie alla loro flessibilità, alla loro capacità di gestire grandi quantità di dati e alla facilità con cui possono essere interrogati.

utilizzando SQL. Tuttavia, come con qualsiasi tecnologia, ci sono casi in cui altri tipi di database (come i database NoSQL) possono essere più adatti.

SQL:

SQL, o Structured Query Language, è il linguaggio standardizzato utilizzato per **interagire** con i database relazionali. Con SQL, è possibile eseguire una varietà di compiti, tra cui creare tabelle e altri oggetti di database, inserire, aggiornare e cancellare dati, e interrogare i dati per ottenere informazioni.

Ecco una panoramica di alcuni dei comandi SQL più comuni, insieme a esempi pratici.

1. CREATE DATABASE e CREATE TABLE

Questi comandi sono utilizzati per creare un nuovo database e nuove tabelle all'interno di un database.

```
CREATE DATABASE myDatabase;
```

```
USE myDatabase;
```

```
CREATE TABLE Employees (
    ID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    BirthDate DATE
);
```

Questo codice crea un nuovo database chiamato myDatabase, quindi definisce una nuova tabella chiamata Employees con quattro colonne: ID, FirstName, LastName e BirthDate.

2. INSERT INTO

Questo comando è utilizzato per inserire nuovi record in una tabella.

```
INSERT INTO Employees (ID, FirstName, LastName, BirthDate)
VALUES (1, 'Mario', 'Rossi', '1980-01-01');
```

Questo codice inserisce un nuovo record nella tabella Employees.

3. SELECT

Questo comando è utilizzato per selezionare dati da una o più tabelle.

```
SELECT FirstName, LastName  
FROM Employees  
WHERE BirthDate > '1985-01-01';
```

Questo codice seleziona i nomi e i cognomi di tutti gli impiegati nati dopo il 1° gennaio 1985.

4. UPDATE

Questo comando è utilizzato per modificare i dati esistenti in una tabella.

```
UPDATE Employees  
SET BirthDate = '1981-01-01'  
WHERE ID = 1;
```

Questo codice aggiorna la data di nascita dell'impiegato con ID 1.

5. DELETE

Questo comando è utilizzato per eliminare i record da una tabella.

```
DELETE FROM Employees  
WHERE ID = 1;
```

Questo codice elimina l'impiegato con ID 1 dalla tabella Employees.

6. JOIN

Questa operazione è utilizzata per combinare tra loro dati appartenenti a due o più tabelle distinte in base a una proprietà comune a tutte quante. Un JOIN può essere dei seguenti tipi:

1. INNER JOIN
2. LEFT JOIN
3. RIGHT JOIN
4. FULL JOIN

Inner Join:

Questa operazione utilizza la keyword INNER JOIN (o anche semplicemente JOIN, `e equivalente)

per selezionare tutte le righe che soddisfano le **condizioni specificate** in entrambe le tabelle.

Il result set sarà quindi formato da tali righe.

Capiamo meglio con un esempio: supponiamo di avere le due tabelle Student e StudentCourse

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

Facciamo una inner join delle due tabelle, combinando la colonna ROLL NO di entrambe le tabelle.

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

Il result set sarà:

COURSE_ID	NAME	Age
1	HARSH	18
2	PRATIK	19
2	RIYANKA	20
3	DEEP	18
1	SAPTARHI	19

Left Join e Right Join:

Queste due funzionano essenzialmente allo stesso modo. Trattiamo quindi solamente il left join.

Questa operazione utilizza la keyword LEFT JOIN per restituire TUTTE le righe della tabella

a SINISTRA del join combinando la proprietà specificata se nella tabella a destra `e presente un

match, inserendo NULL altrimenti. Continuando lo stesso esempio, la seguente query:

```
SELECT Student.NAME,StudentCourse.COURSE_ID  
FROM Student  
LEFT JOIN StudentCourse  
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Produce il seguente result set:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	NULL
ROHIT	NULL
NIRAJ	NULL

Notare come in corrispondenza di NIRAJ non ci sia un valore del COURSE ID, in quanto il suo

ROLL NO (8) non è presente nella tabella a destra.

Full Join:

Questo join utilizza la keyword FULL JOIN per combinare di fatto un LEFT e un RIGHT JOIN.

Il result set sarà quindi formato da tutte le righe di tutte e due le tabelle, quando possibile (ovvero

quando la proprietà specificata matcha tra due righe) queste righe verranno combinate tra loro,

mentre per le righe in cui non si ha un match tale proprietà verrà messa NULL.

```
SELECT Student.NAME,StudentCourse.COURSE_ID
```

```
FROM Student
FULL JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Foreign Key

Una foreign key è una colonna che fa riferimento a un record di un'altra tabella tramite la primary

key di questo. Viene quindi utilizzata per mostrare le relazioni tra le tabelle e fa da ponte tra

queste. La tabella in cui una foreign key viene definita è l'**owning** side della relazione, mentre

la tabella che viene referenziata dalla foreign key è l'**opposite** side. Una foreign key può essere

definita nei seguenti modi:

1. Appena si dichiara la colonna:

```
nome_colonna tipo_colonna FOREIGN KEY REFERENCES TabellaEsterna(tab_est_PK)
```

2. Alla fine delle dichiarazioni delle colonne:

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID) REFERENCES
Persons(
    PersonID)
);
```

Proprietà Foreign Key:

1. Un record non può essere aggiornato se esistono delle FK che lo puntano (almeno per come abbiamo dichiarato le chiavi sopra, vedremo tra un attimo come rimediare);
2. Una FK deve **referenziare** la PK della tabella a cui punta;
3. La FK deve essere dello stesso **tipo** della PK a cui punta;

La proprietà 1 definisce un vincolo molto forte nella relazione. Nella maggior parte dei casi

vorremmo evitare una tale restrizione, per farlo si utilizzano rispettivamente le keyword **ON**

DELETE e **ON UPDATE** subito dopo la dichiarazione fatta sopra, ad esempio:

```
CREATE TABLE Orders (
```

```

        OrderID int NOT NULL,
        OrderNumber int NOT NULL,
        PersonID int,
        PRIMARY KEY (OrderID),
        CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID) REFERENCES
    Persons(
        PersonID)
    ON DELETE CASCADE
);

```

Queste keyword specificano cosa fare quando il record referenziato viene rispettivamente cancellato e/o modificato. I vari comportamenti e le loro rispettive keyword sono i seguenti:

- 1. CASCADE:** `è la più utilizzata, in quanto specifica che le modifiche effettuate sulla PK puntata si debbano propagare alla tabella contenente la FK.
- 2. RESTRICT:** è il comportamento che si ha di default. Non permette la modifica di un record puntato da una FK.
- 3. NO ACTION:** uguale a RESTRICT.
- 4. SET NULL:** setta la FK a NULL se si modifica ciò che referenziava in precedenza. Ciò `è utile ad esempio quando si vogliono mantenere dei record anche quando la tabella a cui facevano riferimento viene cancellata.

Tipi di relazione:

Esistono tre tipi di relazione tra le entità:

1. **One-To-One:** Relazione uno ad uno, associa un record di una tabella ad **uno e un solo** record di un'altra tabella. Esempio tra le tabelle **Persona** e **CF**(codice fiscale), ad una persona è associato un solo cf e viceversa.
2. **One-To-Many:** Relazione uno a molti, associa un record di una tabella ad **uno o più** record di un'altra tabella. Esempio tra le tabelle **Persona** e **Macchina**, una persona può possedere più macchine mentre ogni macchina appartiene solo a quella persona.
3. **Many-To-Many:** Relazione molti a molti, associa **più record di una tabella a più record** di un'altra tabella. In questo caso dovremo gestire la relazione creando una **tabella ponte** che conterrà al suo interno un riferimento alla prima tabella e un riferimento alla seconda. Esempio tra le tabelle **Customer** e **Prodotti**, un Customer può acquistare più prodotti e un Prodotto può essere acquistato da più Customer. In questo caso andremo a creare una tabella di

mezzo che possiamo chiamare **Acquisto** in cui ogni record avrà un riferimento al prodotto acquistato e uno al customer che l'ha acquistato.

Design Pattern

Un design pattern, o pattern di progettazione, è una soluzione generale e **riutilizzabile** a un **problema comune** che si presenta nel contesto della progettazione del software. Non si tratta di un frammento di codice pronto all'uso, ma piuttosto di un modello o di una descrizione di come risolvere un problema, che può essere utilizzato in diverse situazioni.

I pattern di progettazione possono accelerare il processo di sviluppo del software fornendo modelli testati e comprovati per la risoluzione di problemi ricorrenti. Essi consentono agli sviluppatori di evitare di "reinventare la ruota" ogni volta che si imbattono in un tipo di problema comune, permettendo loro di utilizzare una **soluzione esistente e ben collaudata**.

In questa guida affronteremo i pattern che utilizzerete più spesso nel corso della formazione, ovvero:

- **MVC** (Model, View, Controller);
- **DAO** (Data Access Object);
- **DTO** (Data Transfer Object);

MVC:

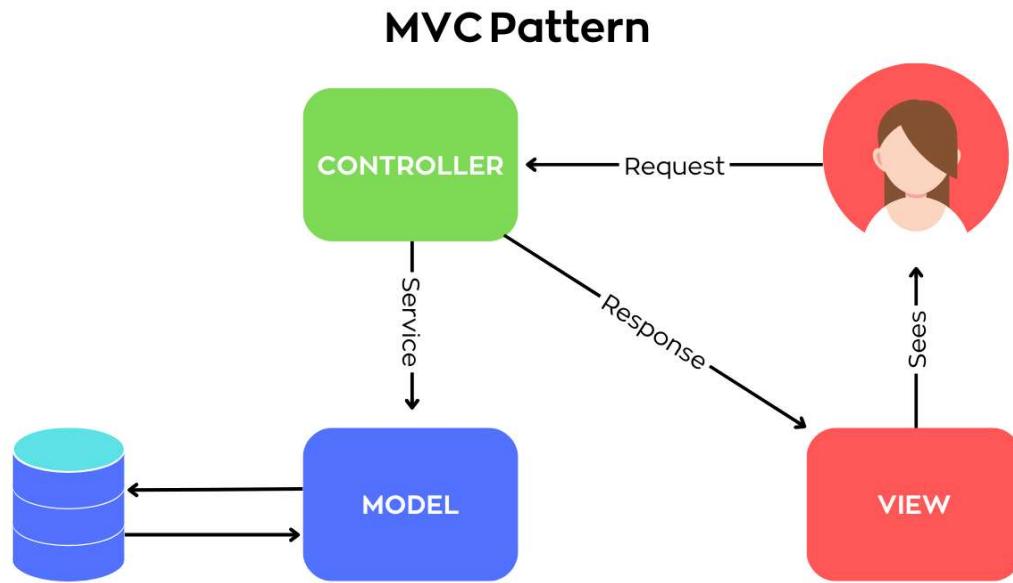
Model-view-controller, in informatica, è un pattern **architettonico** molto diffuso nello sviluppo di sistemi software, in particolare nell'ambito della programmazione orientata agli oggetti e in applicazioni web, in grado di separare la logica di **presentazione** dei dati dalla logica di **business**.

E' composto da tre componenti che interagiscono tra loro:

1. **Model:** rappresenta i **dati** puri dell'applicazione e non include logiche riguardanti la presentazione di tali dati all'utente. Si tratta di dati che fluiscono attraverso l'applicazione, ad esempio dal back-end server alla vista front-end e dalla vista front-end al database.
2. **View:** il componente con cui si **interfaccia** l'utente e con cui **interagisce**. Essa sa come accedere ai dati del Modello ma non ne interpreta il significato né conosce le possibili manipolazioni che l'utente può effettuare.

3. **Controller:** si colloca tra la View ed il Model, ascolta gli eventi innescati dalla View e si occupa di manipolare i Model e restituire una risposta all'utente.

Nota bene: Spesso viene aggiunto un ulteriore layer chiamato **Service** all'interno del quale risiede la logica di business. In questo modo il Controller si occupa solamente di ricevere la **Request** e richiamare il Service corretto.



Proviamo a ripercorrere una classica interazione di un utente che prova a loggare su un'app sviluppata con architettura MVC:

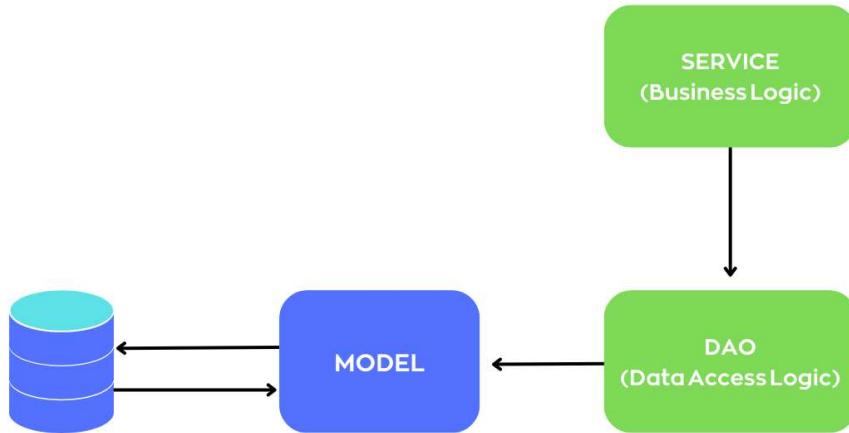
1. L'utente visualizza la View dove è presente il form per il login ed una volta compilato preme sul bottone Login per accedere all'applicazione;
2. Premendo sul bottone l'utente, tramite la View, invia una **Request** (con i dati del form) al Controller;
3. Il Controller riceve la request e richiama il Service che si occupa dell'autenticazione;
4. Il Service interagisce con i Model per verificare che i dati inviati dall'utente corrispondano con le informazioni di accesso presenti sul DB;
5. A seconda che i dati siano presenti o meno viene inviata una **Response** positiva o negativa alla View, la quale si occuperà di mostrare all'utente il risultato della sua Request.

DAO:

Data Access Object, si tratta di un altro pattern architettonale che ci consente di dividere la logica di business dalla **logica di accesso ai dati**, in modo da semplificare la gestione dei dati e migliorare la manutenibilità del codice. In questo modo, se si

decide di cambiare il database utilizzato per l'applicazione, è possibile farlo senza dover modificare l'intera applicazione.

DAO Pattern



Analizziamo il pattern con un esempio:

Supponiamo che all'interno della nostra applicazione sia presente un Model User che rappresenta una tabella del nostro Database. A questo punto andremo a creare una classe chiamata UserDao che al suo interno conterrà tutti i metodi per accedere alla **fonte dati**. In questo caso siccome la nostra fonte dati è rappresentata da una tabella sul Database la classe UserDao sarà composta da tutte le **query** necessarie per accedere a questa tabella.

DTO:

Il Data Transfer Object (DTO) è un pattern di progettazione utilizzato per **trasferire** dati tra vari sottosistemi di un'applicazione software. I DTO sono spesso utilizzati in congiunzione con gli oggetti di accesso ai dati (Data Access Object, DAO) per recuperare i dati da una base di dati.

Il DTO si differenzia dagli oggetti di business o dagli oggetti di accesso ai dati in quanto non ha comportamenti o logiche applicative: **la sua unica responsabilità è quella di trasportare i dati**. In pratica, i DTO sono oggetti che incapsulano i dati necessari per la View.

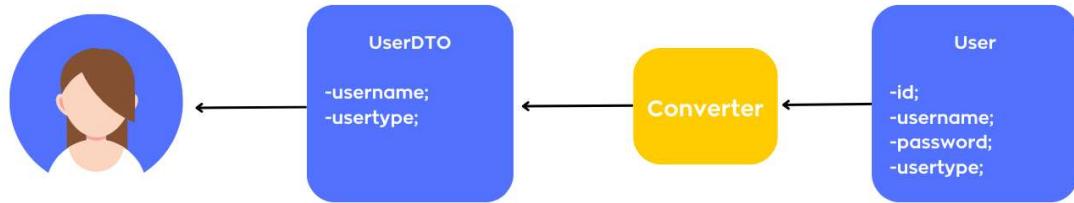
Ad esempio, se un utente desidera visualizzare il suo profilo, sarà necessario recuperare dalla tabella Profilo del database i dati relativi a quell'utente. È possibile, tuttavia, che la tabella Profilo contenga informazioni supplementari non necessarie o

non desiderate per la View. In questo caso, si crea una classe ProfiloDTO che contiene solo i **dati necessari** recuperati dalla tabella Profilo.

Nel pattern DTO, è buona norma creare dei **Converter**, che sono classi specializzate che si occupano di convertire i nostri Model in DTO e viceversa. Questo permette di mantenere separate le responsabilità di rappresentazione dei dati e di trasformazione dei dati, rendendo il codice più pulito e più facile da mantenere.

Inoltre, l'uso di DTO può contribuire a migliorare le prestazioni di un'applicazione riducendo il numero di chiamate remote o la quantità di dati trasferiti su una rete, in quanto consente di raggruppare insieme i dati che devono essere inviati insieme in un unico oggetto DTO.

DTO Pattern



Reflection

Le API reflection che trovate sotto il package `java.lang.reflection` sono un insieme di classi pensate per l'interrogazione **dinamica** di istanze di classi java (proprietà riflessive). In questo modo, a partire da una generica classe possiamo interrogarla e conoscere i nomi dei suoi metodi, dei suoi attributi e di tutto ciò che essa contiene, direttamente **durante la sua esecuzione**. Addirittura è possibile interrogarne i metodi, o invocarli, come se ci si trovasse in un normale flusso di esecuzione.

Con la reflection, è possibile:

- Ottenere i nomi di classi, metodi, campi, costruttori, interfacce ecc.

- Creare istanze di classi in modo dinamico.
- Accedere a campi, metodi e costruttori in modo dinamico.
- Caricare classi dinamicamente e utilizzare le relative istanze.

Esempi di Reflection:

- **Ottenere informazioni su una classe:**

Il primo passo per utilizzare la riflessione è ottenere un oggetto **Class** che rappresenta la classe di cui si desidera ottenere informazioni. Questo può essere fatto in vari modi. Ecco degli esempi:

