Maddalena La Rosa

Test

Java

Che differenza c'è tra classe e oggetto in Java?

- ☐ Una classe è una singola istanza di un oggetto
- ☑ La classe è il modello, mentre l'oggetto è l'istanza concreta basata su quel modello
- ☐ Una classe è un'istanza di un oggetto.

Qual è lo scopo principale del costruttore in Java?

- ☐ Permette di creare nuove variabili all'interno di una classe.
- ☐ Definisce le variabili di istanza di una classe.
- ☑ Inizializza gli oggetti della classe con valori predefiniti o personalizzati.
- ☐ Fornisce metodi per modificare le variabili di istanza di una classe.

Qual è lo scopo principale dell'ereditarietà in Java?

- ☐ Permette di creare più istanze di una classe.
- ☐ Consente di definire le variabili di istanza di una classe.
- ☑ Fornisce un modo per condividere attributi e comportamenti tra classi, quindi consente il riutilizzo del codice
- ☐ Definisce metodi per modificare le variabili di istanza di una classe.

Qual è uno degli scopi principali del polimorfismo in Java?

- ☐ Consentire agli oggetti di essere trattati come istanze di loro superclassi.
- ☐ Consentire la definizione di variabili di istanza di una classe.
- ☑ Fornire un modo per condividere attributi e comportamenti tra classi.
- ☐ Scrivere codice generico che lavora con superclassi o interfacce senza la necessità di conoscere il tipo specifico dell'oggetto al momento della compilazione.

Quale modificatore di visibilità consente di accedere direttamente sia dalla classe in questione che dalle classi figlie?

- ☐ public
- ☑ protected
- ☐ private
- ☐ shared

Spiega cosa sono gli array in Java e in quali casi è utile utilizzarli. Fai anche un esempio pratico.

In Java gli array sono una struttura dati fissa, cioè una volta decisa la sua dimensione essa non si può modificare. Inoltre ogni variabile appartenente all'array può essere solo dello stesso tipo dell'array (quindi un array di stringhe può solo contenere stringhe). È utile quando voglio creare una struttura con un numero fisso di elementi accomunati da caratteristiche comuni. Se ad esempio voglio creare nel mio programma un contenitore per

l'alfabeto inglese, sappiamo che raramente aggiungeremo altre lettere alle 26 già esistenti e potremmo sfruttare la struttura dell'array.

Cos'è un metodo static in Java? Quando è opportuno usarlo?
Un metodo **static** in Java è un metodo che definisco dentro la classe che posso richiamare senza dover creare un oggetto della classe stessa.

Posso utilizzare un metodo static quando non ho l'esigenza di creare un oggetto ma solo di usare quel metodo. Questo mi permette di risparmiare linee di codice inutili e anche memoria. Ad esempio quando un metodo mi torna un risultato matematico o anche un metodo di utilità che serve solo a stampare. Esempi: Math.sqrt(), Integer.parseInt().

Cos'è final e quando è opportuno utilizzarlo?
La parola chiave **final** si utilizza quando si crea una classe e non si vuole sia sovrascritta (con l'overriding) da una sottoclasse. Si utilizza quando vogliamo il compilatore non compili eventuali overriding ad esempio di metodi che per motivi di sicurezza non vogliamo siano sovrascritti.

Che differenza c'è tra ArrayList e un array semplice ([]) in Java?
Visto che l'array è una struttura fissa che contiene elementi che hanno lo stesso tipo, in Java si è sentita la necessità di creare delle strutture dati che invece possano essere dinamiche e che possano contenere più tipi di dati: l'arrayList è una di queste. Ogni cella di memoria rimanda ad un'altra cella tramite un puntatore, questo sistema permette di poter aggiungere o eliminare celle in qualsiasi punto dell'ArrayList, perché l'ordine viene gestito dai puntatori.

Spiega che differenza c'è tra una coda (Queue) e una pila (Stack).
Sia la queue che la stack sono due strutture dati appartenenti alle collection di Java, la loro differenza basilare è che gestiscono in maniera diversa i loro elementi. La queue o coda è una collection che gestisce gli elementi sulla base della politica FIRST IN FIRST OUT (FIFO) quindi, similmente alla gestione della coda alla posta, un elemento che sta prima viene gestito prima e così via in ordine "di arrivo". Invece la stack o pila segue la politica del LAST IN FIRST OUT (LIFO) vale a dire che se l'elemento è l'ultimo sarà il primo ad essere gestito, è una dinamica che utilizziamo quando nel browser clicchiamo su Indietro: ci troviamo sempre all'ultima pagina visitata.

