Test di Progettazione Informatica - Risposte

Differenza tra classe e oggetto

Classe

- La classe è un modello o schema che definisce proprietà (attributi) e comportamenti (metodi) di un tipo di oggetto.
- È un concetto astratto che serve come "ricetta" per creare oggetti.

Oggetto

- L'oggetto è un'istanza concreta di una classe. Rappresenta un'entità reale creata seguendo il modello della classe.

Esempio:

```
// Classe
class Automobile {
    String colore;
    void avvia() {
        System.out.println("Automobile avviata");
    }
}

// Oggetto
Automobile auto1 = new Automobile();
auto1.colore = "Rosso";
auto1.avvia();
```

Interfacce in Java

Definizione

Le **interfacce** in Java sono un contratto che definisce un insieme di metodi (senza implementazione) che una classe deve implementare.

Motivazioni:

1. **Astrazione**: Permettono di definire cosa una classe dovrebbe fare, senza specificare come.
2. **Polimorfismo**: Consentono a diverse classi di essere trattate in modo uniforme.
3. **Ereditarietà multipla**: Una classe può implementare più interfacce, superando il limite dell'ereditarietà multipla con classi.

Sintassi:

1. **Dichiarazione di un'interfaccia**:

```
public interface Animale {
    void mangia(); // Metodo astratto
    void dorme();
}
```

2. **Implementazione di un'interfaccia**:

```
public class Cane implements Animale {
    @Override
    public void mangia() {
        System.out.println("Il cane mangia ossa.");
    }

    @Override
    public void dorme() {
        System.out.println("Il cane dorme nella cuccia.");
    }
}
```

3. **Utilizzo**:

```
Animale animale = new Cane();
animale.mangia();
animale.dorme();
```

Caratteristiche:

- Da Java 8: le interfacce possono avere metodi **default** (con implementazione) e **statici**.
- Da Java 9: supportano metodi **privati**.

Static e stato della memoria

Definizione di `static`

La parola chiave `static` indica che un membro (variabile o metodo) appartiene alla classe e non alle istanze.

Proprietà di `static`:

1. Esiste **un'unica copia** condivisa da tutte le istanze della classe.
2. Può essere utilizzato senza creare un'istanza della classe.

Esempio di utilizzo di `cont`

La variabile `cont` potrebbe essere un **contatore globale** per assegnare un ID univoco alle istanze.

```
class Persona {
    String nome;
    String cognome;
    int id;
    static int cont = 1000; // Variabile statica condivisa

    Persona(String nome, String cognome) {
        this.nome = nome;
        this.cognome = cognome;
        this.id = cont++; // Assegna l'ID univoco e incrementa cont
    }
}
```

Stato della memoria dopo p1 e p2

Codice della classe `Logica`:

```
class Logica {
    Persona p1 = new Persona("Mario", "Rossi");
    Persona p2 = new Persona("Luigi", "Verdi");
}
```

Heap (Oggetti creati):

- `p1`:
    - `nome = "Mario"`
    - `cognome = "Rossi"`
    - `id = 1000`
- `p2`:
    - `nome = "Luigi"`
    - `cognome = "Verdi"`
    - `id = 1001`

Area statica della classe:

- `Persona.cont = 1002` (incrementato ad ogni istanza).

Design Pattern - Low Coupling e High Cohesion

Design Pattern

I **design pattern** sono soluzioni riutilizzabili a problemi comuni nell'ingegneria del software. Forniscono linee guida per scrivere codice più leggibile, manutenibile e scalabile.

Low Coupling (Basso Accoppiamento)

Definizione:

- Si riferisce al grado di dipendenza tra classi o moduli. Con basso accoppiamento, i moduli sono meno interdipendenti, facilitando manutenzione e modifiche.

Violazione:

Una classe dipende direttamente da un'altra, aumentando l'accoppiamento:

```
class A {
    B b = new B(); // Dipendenza diretta
    void doSomething() {
        b.action();
    }
}

class B {
    void action() {
        System.out.println("Azione in B");
    }
}
```

Soluzione: Usare un'interfaccia per ridurre l'accoppiamento:

```
interface Azione {
    void action();
}

class B implements Azione {
    @Override
    public void action() {
        System.out.println("Azione in B");
    }
}

class A {
    Azione azione;
    A(Azione azione) {
        this.azione = azione;
    }
    void doSomething() {
        azione.action();
    }
}
```

High Cohesion (Alta Coesione)

Definizione:

- Si riferisce al grado in cui i metodi di una classe lavorano insieme per raggiungere un obiettivo comune.
- Una classe con alta coesione ha una responsabilità ben definita.

Violazione:

Una classe ha troppe responsabilità:

```
class Ordine {
    void calcolaTotale() {
        // Calcolo totale ordine
    }

    void inviaEmailConferma() {
        // Logica per inviare email
    }

    void generaFattura() {
        // Genera fattura
    }
}
```

Problema: La classe si occupa di calcoli, comunicazioni e fatturazione, violando il principio di responsabilità unica.

Soluzione: Separare le responsabilità in più classi:

```
class Ordine {
    void calcolaTotale() { /* Calcolo */ }
}

class EmailService {
    void inviaEmailConferma() { /* Invio */ }
}

class FatturaService {
    void generaFattura() { /* Generazione */ }
}
```