```
Class<?> myClass = MyClass.class;
```

Si può anche ottenere un oggetto Class da un'istanza di un oggetto:

```
MyClass myObject = new MyClass();
Class<?> myClass = myObject.getClass();
```

O, si può ottenere un oggetto Class dal nome della classe:

```
String className = "com.example.MyClass";
Class<?> myClass = Class.forName(className);
```

Una volta ottenuto l'oggetto Class, è possibile ottenere informazioni su metodi, campi, costruttori ecc.

Ad esempio:

```
Constructor<?>[] constructors =
myClass.getConstructors();
Method[] methods = myClass.getMethods();
Field[] fields = myClass.getFields();
```

- **Creazione dinamica di istanze:**

Con la riflessione, è possibile creare dinamicamente istanze di classi. Questo può essere fatto ottenendo un costruttore della classe e invocandolo:

```
Class<?> myClass = MyClass.class;
Constructor<?> constructor =
myClass.getConstructor();
MyClass myObject = (MyClass)
constructor.newInstance();
```

- **Accedere a campi e metodi in modo dinamico:**

Con la riflessione, è possibile accedere e modificare i campi di un oggetto e invocare i suoi metodi in modo dinamico. Ad esempio:

```
 MyClass myObject = new MyClass();
Class<?> myClass = myObject.getClass();

// Accedere a un campo
Field field = myClass.getField("myField");
field.set(myObject, newValue);