Cos'è il concetto di incapsulamento in Java e perché è importante nella programmazione ad oggetti?
L'incapsulamento è uno dei paradigmi della programmazione ad oggetti, ci permette tramite tre modificatori (public, private, protected) di gestire la sicurezza del nostro programma, proteggendo i nostri dati grazie a vari livelli di protezione da dare agli attributi o ai metodi. In particolare:

- public rende l'attributo o il metodo accessibile ovunque, quindi anche al di fuori della classe in cui è stato creato.
- private rende l'attributo o il metodo accessibile solo all'interno della classe in cui è stato creato

- protected rende l'attributo o il metodo accessibile non solo all'interno della classe ma anche dalle sottoclassi

Se non utilizziamo nessuno dei tre modificatori, di default l'attributo o il metodo è accessibile all'interno del package.
L'incapsulamento è importante per gestire la sicurezza dei dati del nostro programma, ci permette di controllare chi accede o modifica un attributo o un metodo.

Che ruolo ha il modificatore private in una classe? Perché è utile usarlo insieme ai metodi getter e setter?
Dato che è altamente sconsigliato che gli attributi siano public per ragioni di sicurezza, quando creiamo gli attributi di una classe li creiamo private. In questo modo il programma non può modificare in altre parti del codice al di fuori della classe il contenuto degli attributi di quest'ultima, ma si può accedere agli attributi tramite i metodi adibiti a interagire con essi: i setter (i metodi per impostare il contenuto di un determinato attributo) e i getter (i metodi per "prendere" e poter utilizzare il contenuto di un attributo). La best practice è settare gli attributi come private, e i metodi come public.

Scrivi una classe Java che rappresenta un "Libro" con almeno 3 attributi e un metodo per stampare le informazioni del libro.

```
public class Libro {
 private String titolo;
 private int pagine;
 private String genere;
  public Libro(String titolo, int pagine, String genere) {
   this.titolo = titolo;
   this.pagine = pagine;
   this.genere = genere;
 }

 public String getTitolo() {
   return titolo;
 }

 public void setTitolo(String titolo) {
   this.titolo = titolo;
 }

 public int getPagine() {
   return pagine;
 }
```

```
 public void setPagine(int pagine) {
   this.pagine = pagine;
 }

 public String getGenere() {
   return genere;
 }

 public void setGenere(String genere) {
   this.genere = genere;
 }

 @Override
 public String toString() {
   return "Libro [titolo=" + titolo + ", pagine=" + pagine + ", genere=" +
genere + "]";
 }
}
```

Cosa si intende per overriding di un metodo? Quando e perché lo si usa?
L'overriding di un metodo fa parte del paradigma della programmazione di Java chiamato ereditarietà. Ci permette di poter utilizzare il nome di una funzione più volta, ma cambiando di volta in volta i parametri. In base ai parametri che inseriamo, possiamo cambiare l'output di quella funzione. Mi spiego con un esempio:
possiamo creare la funzione somma che accetta come parametri due interi e ritorna un intero:

```
function somma (int a, int b) {...}
```

possiamo creare un'altra funzione somma ma che accetta due float e ritorna un float

```
function somma (float a, float b) {...}
```

o ancora possiamo creare un'altra funzione somma che accetta più di due parametri int

```
function somma (int a, int b, int c) {...}
```

Riepilogando, l'overriding ci permette di creare più "versioni" di una funzione mantenendo lo stesso nome ma cambiando la firma. La firma è composta dal nome della funzione e i parametri che vuole in entrata. Esempio:

```
somma (int a, int b)
```

Spiega cosa fa il costrutto super e quando è utile utilizzarlo.

Il costruttore super() è utile quando stiamo creando una sottoclasse che estende una classe padre, e vogliamo che erediti i costruttori degli attributi della classe padre. In questo modo il programma fa ereditare alla classe figlia correttamente gli attributi della classe padre. E poi eventualmente si aggiungono i costruttori degli attributi in più.

In molti linguaggi di programmazione una stringa è trattata semplicemente come un array di caratteri. In Java, invece, le stringhe sono oggetti della classe `String`. Spiega cosa significa che una stringa è un oggetto in Java e quali vantaggi comporta.
Le stringhe appunto non vengono trattate semplicemente come un array di caratteri, ma come oggetti della classe String. Per dichiararle e inizializzarle basta usare la parola chiave String con la prima lettera maiuscola (che si utilizza per dare il nome alle classi) e dopo di ché il pacchetto base di Java rende disponibile tutta una serie di metodi utili quando si lavora con le stringhe.
.equals() per poter confrontare il contenuto delle stringhe (ricordiamo che invece il == confronta la loro posizione in memoria e non il loro contenuto)
.length che restituisce la lunghezza della stringa
.toLowerCase() o toUpperCase() per rendere le stringhe rispettivamente tutte in minuscolo o tutte in maiuscolo

Parlami di un argomento a piacere tra tutti quelli che abbiamo trattato, specificando in quali contesti è opportuno utilizzarlo e descrivi anche un esempio.

