



# Domande orale

## ▼ Garbage Collector

Il garbage collector è un componente del sistema Java che si occupa della gestione della memoria dinamica. Il suo compito è quello di individuare e rimuovere dalla memoria gli oggetti che non sono più utilizzati dal programma, in modo da liberare spazio e prevenire l'esaurimento della memoria, ovviamente evitando ogni tipo di frammentazione dovuto dalla continue allocazioni e deallocazioni.

Il garbage collector funziona esaminando periodicamente l'heap, la porzione di memoria dove gli oggetti vengono allocati durante l'esecuzione del programma. Quando il garbage collector individua un oggetto che non è più raggiungibile dal programma, lo marca come candidato all'eliminazione e successivamente lo rimuove dalla memoria. In questo modo, il garbage collector mantiene l'heap pulito e prevenendo le perdite di memoria e gli errori di allocazione.

Il garbage collector di Java è un processo automatico e trasparente, che non richiede l'intervento diretto del programmatore per liberare la memoria. Tuttavia, la sua attività può influire sulle prestazioni dell'applicazione, poiché il processo di rilevamento e rimozione degli oggetti non utilizzati può richiedere risorse di sistema. Per questo motivo, è importante capire il funzionamento del garbage collector e ottimizzare il proprio codice per minimizzare il carico di lavoro del garbage collector e massimizzare le prestazioni dell'applicazione.

## ▼ Finalize() (metodo alternativo java come delete in C, invocato prima che viene deallocated)

Il metodo `finalize()` è un metodo speciale che viene chiamato dal garbage collector quando un oggetto non ha più riferimenti che lo referenziano e quindi è destinato ad essere eliminato dalla memoria.

Il metodo `finalize()` viene ereditato dalla classe `Object`, la classe base di tutte le classi Java, e può essere sovrascritto nelle classi derivate per eseguire delle azioni personalizzate prima che l'oggetto venga eliminato dalla memoria.

La sintassi del metodo `finalize()` è la seguente:

```
protected void finalize() throws Throwable {
    // codice per eseguire operazioni di pulizia o altre azioni personalizzate
}
```

Il metodo `finalize()` viene chiamato solo una volta per ogni oggetto e il suo comportamento è indefinito se viene chiamato direttamente dal programma.

## ▼ Programmazione generica

La programmazione generica è un approccio alla scrittura del codice che consente di creare algoritmi e strutture dati che possono essere utilizzati con diversi tipi di dati, senza dover scrivere codice specifico per ogni tipo di dato. In questo modo, è possibile scrivere codice più modulare, flessibile e riutilizzabile.

In Java, la programmazione generica viene implementata attraverso le Java Generics, un meccanismo introdotto a partire dalla versione 5 del linguaggio. Le Java Generics consentono di definire classi, interfacce e metodi che possono essere utilizzati con uno o più tipi di dati generici, indicati con il placeholder `<T>`.

Quando si utilizza una classe o un metodo generico, il tipo effettivo viene specificato al momento della creazione dell'istanza o dell'invocazione del metodo.

**Esempio di Tipo Generics** - `ArrayList<T> aList = new ArrayList<T>();`

#### ▼ Casting (implicito ed esplicito)

Il casting è un'operazione che consente di convertire il tipo di una variabile in un altro tipo. In Java, il casting può essere esplicito o implicito.

Il casting **esplicito** (o forzato) viene effettuato utilizzando l'operatore di casting "(tipo) variabile", mentre il casting **implicito** (o automatico) viene eseguito in modo automatico dal compilatore quando è possibile garantire la compatibilità dei tipi.

```
double varDouble = 3.14;
int varInt = (int) varDouble; // effettua il casting esplicito da double a int
System.out.println(varInt); // stampa 3
```

In questo esempio, il valore della variabile "varDouble" viene convertito in un valore intero tramite un casting esplicito, e il risultato viene assegnato alla variabile "varInt".

Il valore stampato a video sarà 3, poiché la parte decimale viene eliminata nella conversione da double a int.

È importante notare che il casting può comportare la perdita di precisione o la generazione di eccezioni, soprattutto quando si convertono tipi di dati incompatibili o quando si cerca di convertire valori al di fuori del range ammesso dal tipo di destinazione.

Per questo motivo, è importante utilizzare il casting con cautela e verificare sempre la compatibilità dei tipi prima di effettuare una conversione.

#### ▼ Downcasting e upcasting

In Java, il casting è il processo di conversione di un valore da un tipo di dato a un altro tipo di dato. Esistono due tipi di casting: upcasting e downcasting.