// Invocare un metodo
Method method = myClass.getMethod("myMethod");
method.invoke(myObject);
```

Si noti che queste operazioni potrebbero violare le regole di accesso (ad esempio, accedere a un campo privato o invocare un metodo privato), ma la reflection permette di farlo. Tuttavia, si dovrebbe usare questa funzionalità con cautela.

In generale, la reflection è uno strumento potente che consente di realizzare codice molto flessibile e dinamico. Tuttavia, ha i suoi svantaggi: è più lenta rispetto alle operazioni non riflettive, e l'uso improprio può portare a codice difficile da comprendere e mantenere. Pertanto, si dovrebbe usare la reflection solo quando è realmente necessario.

Git

Git è il sistema di controllo delle versioni moderno di gran lunga più utilizzato attualmente a livello globale. Git è un progetto open source maturo e gestito attivamente che è stato originariamente sviluppato nel 2005 da Linus Torvalds, il famoso creatore del kernel del sistema operativo Linux. Un numero impressionante di progetti software si affida a Git per il controllo delle versioni, inclusi i progetti commerciali e open source. Gli sviluppatori che hanno usato Git rappresentano una fetta importante del pool di talenti dello sviluppo software presenti nel mondo del lavoro e questo strumento funziona bene su un'ampia gamma di sistemi operativi e IDE (Integrated Development Environment).

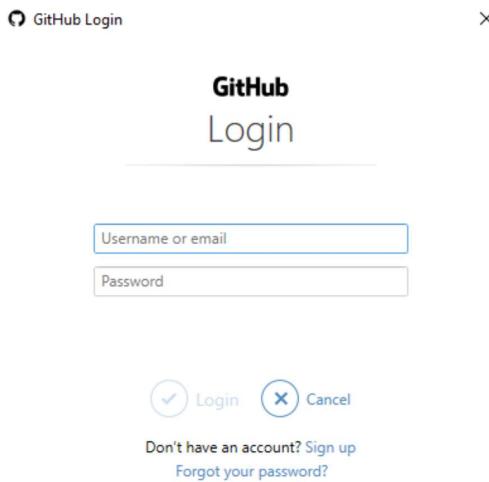
Grazie all'architettura distribuita, Git è un esempio di DVCS (Distributed Version Control System, sistema di controllo delle versioni distribuito). Piuttosto che avere un'unica posizione per la cronologia completa delle versioni del software, come è comune nei sistemi di controllo delle versioni un tempo popolari come CVS o Subversion (noto anche come SVN), in Git la copia di lavoro del codice di ogni

sviluppatore è anche un repository che può contenere la cronologia completa di tutte le modifiche.

Oltre ad essere distribuito, Git è stato progettato per fornire prestazioni, sicurezza e flessibilità.

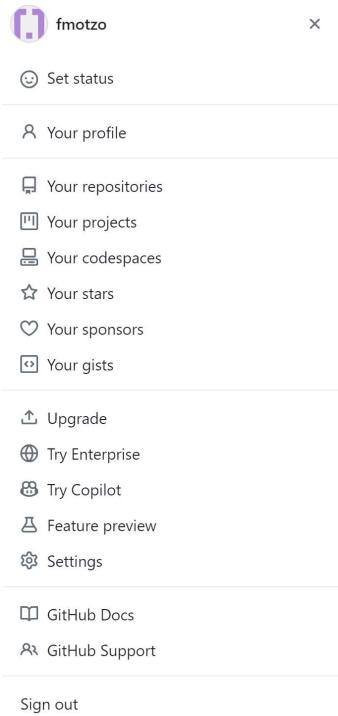
Setup

Come prima cosa è necessario creare un account sulla piattaforma GitHub <https://github.com/signup>. Una volta creato l'account avremo la possibilità di accedere a tutte le nostre repository. Successivamente andiamo ad installare Git sul nostro pc link <https://git-scm.com/download/win>. A questo punto potremo utilizzare Git direttamente da linea di comando. La prima volta che proveremo ad eseguire un comando da terminale, ad esempio per clonare una nostra repository sul pc, dovremo effettuare l'autenticazione.

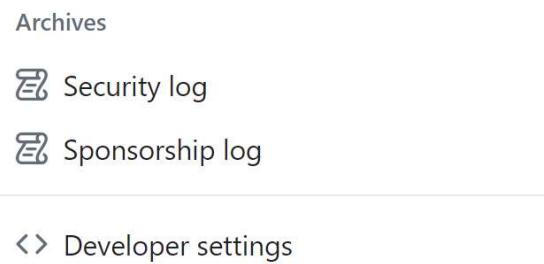


Nota Bene: la password che vi viene richiesta in questo passaggio non corrisponde alla password del vostro account ma al token di accesso che dovrete creare. Vediamo come:

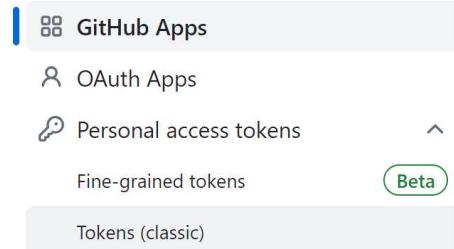
1. Cliccate sull'immagine del vostro profilo e dal menu a discesa selezionate “Settings”



2. Successivamente nella lista di opzioni a sinistra seleziona l'ultima “Developer settings”



3. Poi premete su “Personal Access Tokens” e subito dopo su “Tokens (classic)”



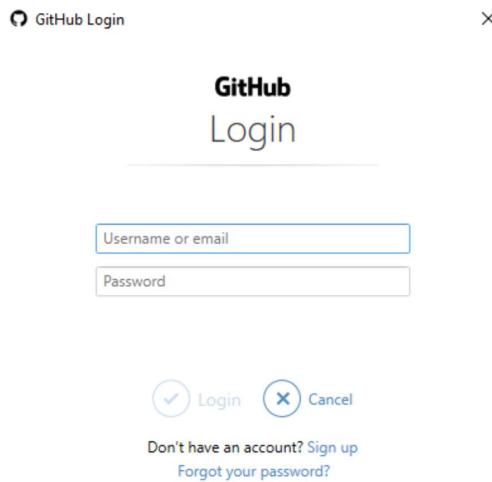
4. Subito dopo andate su “Generate new token (classic)”

The screenshot shows the GitHub 'Personal access tokens (classic)' page. At the top right are two buttons: 'Generate new token' (with a dropdown arrow) and 'Revoke all'. Below them is a table with a single row. The row contains a 'Generate new token' button with a 'Beta' badge, a description 'Fine-grained, repo-scoped', and a note 'For general use'. To the right of the table is a 'Delete' button.

5. Inserite un nome per il vostro token, selezionate i permessi (potete mettere tutti i permessi), e infine premete su “Generate token”

The screenshot shows a 'Generate token' dialog. It has two checkboxes: 'write:ssh_signing_key' (checked) and 'read:ssh_signing_key' (checked). Below the checkboxes are two buttons: 'Generate token' (green) and 'Cancel' (blue).

6. A questo punto avrete il vostro token personale che potrete copiare ed utilizzare per effettuare l’accesso nella schermata vista in precedenza.



Andrà inserito nel campo password!

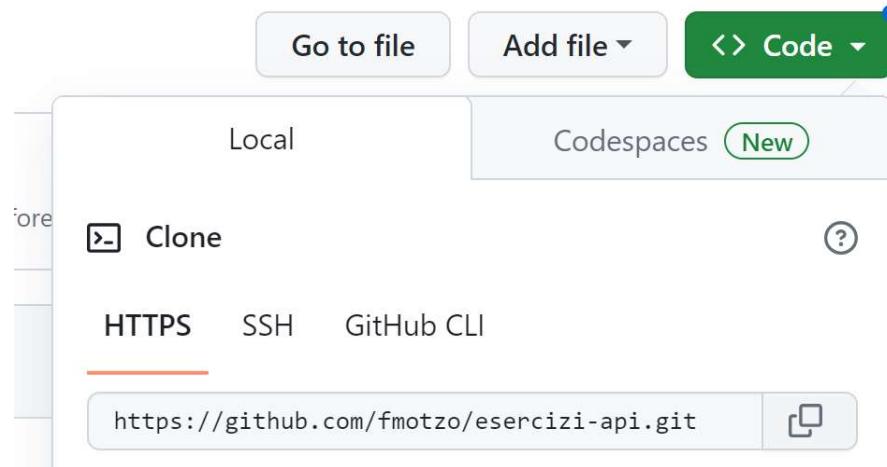
Comandi Principali

Ora che avete installato Git ed effettuato l’accesso potete iniziare a lavorare sulle vostre repository. Per iniziare a lavorare su una repository è necessario eseguire il comando **git clone**, utilizzato per creare una copia locale di un repository Git remoto. Questo comando crea una nuova directory sul tuo computer, inizializza un nuovo repository Git in quella directory, e poi scarica tutti i dati del repository remoto in quella nuova directory locale.

La sintassi generale del comando git clone è la seguente:

git clone <url>

Dove <url> è l'URL del repository Git remoto che vuoi clonare, che possiamo andare a recuperare su GitHub nella pagina principale della repository premendo su "Code":



Dunque ad esempio per clonare questa repository bisognerà aprire il terminale all'interno della cartella nella quale vogliamo clonare il nostro progetto ed inviare il comando

git clone <https://github.com/fmotzo/esercizi-api.git>

Una volta clonato il progetto possiamo iniziare a lavorarci. La repository del vostro progetto conterrà diversi branch, uno per ogni sprint. Ma cosa si intende con branch?

In Git, un "branch" è essenzialmente un puntatore univoco a una serie di commit.

Potete pensare a un branch come a una strada alternativa nello sviluppo del progetto. Quando create un nuovo branch, state creando un ambiente dove poter sperimentare, sviluppare nuove funzionalità o correggere bug senza intaccare il branch principale, che rimane stabile e sicuro da cambiamenti inaspettati.

Dunque quando si lavora in team ad un progetto è buona norma andare a creare dei branch separati per evitare di intaccare il lavoro degli altri. Il comando per creare un nuovo branch è il seguente:

git checkout -b <nome_del_nuovo_branch> <nome_del_branch_esistente>

In questo comando, <nome_del_nuovo_branch> è il nome del nuovo branch che stai creando, e <nome_del_branch_esistente> è il nome del branch da cui stai creando il nuovo branch.

In questo modo avremo creato un branch a partire da un branch già esistente che conterrà quindi tutto il codice del branch originale sul quale potremo andare ad apportare le nostre modifiche senza andare a modificare il principale.

Ma cosa succede quando ho finito di effettuare la mia modifica e voglio aggiungerla al branch principale?

Innanzitutto una volta che la modifica è stata testata ed è terminata dobbiamo procedere a caricarla sulla repository remota, permettendo così agli altri membri del team di vedere e accedere alle tue modifiche. I passaggi da seguire sono i seguenti:

1. “git add .”:

Il comando “git add” è usato per aggiungere i file alla “staging area” di Git, che è un’area di preparazione per i commit. Dopo aver fatto delle modifiche ai tuoi file, è possibile usare git add per indicare i cambiamenti da includere nel prossimo commit, aggiungendo “.” andremo ad includere tutte le modifiche fatte.

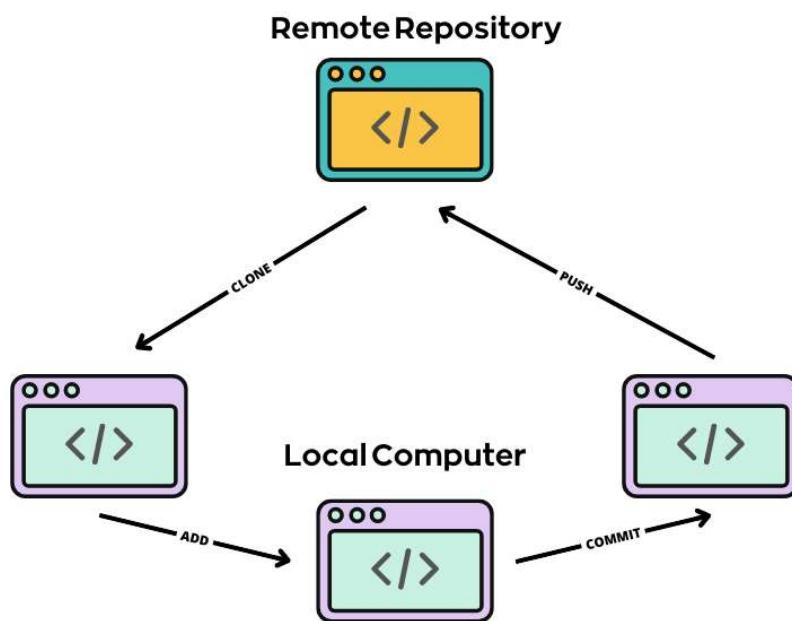
2. “git commit”:

Il comando git commit viene usato per creare un nuovo commit, che è essenzialmente un istantanea dei cambiamenti aggiunti con git add. Ogni commit ha un messaggio associato che descrive le modifiche che contiene, che deve essere fornito quando si esegue il comando git commit. Ecco come utilizzarlo:

git commit -m <messaggio>: crea un nuovo commit con il messaggio fornito.

3. “git push origin <branch>”:

Con il comando git push andiamo effettivamente a caricare le nostre commit sul branch selezionato presente sulla repository remota rendendole visibili e disponibili a tutti.



Una volta che abbiamo terminato di testare la nuova modifica dobbiamo riportarla sul branch principale, dobbiamo effettuare il cosiddetto **merge**.

Vediamo un esempio:

Supponiamo che il nostro branch principale sia il branch “console” e che ci sia la necessità di sviluppare una funzionalità che permetta di inserire un nuovo prodotto sul DataBase. Ecco i passaggi da seguire:

1. Creare un nuovo branch chiamato “newProductFeature” partendo dal principale “console” tramite il comando
git checkout -b “newProductFeature” “console”
2. Ora che abbiamo creato il branch e ci siamo spostati su di esso, possiamo iniziare a sviluppare la nostra feature;
3. Al termine dello sviluppo, dopo aver svolto tutti i test, possiamo andare a pushare le nostre modifiche in remoto eseguendo i seguenti comandi:
git add .
git commit -m “Sviluppo funzionalità aggiunta prodotto terminato”
git push
4. A questo punto il nostro branch “newProductFeature” creato in precedenza conterrà la nuova feature che sarà **accessibile** a tutti i membri della repository.
5. Non resta che integrare la nuova funzionalità nel branch principale “console” eseguendo i comandi:
Cambia al branch principale
git checkout console