Un argomento a piacere che non è stato trattato nel resto dell'esame potrebbe essere l'interfaccia. L'interfaccia è uno strumento messo a disposizione da Java per colmare la caratteristica intrinseca di una sottoclasse che può essere figlia solo di una classe padre. Quindi quando alcune sottoclassi hanno altre caratteristiche che non sono date dal padre, ma sono in comune tra alcune, si utilizzano tramite la parola chiave implements una o più interfacce. Abbiamo fatto l'esempio della classe padre Veicolo che potrebbe avere tante classi figlie come Automobile, Camion, Camper, Moto, ecc. Ma esistono anche le automobili elettriche, le automobili ibride, i camper elettrici, o ibridi, ecc. Quindi una soluzione è creare anche le interfacce Ibrido e Elettrico che vengono implementate solo quando è necessario.

Esericizio 1 - Classi e Oggetti

https://github.com/larosamad/its-programmazione1/tree/main/test1_classioggetti

Crea una classe `Studente` con i seguenti attributi:

- `nome` (String)
- `cognome` (String)
- `annoNascita` (int)

Crea un costruttore per inizializzare questi attributi e un metodo `stampaScheda()` che stampi una frase come:

"Mario Rossi, nato nel 2004"

Poi, nel `main`, crea 2 oggetti `Studente` e chiama il metodo `stampaScheda()`.

Esericizio 2 - Ereditarietà e Override

https://github.com/larosamad/its-programmazione1/tree/main/test2_ereditarietaoverride

Crea una gerarchia di classi per rappresentare le persone che fanno parte di una scuola.

Consegna anche il diagramma UML che rappresenta lo schema delle classi.

1. Classe `Persona`

- Attributi:
  - `nome` (String)
  - `cognome` (String)
- Costruttore che inizializza nome e cognome
- Metodo `presentati()` che stampa:
  "Ciao, sono [nome] [cognome]"

2. Classe `Studente` (estende `Persona`)

- Attributo aggiuntivo: `matricola` (String)
- Costruttore che riceve anche la matricola
- Override del metodo `presentati()` per stampare:
  "Sono lo studente [nome] [cognome], matricola: [matricola]"

3. Classe `Professore` (estende `Persona`)

- Attributo aggiuntivo: `materia` (String)
- Costruttore che riceve anche la materia
- Override del metodo `presentati()` per stampare:
  "Sono il prof. [nome] [cognome], insegno [materia]"

🧪 Nel `main`

- Crea 3 oggetti:
  - 1 `Studente`
  - 1 `Professore`
  - 1 `Persona` generica
- Inseriscili in un array di tipo `Persona[]`
- Scorri l'array con un ciclo `for` e chiama `presentati()` su ciascuno

Esericizio 3 - Sistema di autenticazione utenti

https://github.com/larosamad/its-programmazione1/tree/main/test3_autentificazioniutente
Mi sono aiutata per alcune implementazioni dell'interfaccia come l'uso di throws, throw new, instanceof
Su cosa scrive dentro la classe AutenticazioneException
Questa riga è stata aggiunta per evitare un warning dell'IDE:

```
private static final long serialVersionUID = 1L;
```

Il programma gestisce diverse tipologie di utenti (Studenti, Professori, Utenti generici) e consente l'autenticazione solo agli utenti che implementano l'interfaccia Autenticabile. Se la password inserita non è corretta, viene sollevata un'eccezione personalizzata.

Consegna anche il diagramma UML che rappresenta lo schema delle classi.

1. Classe base `Utente`

- Attributi: `username`, `email`
- Costruttore
- Metodo `presentati()` che stampa:
  "Utente generico: [username], email: [email]"

---

2. Interfaccia `Autenticabile`

- Metodo: `autentica(String password)`
- Lancia `AutenticazioneException` se la password è errata

---

3. Classi derivate:

`Studente` estende `Utente` e implementa `Autenticabile`

- Attributo: `matricola`
- `autentica()`: accetta solo la password `"studente123"`
- `presentati()` stampa:
  "Studente [username], matricola: [matricola]"

`Professore` estende `Utente` e implementa `Autenticabile`

- Attributo: `materia`

- `autentica()`: accetta solo la password `"prof2024"`
- `presentati()` stampa:
  "Professore [username], insegna: [materia]"

`Segreteria` estende `Utente`

- Attributo: `ufficio`
- Non implementa: `Autenticabile`
- `presentati()` stampa:
  "Personale di segreteria [username], ufficio: [ufficio]"

---

4. Eccezione personalizzata `AutenticazioneException`

- Estende `Exception`
- Costruttore con messaggio

🧪 Nel `main`

- Crea una `ArrayList<Utente>` con:
  - 1 `Studente`
  - 1 `Professore`
  - 1 `Segretario` (non autenticabile)
- Per ogni utente nella lista:
  - Chiama `presentati()`
  - Gestisci eventuali eccezioni con `try-catch`
  - Se l'oggetto è Autenticabile, allora **invocare autentica()** e gestire l'eccezione se la password è errata