Risposte mock exam 09/06/2025

Tutorato MDP-MGC

Descrivere il principio di sostituzione di Liskov mostrando un esempio di violazione.

Il principio di sostituzione di Liskov (Liskov Substitution Principle o LSP) è un principio fondamentale della programmazione orientata agli oggetti, che afferma che **gli oggetti di una classe derivata dovrebbero poter essere utilizzati al posto degli oggetti della classe base senza che questo causi problemi per il programma.**
In altre parole, se S è una sottoclasse di T, allora ogni istanza di S dovrebbe poter essere utilizzata al posto di un'istanza di T senza causare problemi di tipo o di comportamento. Ciò significa che ogni metodo della classe T dovrebbe funzionare correttamente quando chiamato su un'istanza di S.
Ad esempio, se abbiamo una classe base "Animal" e una classe derivata "Dog", allora ogni metodo della classe "Animal" dovrebbe funzionare correttamente quando chiamato su un oggetto della classe "Dog". In questo modo, possiamo utilizzare gli oggetti della classe "Dog" al posto degli oggetti della classe "Animal" in tutte le parti del programma in cui viene utilizzata la classe "Animal".
Per poter capire meglio questo principio, prendiamo un esempio di violazione:

```
/**
 * La classe base definisce il "contratto" di un conto generico:
 * si può depositare e si può prelevare, a patto di avere fondi sufficienti.
 */

class Conto {
    protected double saldo;

    public Conto(double saldoIniziale) {
        this.saldo = saldoIniziale;
    }

    public void deposita(double importo) {
        if (importo > 0) {
            this.saldo += importo;
        }
    }

    /**
     * Il contratto di questo metodo è chiaro: se i fondi sono sufficienti,
     * il saldo viene diminuito. Altrimenti, viene segnalato un errore.
     */
    public void preleva(double importo) {
        if (importo <= this.saldo) {
            this.saldo -= importo;
        } else {
            throw new IllegalArgumentException("Fondi insufficienti.");
```

```
        }
    }

    public double getSaldo() {
        return this.saldo;
    }
}


/**
 * Sottoclasse che rappresenta un conto vincolato.
 * Logicamente, "è un tipo di" conto, ma ha una restrizione chiave: non si può prelevare.
 */
class ContoDepositoVincolato extends Conto {

    public ContoDepositoVincolato(double saldoIniziale) {
        super(saldoIniziale);
    }

    /**
     * QUI AVVIENE LA VIOLAZIONE LSP.
     * Per implementare la restrizione, modifichiamo il comportamento del metodo ereditato.
     * Invece di prelevare, lanciamo un'eccezione che il contratto della classe base non
prevedeva.
     * Questo cambia radicalmente il comportamento atteso da un 'Conto'.
     */
    @Override
    public void preleva(double importo) {
        throw new UnsupportedOperationException("I prelievi non sono permessi per i conti
deposito vincolati.");
    }
}


public class EsempioViolazioneLSP {
    public static void main(String[] args) {
        // Un client che lavora con l'astrazione 'Conto' si aspetta un certo comportamento.
        Conto contoNormale = new Conto(1000.0);
        System.out.println("Operazioni su conto normale:");
        processaConto(contoNormale);

        System.out.println("\nOperazioni su conto vincolato:");
        // Provo a sostituire la classe base con la sottoclasse.
        Conto contoVincolato = new ContoDepositoVincolato(5000.0);
        processaConto(contoVincolato); // Il programma si comporterà in modo inaspettato.
    }

    /**
```

```
     * Questo metodo client si fida del contratto della classe Conto.
     * Non sa, e non dovrebbe aver bisogno di sapere, quale tipo specifico di conto sta gestendo.
     */
    public static void processaConto(Conto conto) {
        try {
            System.out.println("Saldo attuale: " + conto.getSaldo());
            System.out.println("Tento un prelievo di 100 euro...");
            conto.preleva(100.0);
            System.out.println("Prelievo riuscito. Nuovo saldo: " + conto.getSaldo());
        } catch (Exception e) {
            // Questa eccezione dovrebbe verificarsi solo per 'Fondi insufficienti',
            // secondo il contratto di 'Conto'. L'UnsupportedOperationException è una sorpresa.
            System.err.println("ERRORE INASPETTATO: " + e.getMessage());
        }
    }
}
```

Descrivere ruolo dei metodi equals, toString e hashcode definiti nella classe Object ed il loro contratto d'uso.

equals : il metodo equals è utilizzato per confrontare due oggetti per uguaglianza. È definito nella classe Object e deve essere sovrascritto nelle classi che vogliono definire il concetto di uguaglianza. Il metodo equals prende un oggetto come parametro e restituisce un valore booleano che indica se l'oggetto corrente è uguale all'oggetto passato come parametro.
Esso deve essere