```
# Unisci il branch "newProductFeature" nel branch principale  
git merge newProductFeature
```

A questo punto tutti i cambiamenti nel branch “newProductFeature” che non erano già nel branch principale vengono uniti nel branch “console”.

Conflitti:

A volte, quando si cerca di unire due branch, ci possono essere dei **“conflitti di merge”**. Questo accade quando lo stesso file è stato modificato in **entrambi** i branch e Git non sa quale versione del file mantenere.

Quando si verifica un conflitto, Git inserisce dei **marcatori** nel file per aiutarci a identificare le parti del codice che sono in conflitto. Ecco un esempio di come potrebbe apparire un conflitto:

```
<<<<< HEAD  
// Questo è il codice nel tuo branch corrente  
int x = 10;  
=====  
// Questo è il codice nel branch che stai cercando di unire  
int x = 20;  
>>>>> branch-da-unire
```

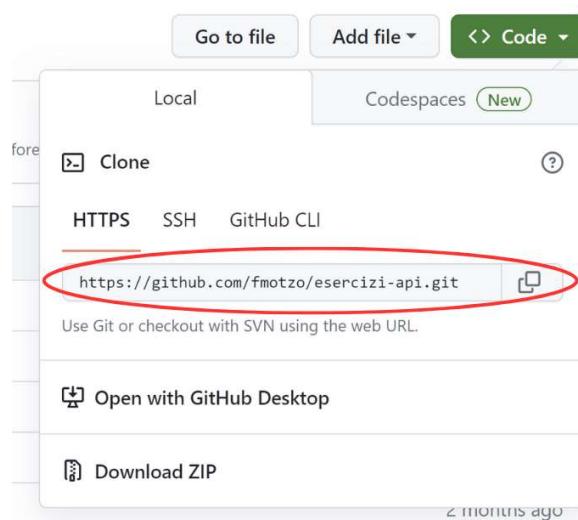
Per risolvere il conflitto, dovremo modificare il file per decidere quale versione del codice **mantenere**, quindi aggiungere il file all'area di staging con **git add** e fare un commit con **git commit** per finalizzare la risoluzione del conflitto.

Analisi del codice

In questa sezione andremo ad analizzare il codice della repository riguardante lo sprint **Java console**, in particolare le classi ed i flussi principali.

Prima però andiamo a vedere come effettuare il setup della nostra applicazione:

1. Come prima cosa sarà necessario andare a **clonare** la repository di Git sul vostro computer. Per fare ciò dovrete accedere alla vostra repository su GitHub e andare a copiare il link per clonare il progetto in questo modo:



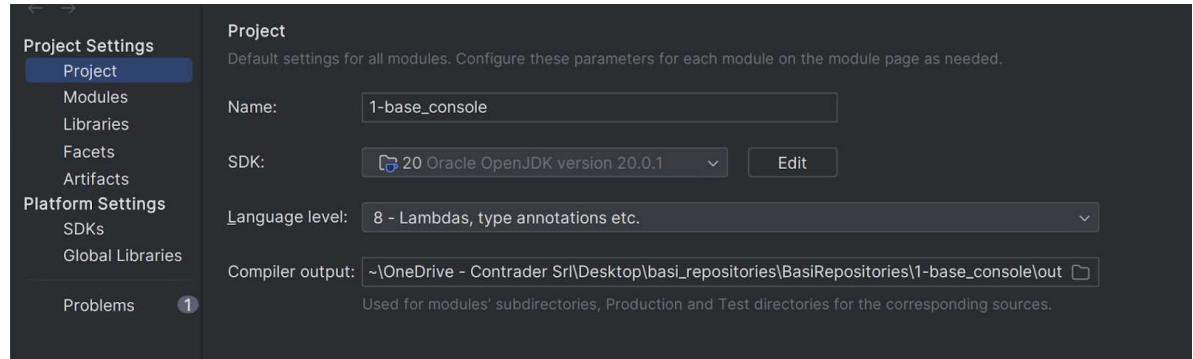
2. Successivamente andremo a clonare la nostra repository tramite il comando **git clone**. Dunque apriamo il terminale all'interno della cartella dove vogliamo clonare il progetto ed inseriamo il comando git clone + "link repository":

```
Command Prompt
Microsoft Windows [Version 10.0.22621.1848]
(c) Microsoft Corporation. All rights reserved.

C:\Users\fmotz>git clone https://github.com/fmotzo/esercizi-api.git
```

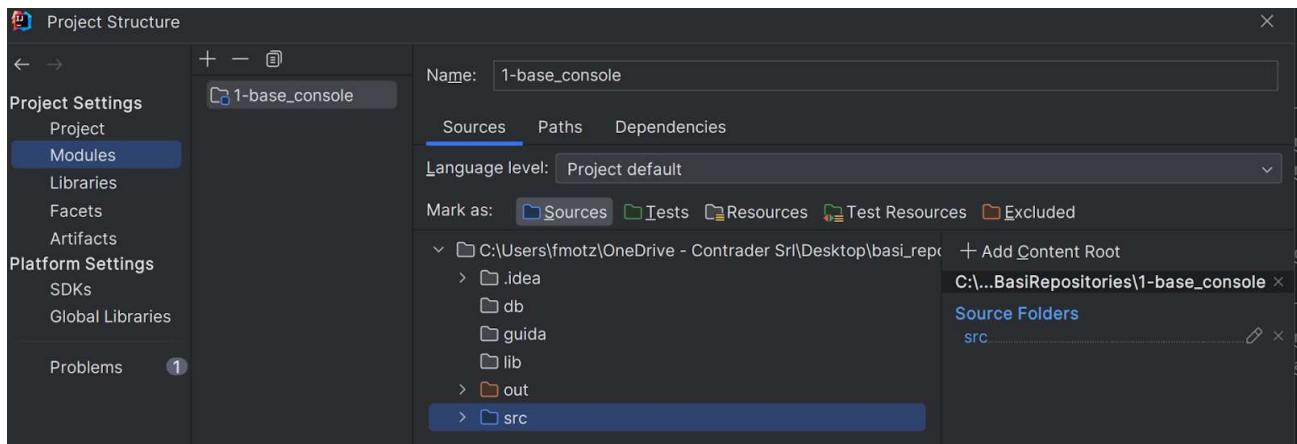
A screenshot of a Windows Command Prompt window. The title bar says 'Command Prompt'. The window shows the following text:
Microsoft Windows [Version 10.0.22621.1848]
(c) Microsoft Corporation. All rights reserved.
C:\Users\fmotz>git clone https://github.com/fmotzo/esercizi-api.git

- Una volta clonato avremo la cartella del nostro progetto che dovremo andare ad aprire su IntelliJ. Aperto il progetto dovremo andare su **Project Structure** nella sezione **Project** e settare l'**SDK** ed il **Language Level**.



- Infine dovremo andare a importare la **dipendenza** di mysql-connector-java-8.0.15.jar nel nostro progetto. Senza questa libreria non sarà possibile connettersi al nostro db MySQL. Vediamo come fare. Andiamo sulla sezione **Modules** dopodichè selezioniamo il tab **Dependencies** e premiamo sull'icona '+'. Infine selezioniamo il nostro mysql-connector-java-8.0.15.jar che si troverà all'interno della cartella **lib** e applichiamo le modifiche.
- A questo punto vi basterà fare tasto destro sulla classe Application e premere su '**Run**' per far partire l'applicazione.

Nota Bene: Nel caso in cui non dovesse comparire l'opzione di Run premendo con il tasto destro sulla classe Application assicuratevi che nella sezione **Project Structure** -> **Modules** -> **Sources** la cartella **src** sia marcata come **Sources**:



Classi Package Main

Application:

Ogni applicazione Java necessita di una funzione **main()**.

Essa sarà di fatto la **prima funzione** chiamata dalla JVM, la sua mancanza verrà infatti notata dall'interprete che quindi non permetterà l'avvio del programma. Banalmente qualsiasi applicazione a cascata (stile C) potrebbe essere interamente scritta all'interno del main ma questo non avrebbe senso all'interno di un linguaggio di programmazione ad oggetti.

All'interno del main possiamo vedere la seguente istruzione:

```
public static void main(String[] args) {  
    MainDispatcher.getInstance().callView( view: "Login", response: null);  
}
```

Essa richiama il metodo callView della classe **MainDispatcher**(che approfondiremo più avanti), per adesso vi basti sapere che tramite questo comando non appena l'applicazione si avvia veniamo reindirizzati alla prima pagina della nostra applicazione, ovvero il **Login**.

MainDispatcher:

La classe MainDispatcher è una componente fondamentale in un'applicazione che segue il pattern Model-View-Controller (MVC). Essa gestisce **la logica di routing** dell'applicazione, direzionando le richieste agli appropriati **Controller** e inviando le risposte alla giusta **View**. Inoltre, MainDispatcher implementa il pattern **Singleton**, garantendo che ci sia solo una singola istanza di MainDispatcher nell'intera applicazione.

Ecco una descrizione dettagliata dei metodi e degli attributi:

Attributi:

```
private static MainDispatcher instance;
```

Questa è l'unica istanza di MainDispatcher che viene creata nell'applicazione. La visibilità **private** garantisce che l'istanza possa essere acceduta solo attraverso il metodo getInstance().

Metodi:

```
private MainDispatcher() {}
```

Questo è un costruttore privato, che previene l'istanziazione diretta della classe MainDispatcher. Rappresenta un aspetto chiave del pattern Singleton.

```
public static MainDispatcher getInstance()
```

Questo metodo statico fornisce un accesso controllato alla singola istanza di MainDispatcher. Se l'istanza non è ancora stata creata, viene istanziata all'interno di questo metodo richiamando il costruttore privato.

```
public void callAction(Request request)
```

Il metodo `callAction(Request request)` è un metodo della classe `MainDispatcher` che utilizza il concetto di **reflection** (vedere capitolo Reflection sulla parte di teoria) in Java per creare dinamicamente un'istanza di un controller specifico e invocare un metodo su di esso. Il controller da istanziare e il metodo da chiamare sono specificati all'interno dell'oggetto **Request** passato come parametro.

La classe `Request` è una semplice classe contenitore che incapsula le informazioni necessarie per effettuare una richiesta a un controller. Ecco una descrizione più dettagliata dei suoi campi:

- **private String controller:** questo campo contiene il nome del controller che deve essere istanziato e utilizzato per gestire la richiesta.
- **private String method:** questo campo contiene il nome del metodo che deve essere invocato sul controller.
- **private Map<String, Object> body:** questa mappa contiene i dati che devono essere passati al metodo del controller.

Il metodo `callAction(Request request)` inizia creando un'istanza del controller specificato utilizzando la reflection. Poi, se il l'oggetto Controller viene trovato viene invocato il metodo `doControl`, presente all'interno di ogni oggetto Controller.

Se il controller o il metodo non possono essere trovati, o se si verifica un errore durante l'invocazione del metodo, viene stampato un stack trace dell'eccezione.

```
public void callView(String view, Response response)
```

Il metodo ``callView(String view, Response response)`` è un metodo della classe ``MainDispatcher`` che utilizza il concetto di reflection in Java per creare dinamicamente un'istanza di una specifica classe **View** e chiamare una serie di metodi su di essa. La classe View da istanziare è specificata come una stringa nel parametro ``view`` del metodo ``callView``.

La classe ``Response`` è una semplice classe contenitore che incapsula i dati che devono essere restituiti come risultato di una richiesta a un controller. Ecco una descrizione più dettagliata dei suoi campi:

private Map<String, Object> body: questa mappa contiene i dati che devono essere restituiti come risultato della richiesta. Ogni dato è associato a una chiave, che può essere utilizzata per recuperare il dato dalla mappa.

Il metodo ``callView(String view, Response response)`` inizia creando un'istanza della classe View specificata utilizzando la reflection. Successivamente, chiama il metodo ``showResults(response)`` sulla classe View, passando l'oggetto ``Response`` come

argomento. Questo metodo è tipicamente utilizzato per visualizzare i risultati della richiesta all'utente.

Successivamente, il metodo `callView` chiama il metodo `showOptions()` sulla classe View, che può essere utilizzato per visualizzare una serie di opzioni all'utente.

Infine, chiama il metodo `submit()` sulla classe View, che può essere utilizzato per ottenere l'input dell'utente in risposta alle opzioni visualizzate.

Se la classe View non può essere trovata, o se si verifica un errore durante l'invocazione di uno dei metodi, viene generata un'eccezione `RuntimeException`.

In sintesi, la classe `Response` e il metodo `callView(String view, Response response)` forniscono un meccanismo flessibile e dinamico per visualizzare i risultati delle richieste e interagire con l'utente in un'applicazione MVC

ConnectionSingleton:

La classe ConnectionSingleton è un esempio di un design pattern Singleton applicato a un contesto specifico: **la connessione a un database**.

Questa classe è utilizzata per creare una singola connessione a un database che può essere utilizzata in tutto il programma, garantendo l'efficienza e l'integrità dei dati.

Di seguito, i dettagli dei principali componenti di questa classe:

```
private static Connection connection = null;
```

Questo campo statico memorizza l'istanza della connessione al database. È inizialmente impostato su null.