- **consistente**: deve restituire sempre lo stesso risultato per gli stessi oggetti;
- **riflessivo**: deve restituire true se l'oggetto è uguale a se stesso;
- **simmetrico**: restituire true se due oggetti sono uguali in entrambe le direzioni (cioè a.equals(b) è vero se e solo se b.equals(a) è vero);
- **transitivo**: deve restituire true se tre oggetti sono uguali in entrambe le direzioni (cioè se a.equals(b) e b.equals(c) sono entrambi veri, allora a.equals(c) deve essere vero).

toString: l'invocazione del metodo toString deve restituire una stringa che rappresenti l'oggetto in modo significativo. In particolare, la rappresentazione stringa dell'oggetto restituita dal metodo toString() deve essere una descrizione testuale che **includa le informazioni rilevanti** sull'oggetto, come ad esempio il suo **stato** interno o le sue **proprietà**.
Inoltre, la rappresentazione stringa dell'oggetto restituita dal metodo toString() deve essere in grado di **distinguere l'oggetto** da altri oggetti della stessa classe. In altre parole, due oggetti della stessa classe dovrebbero avere rappresentazioni stringa diverse se hanno stati interni diversi.
Il metodo toString() non ha alcuna limitazione specifica sulla lunghezza o sul formato della rappresentazione stringa restituita. Tuttavia, è generalmente buona pratica produrre una rappresentazione stringa concisa e facilmente leggibile.

hashCode : il metodo hashCode restituisce un valore hash dell'oggetto corrente. Il valore hash è un intero che viene utilizzato per l'indicizzazione in strutture dati come le tabelle hash. Il metodo hashCode è definito nella classe Object e può essere sovrascritto nelle classi derivate. Ogni volta che il metodo viene invocato su uno stesso oggetto, esso deve restituire in modo consistente lo stesso valore intero. Il metodo hashCode deve essere consistente anche con il metodo equals, il che significa che due oggetti uguali devono restituire lo stesso valore hash. Tuttavia, due oggetti con lo stesso valore hash non sono necessariamente uguali.
Inoltre, il metodo hashCode deve essere implementato in modo efficiente, in modo da non rallentare l'elaborazione delle strutture dati che lo utilizzano.

Il contratto d'uso di questi metodi può essere riassunto come segue:
equals : il metodo equals deve essere sovrascritto per definire il concetto di uguaglianza tra gli oggetti. Il metodo deve essere consistente, riflessivo, simmetrico e transitivo.

toString : il metodo toString deve restituire una rappresentazione in formato stringa dell'oggetto corrente. Il metodo può essere sovrascritto per fornire una rappresentazione più significativa dell'oggetto.
hashCode : il metodo hashCode deve restituire un valore hash dell'oggetto corrente. Il metodo deve essere consistente con il metodo equals e deve essere implementato in modo efficiente.

Descrivere le interfacce funzionali e le loro relazioni con i riferimenti a metodo.

In Java, un'interfaccia funzionale è un'interfaccia che definisce esattamente un metodo astratto. L'utilizzo principale delle interfacce funzionali è supportare la programmazione funzionale in Java, che si basa sulla manipolazione di funzioni e oggetti funzionali. Grazie alle interfacce funzionali, è possibile definire oggetti funzionali e passarli come parametri ad altre funzioni.

Le lambda expressions sono un costrutto introdotto in Java 8 per consentire la creazione di oggetti funzionali senza la necessità di creare una classe separata. Una lambda expression è essenzialmente una funzione anonima che può essere utilizzata come parametro di un'altra funzione o come corpo di un metodo.

Le lambda expressions sono composte da una lista di parametri separati da virgole, una freccia → e il corpo della funzione. Il corpo della funzione può essere una singola espressione o un blocco di istruzioni racchiuso tra parentesi graffe {} . Ad esempio, la seguente lambda expression calcola il doppio di un numero:

```
Function<Integer, Integer> square = x → x * x;
```

I riferimenti a metodo (method reference) sono un costrutto introdotto in Java 8 per consentire di utilizzare un metodo esistente come implementazione di una interfaccia funzionale. In altre parole, i riferimenti a metodo offrono un modo semplice e leggibile per passare un metodo come parametro a un'altra funzione o per utilizzarlo come corpo di una lambda expression.

```
class CalcoliMatematici {

        public static int calcolaQuadrato(int n) {

        return n * n;

        }

}

Function<Integer, Integer> squareRef = CalcoliMatematici::calcolaQuadrato;



int numero = 5;

System.out.println("Risultato con lambda: " + square.apply(numero));

System.out.println("Risultato con rif. a metodo: " + squareRef.apply(numero));
```

Le interfacce funzionali e i riferimenti a metodo sono strettamente correlati. I riferimenti a metodo sono utilizzati per fornire un'implementazione del metodo astratto definito nell'interfaccia funzionale, in modo che possa essere passato come parametro a un'altra funzione. Ad esempio, se si ha un'interfaccia funzionale che definisce un metodo con un parametro di tipo String e un valore di ritorno di tipo int, è possibile fornire un'implementazione di questo metodo utilizzando un riferimento a un metodo esistente che abbia la stessa firma. In questo modo, si può passare il riferimento al metodo come parametro a un'altra funzione che richiede un'interfaccia funzionale con la stessa firma.