```
private ConnectionSingleton() { }
```

Il costruttore è dichiarato come privato per impedire la creazione di nuove istanze della classe attraverso il costruttore.

```
public static Connection getInstance()
```

Questo metodo è il cuore del pattern Singleton. Controlla se esiste già un'istanza della connessione al database (verificando se `connection` è null) e, se non esiste, la crea. La connessione al database viene creata utilizzando le **proprietà del database** specificate nel file **config.properties**.

```

InputStream inputStream = Files.newInputStream(Paths.get(first: "config.properties"));
properties.load(inputStream);

String vendor = properties.getProperty("db.vendor");
String driver = properties.getProperty("db.driver");
String host = properties.getProperty("db.host");
String port = properties.getProperty("db.port");
String dbName = properties.getProperty("db.name");
String username = properties.getProperty("db.username");
String password = properties.getProperty("db.password");
String jdbcAdditionalParams=properties.getProperty("db.jdbc_params");

```

In particolare, all'interno di `getInstance()`, si utilizza la classe `Properties` per caricare le proprietà del database da un file di configurazione, ovvero il file config.properties. Quindi, queste proprietà vengono utilizzate per creare l'URL di connessione e stabilire la connessione al database utilizzando `DriverManager.getConnection()`, tramite la libreria **mysql-connector-java-8.0.15.jar**.

```

Class<?> c = Class.forName(driver);
String url = "jdbc:" + vendor + ":" + host + ":" + port + "/" + dbName+"?" + jdbcAdditionalParams;
connection = DriverManager.getConnection(url, username, password);

```

Il metodo `getInstance()` restituisce sempre la stessa istanza della connessione al database, garantendo che tutte le parti del programma utilizzino la stessa connessione.

Infine, se si verifica un errore durante la creazione della connessione al database (ad esempio, se il file di configurazione non può essere trovato o se le proprietà del database sono errate), viene generata un'eccezione che viene catturata e il suo stack trace viene stampato.

ReflectionUtils:

La reflection in Java è una funzionalità che permette di esaminare o modificare il comportamento delle classi, delle interfacce, dei campi e dei metodi in tempo di esecuzione. Questa potente funzionalità permette di creare codice molto flessibile e dinamico.

La classe ReflectionUtils fornisce un metodo di utilità per la creazione dinamica di istanze di classi utilizzando la riflessione. Vediamola nei dettagli:

Il metodo **instantiateClass** utilizza la riflessione per creare un'istanza di una classe a partire dal suo nome completo. Prima di tutto, il metodo **Class.forName()** viene

utilizzato per ottenere un oggetto Class corrispondente alla classe specificata. Il nome della classe deve includere il pacchetto.

Una volta ottenuto l'oggetto Class, si ottiene un riferimento al costruttore senza parametri della classe chiamando **getDeclaredConstructor()**. Si presuppone che la classe abbia un costruttore senza parametri; se non è presente, verrà sollevata **un'eccezione**.

Infine, si chiama newInstance() sul costruttore per creare una nuova **istanza** della classe. Se la creazione dell'istanza ha successo, l'oggetto viene restituito. Se si verifica un errore (ad esempio, la classe non esiste, o la classe non ha un costruttore senza parametri), viene catturata un'eccezione e il metodo restituisce null.

Esempio d'uso:

```
Controller oggettoController = (Controller)
ReflectionUtils

.instantiateClass("it.contrader.controller." +
request.getController() + "Controller");
```

In questo caso stiamo utilizzando il metodo instantiateClass per andare a creare un oggetto della classe del controller specificato nella request. Si noti che poiché instantiateClass restituisce un oggetto di tipo Object, è necessario effettuare un **cast** all'oggetto al tipo appropriato prima di utilizzarlo.

La riflessione è uno strumento potente, ma dovrebbe essere usato con cura. È più lento rispetto all'accesso diretto ai costruttori, ai metodi e ai campi, e il suo uso improprio può portare a problemi di sicurezza e manutenibilità. Tuttavia, in alcuni casi, come la **creazione dinamica di oggetti o l'invocazione di metodi**, può essere molto utile.

Flusso di interazione:

In questa sezione andremo ad analizzare un tipico flusso di interazione dell'utente all'interno della nostra applicazione. Prenderemo come esempio il flusso di Login che rappresenta il flusso principale dell'utente appena si avvia l'applicazione.

Come visto precedentemente nella classe main è presente l'istruzione:

```
public static void main(String[] args) {
    MainDispatcher.getInstance().callView( view: "Login", response: null);
}
```

Come detto anche in precedenza viene richiamata la funzione callView del MainDispatcher passando come argomenti “Login”, ovvero il nome della View in cui vogliamo reindirizzare l’utente, e null poichè non abbiamo bisogno di passare nessuna Response da visualizzare nella View.

```
public void callView(String view, Response response) {  
    View oggettoView = (View) ReflectionUtils.instantiateClass( nomeClasse: "it.contrader.view." + view + "View");  
    if(oggettoView != null) {  
        oggettoView.showResults(response);  
        oggettoView.showOptions();  
        oggettoView.submit();  
    } else throw new RuntimeException("404 pagina " + view + " non trovata");  
}
```

Il metodo callView in questo caso sfrutta la potenza di due concetti chiave nella programmazione orientata agli oggetti: la **Reflection** ed il **Polimorfismo**.

Il metodo callView usa la Riflessione per creare dinamicamente un’istanza di una classe View specifica, come LoginView, che risiede nel package it.contrader.view. Questo processo è potente perché permette di creare oggetti a tempo di esecuzione, basandosi su informazioni che possono variare dinamicamente.

Una volta creata l’istanza, callView invoca sequenzialmente i metodi **showResults**, **showOptions** e **submit**.

Ma come fa a sapere che questi metodi esistono per qualsiasi classe che stiamo recuperando tramite il metodo callView? E come siamo sicuri di poter eseguire un casting sicuro a View?

```
no usages  ↗ giampaolotizz *  
public class LoginView extends AbstractView {
```

```
public abstract class AbstractView implements View {  
    13 usages  ↗ giampaolotizz *  
    public String getInput() {  
        Scanner scanner = new Scanner(System.in);  
        return scanner.nextLine();  
    }  
}
```

Consideriamo l'esempio di LoginView. Questa classe estende AbstractView, che a sua volta implementa l'interfaccia View. AbstractView è una classe astratta che definisce il metodo **getInput()**, utilizzato da tutte le sue sottoclassi per ricevere l'input dell'utente.

```
public interface View {  
    8 implementations new *  
    void showResults (Response response);  
    8 implementations new *  
    void showOptions ();  
    8 implementations new *  
    void submit();  
}
```

La vera magia, tuttavia, sta nell'**interfaccia View**. In View, definiamo i metodi showResults, showOptions e submit solamente in termini di tipo di ritorno, nome e parametri, senza fornire un'implementazione specifica. Saranno le classi concrete che implementano questa interfaccia a fornire l'implementazione di questi metodi.

```

public class LoginView extends AbstractView {

    2 usages
    private String username;

    2 usages
    private String password;

    new *
    public void showResults(Response response) {
    }

    /**
     * Chiede in input all'utente uno username e una password usando il metodo getInput() presente in AbstractView
     */
    ▲ giampaolotizz
    public void showOptions() {

        System.out.println("----- .:LOGIN:. -----");

        System.out.println(" Nome utente:");
        this.username = getInput();

        System.out.println(" Password:");
        this.password = getInput();
    }

    /**
     * Impacchetta una request (metodo request.put("chiave", valore)) e la manda al controller Home tramite il Dispatcher
     */
    ▲ giampaolotizz
    public void submit() {
        LoginDTO loginDTO = new LoginDTO(username, password);
        Request request = new Request();
        request.getBody().put("LoginDTO", loginDTO);
        request.setController("Home");
        request.setMethod("Login");
        MainDispatcher.getInstance().callAction(request);
    }

}

```

Ma perché si fa tutto ciò? Qual è il vantaggio di definire un'interfaccia con solamente la firma dei metodi e senza un'implementazione precisa?

Creando un'interfaccia, possiamo trattare tutte le classi che la implementano come se fossero dello **stesso tipo**, in questo caso View. Questo ci permette di creare un metodo, come callView, che può operare su un'istanza di View indipendentemente dalla classe specifica che l'ha implementata. Questo si traduce in un codice più generico, flessibile ed estendibile, un vero totem della programmazione orientata agli oggetti.

Possiamo fare un discorso simile per quanto riguarda il metodo callAction che in questo caso agisce su oggetti di tipo Controller e richiama il metodo doControl definito all'interno dell'interfaccia **Controller**.

```
public void callAction(Request request) {
    Controller oggettoController = (Controller) ReflectionUtils
        .instantiateClass( nomeClasse: "it.contrader.controller." + request.getController() + "Controller");
    try {
        if(oggettoController != null) {
            oggettoController.doControl(request);
        } else throw new RuntimeException("404 controller " + request.getController() + " non trovato");
    } catch (Throwable e) {
        e.printStackTrace();
    }
}
```

```
public interface Controller {
    1 usage  2 implementations  ↗ giampaolizz
        public void doControl(Request request);
}
```

Terminato di esaminare in dettaglio i metodi del MainDispatcher proseguiamo con il flusso di Login. Come detto verrà inizialmente richiamata la LoginView e come visto nel metodo callView verranno richiamati in ordine i metodi showResults, showOptions e submit. Ma cosa fanno effettivamente questi metodi all'interno della classe LoginView?

```
    ...
    public void showResults(Response response) {
    }
```

Nel caso della LoginView il metodo showResults sarà vuoto in quanto non c'è la necessità di mostrare nessun risultato all'utente essendo la prima View con cui esso interagisce.

```
public void showOptions() {

    System.out.println("----- .:LOGIN:. -----");

    System.out.println(" Nome utente:");
    this.username = getInput();

    System.out.println(" Password:");
    this.password = getInput();
}
```

Questo metodo andrà a simulare il form di login di una classica applicazione nel quale l'utente andrà ad inserire le sue **credenziali** per l'accesso. Gli input dell'utente verranno salvati nelle variabili di classe **username** e **password** tramite il metodo `getInput()` della classe astratta.

```
public void submit() {  
    LoginDTO loginDTO = new LoginDTO(username, password);  
    Request request = new Request();  
    request.getBody().put("loginDTO", loginDTO);  
    request.setController("Home");  
    request.setMethod("Login");  
    MainDispatcher.getInstance().callAction(request);  
}
```

Infine abbiamo il metodo **submit** che si occuperà di impacchettare le informazioni all'interno di un **DTO** ed inviare una **request**, tramite il metodo `callAction`, al Controller Home richiedendo l'operazione di “**login**”. Come già ripetuto in precedenza il metodo `callAction` si occuperà di istanziare l'oggetto Controller corrispondente alla Request passata e richiamare il metodo `doControl` al suo interno passando la Request. In questo caso verrà richiamato il metodo `doControl` della classe `HomeController`. Analizziamo questo metodo:

```

public void doControl(Request request) {
    if (request != null && request.getMethod().equals("login")) {

        switch (request.getMethod().toUpperCase()) {
            case "LOGIN":
                UserDTO userDTO = loginService.login((LoginDTO) request.getBody().get("loginDTO"));

                if (userDTO != null) {
                    Response response = new Response();
                    response.put("username", userDTO.getUsername());
                    switch (userDTO.getUserType()) {

                        case "ADMIN":
                            MainDispatcher.getInstance().callView( view: "HomeAdmin", response);
                            break;

                        case "USER":
                            MainDispatcher.getInstance().callView( view: "HomeUser", response);
                            break;

                        default:
                            MainDispatcher.getInstance().callView( view: "Login", response: null);
                            break;
                    }
                } else {
                    System.out.println("Credenziali errate");
                    MainDispatcher.getInstance().callView( view: "Login", response: null);
                }
                break;
            case "REGISTER":
                //Buona Implementazione ;
                break;
            default:
                System.out.println("Nessun metodo corrisponde al metodo inserito: " + request.getMethod());
                MainDispatcher.getInstance().callView( view: "Login", response: null);
        }
    } else MainDispatcher.getInstance().callView( view: "Login", response: null);
}

```

All'inizio, il metodo controlla se l'oggetto Request ricevuto come argomento è diverso da null. Se la richiesta è null, allora il metodo richiama la View Login attraverso MainDispatcher.getInstance().callView("Login", null);.

Se la richiesta non è null, viene recuperato il metodo dalla richiesta (**request.getMethod()**), lo si converte in maiuscolo (toUpperCase()) e si esegue uno switch basato su di esso.

Se il metodo della richiesta è "LOGIN", il sistema tenta di effettuare il login dell'utente utilizzando un servizio di login (**loginService**). La richiesta deve contenere un oggetto LoginDTO nel suo body, che viene passato al servizio di login.

```

public UserDTO login (LoginDTO loginDTO) {
    return userConverter.toDTO(this.loginDAO.login(loginDTO.getUsername(), loginDTO.getPassword()));
}

```

Il servizio di login è il livello in cui si gestisce la logica di business dell'autenticazione. Il metodo login accetta un oggetto LoginDTO (Data Transfer Object) che contiene il nome utente e la password forniti dall'utente che tenta di effettuare l'accesso. Successivamente richiama il metodo login del **DAO** che andrà a verificare se le credenziali inserite corrispondono a quelle che risiedono sul DB.

```
private final String QUERY_LOGIN = "SELECT * FROM user WHERE username = ? AND password = ?";  
  
1 usage  ▲ giampaolotizz*  
public User login (String username, String password) {  
  
    Connection connection = ConnectionSingleton.getInstance();  
    try {  
        PreparedStatement statement = connection.prepareStatement(QUERY_LOGIN);  
  
        statement.setString( parameterIndex: 1, username);  
        statement.setString( parameterIndex: 2, password);  
  
        User user = null;  
  
        ResultSet resultSet;  
  
        if(statement.executeQuery().next()) {  
            resultSet = statement.executeQuery();  
            resultSet.next();  
            user = new User(resultSet.getInt( columnLabel: "id"), resultSet.getString( columnLabel: "username"),  
                resultSet.getString( columnLabel: "password"), resultSet.getString( columnLabel: "usertype"));  
        }  
  
        return user;  
    }  
  
    catch (SQLException e) {  
  
        return null;  
    }  
}
```

Nel metodo login, viene creata una connessione al database e viene preparata una query SQL per cercare un utente che corrisponda alle credenziali fornite. La query è un **PreparedStatement**, il che significa che può accettare parametri al momento dell'esecuzione. I parametri sono forniti tramite i metodi `setString`.

Una volta eseguita la query, i risultati vengono estratti e utilizzati per creare un nuovo oggetto User. Se la query non restituisce risultati (ovvero, nessun utente corrisponde alle credenziali fornite), l'oggetto User rimane **null**.

Una volta ottenuto l'oggetto User questo viene convertito attraverso il **converter** in DTO e restituito al Controller. Il metodo quindi crea una nuova Response, aggiunge l'username dell'utente alla risposta e controlla il tipo di utente. In base al tipo

di utente (ADMIN o USER), chiama la vista corrispondente attraverso il MainDispatcher. Il flusso del Login termina quando si arriva alla View che mostrerà il risultato contenuto nella response e successivamente le diverse opzioni a seconda del tipo di Utente loggato.

```
public void showResults(Response response) {
    if (response != null) {
        System.out.println("\n Benvenuto in SAMPLE PROJECT " + response.getBody().get("username").toString() + "\n");
    }
}
```