Definire il concetto di signature di un metodo ed il ruolo giocato nel overloading e nel overriding

In programmazione, la firma (signature) di un metodo indica il nome del metodo e i tipi e l'ordine dei suoi parametri. La firma di un metodo **è univoca** all'interno della classe che lo definisce.

Nell'overloading dei metodi, due o più metodi della stessa classe possono avere lo stesso nome, ma devono avere firme diverse, cioè devono avere un diverso numero di parametri o tipi di parametri diversi. In questo modo, il compilatore può distinguere tra i metodi con lo stesso nome a seconda dei parametri che vengono passati al metodo quando viene chiamato.

Nell'overriding dei metodi, invece, una classe figlia può definire un metodo con lo stesso nome e gli stessi parametri della classe madre. In questo caso, la firma del metodo nella classe figlia deve essere identica a quella nella classe madre. In questo modo, quando si chiama il metodo sulla classe figlia, il metodo nella classe figlia viene eseguito invece del metodo nella classe madre.

In generale, la firma del metodo viene utilizzata per determinare quale metodo verrà chiamato in base ai parametri passati e al tipo di oggetto che chiama il metodo.

public void stampaNumero(int numero)

In questo caso, la signature del metodo è stampaNumero(int). Il nome del metodo è "stampaNumero", il tipo di parametro è "int" e non ci sono parametri aggiuntivi. Questa firma indica che il metodo prende un argomento di tipo intero e non restituisce nulla. Se avessimo definito un altro metodo con lo stesso nome "stampaNumero", ma con un diverso tipo di parametro come ad esempio "double", allora avremmo avuto una firma differente: stampaNumero(double). In questo modo, il compilatore può distinguere tra i due metodi in base alla firma, permettendo di effettuare il cosiddetto "overloading" del metodo.

Il tipo di ritorno di un metodo non fa parte della sua firma.

Descrivere il meccanismo di elaborazione dei flussi di dati basato su stream

In Java, lo stream è una view su una sequenza di elementi che supporta operazioni parallele e sequenziali di elaborazione dei dati. Lo stream consente di manipolare facilmente una grande quantità di dati utilizzando le operazioni di **filter, map e reduce**.

Il processo di elaborazione dei flussi di dati basato su stream inizia con la creazione di uno stream di elementi da una fonte di dati, ad esempio una lista, un array o un file. Successivamente, gli elementi dello stream possono essere manipolati utilizzando le varie operazioni disponibili.

L'elaborazione sequenziale dei flussi di dati inizia con l'applicazione di una o più operazioni di filtro al flusso di dati, utilizzando il metodo filter(). Questo metodo viene utilizzato per filtrare gli elementi di uno stream in base a un criterio specificato. Ad esempio, se avessimo uno stream di numeri interi e volessimo selezionare solo quelli pari, potremmo utilizzare il metodo filter() con la funzione lambda x: x%2 == 0, che restituisce True se x è pari e False altrimenti. Il risultato sarebbe uno stream contenente solo i numeri pari.

Successivamente, possiamo applicare le operazioni di mappatura map() e riduzione (reduce()) per manipolare gli elementi dello stream. L'operazione di mappatura viene utilizzata per trasformare gli elementi dello stream in altri elementi. Ad esempio, potremmo utilizzare la funzione lambda x: x*2 per raddoppiare ogni numero nello stream.

Infine, l'operazione di riduzione (reduce()) viene utilizzata per combinare gli elementi dello stream in un unico valore. La funzione di riduzione deve prendere due argomenti, solitamente chiamati "accumulator" e "current", che rappresentano rispettivamente il valore accumulato fino a quel momento e l'elemento corrente dello stream. Ad esempio, potremmo utilizzare la funzione lambda x, y: x+y per sommare tutti i numeri nello stream raddoppiati dalla mappatura precedente.

L'elaborazione parallela dei flussi di dati utilizza il concetto di parallelismo dei dati, che consiste nell'elaborare i dati in parallelo su più processori o core della CPU. In Java, l'elaborazione parallela di uno stream viene effettuata utilizzando il metodo parallelStream(), che restituisce uno stream parallelo. Gli stream paralleli consentono di suddividere il flusso di dati in più parti e di elaborare ogni parte in parallelo.