**Upcasting** è quando si converte un oggetto da un tipo derivato a un tipo di base. Questo è possibile in quanto il tipo derivato ha tutti i metodi e le proprietà del tipo di base, oltre a eventuali altri metodi e proprietà che potrebbe avere definito. Ad esempio:

```
public class Animale {
    public void mangia() {
        System.out.println("L'animale sta mangiando.");
    }
}

public class Cane extends Animale {
    public void abbaia() {
        System.out.println("Il cane sta abbaiano.");
    }
}

// upcasting: il cane viene convertito in un animale
Cane cane = new Cane();
Animale animale = (Animale) cane;

// ora è possibile chiamare il metodo "mangia" dell'oggetto "animale"
animale.mangia();
```

Il **downcasting**, invece, è quando si converte un oggetto da un tipo di base a un tipo derivato. Questo è possibile solo se l'oggetto in questione è effettivamente un'istanza del tipo derivato. In caso



contrario, si verifica una ClassCastException a runtime.

Ad esempio:

```
public class Animale {  
    public void mangia() {  
        System.out.println("L'animale sta mangiando.");  
    } }  
  
public class Cane extends Animale {  
    public void abbaia() {  
        System.out.println("Il cane sta abbaiano.");  
    } }  
  
// upcasting: il cane viene convertito in un animale  
Cane cane = new Cane();  
Animale animale = (Animale) cane;  
  
// downcasting: l'animale viene convertito di nuovo in un cane  
Cane caneDiNuovo = (Cane) animale;  
  
// ora è possibile chiamare il metodo "abbaia" dell'oggetto "caneDiNuovo"  
caneDiNuovo.abbaia();
```

È importante notare che il downcasting può essere pericoloso se non viene eseguito correttamente, poiché può portare a errori a runtime.

Pertanto, è importante verificare sempre che un oggetto sia effettivamente un'istanza del tipo derivato prima di eseguire il downcasting.

#### ▼ Static

La parola chiave "static" può essere utilizzata per indicare che un'entità, come una variabile, un metodo o una classe, appartiene alla classe stessa piuttosto che alle istanze della classe.

```
public class MiaClasse {  
    public static int numero = 42;  
  
    public void mioMetodo() {  
        // codice del metodo  
    }  
}  
  
// accedere alla variabile "numero" senza istanziare un oggetto della classe "MiaClasse"  
int var = MiaClasse.numero;
```

In generale, l'utilizzo di "static" può aiutare a organizzare e gestire meglio il codice, e a migliorare le prestazioni in determinati casi.

Tuttavia, è importante utilizzare "static" con attenzione e solo quando necessario, per evitare di creare codice complicato o difficile da mantenere.

#### ▼ Final

In Java, la parola chiave "final" può essere utilizzata per indicare che un'entità, come una variabile, un metodo o una classe, non può essere modificata o estesa una volta dichiarata.

Nel caso delle variabili, l'utilizzo di "final" indica che il valore della variabile non può essere modificato una volta assegnato. Ad esempio:

```
final int numero = 42;  
numero = 10; // genera un errore di compilazione, poiché "numero" è una variabile final
```



Nel caso dei metodi, l'utilizzo di "final" indica che il metodo non può essere sovrascritto dalle sottoclassi.

Questo significa che il comportamento del metodo sarà costante in tutte le sottoclassi, garantendo la stabilità del codice. Ad esempio:

```
public class MiaClasse {  
    public final void mioMetodo() {  
        // codice del metodo  
    }  
}  
  
public class MiaSottoclasse extends MiaClasse {  
    // genera un errore di compilazione, poiché "mioMetodo" è un metodo final e non può essere sovrascritto  
}
```

Nel caso delle classi, l'utilizzo di "final" indica che la classe non può essere **EREDITATO** da altre classi.

Questo significa che la classe è completa e non può essere modificata o specializzata. Ad esempio:

```
public final class MiaClasse {  
    // codice della classe  
}  
  
public class MiaSottoclasse extends MiaClasse {  
    // genera un errore di compilazione, poiché "MiaClasse" è una classe final e non può essere estesa  
}
```

In generale, l'utilizzo di "final" può aiutare a garantire la stabilità del codice e prevenire errori o problemi derivanti dalla modifica di variabili, metodi o classi.

Tuttavia, è importante utilizzare "final" con attenzione e solo quando necessario, per evitare di limitare la flessibilità del codice o rendere difficile la manutenzione o l'estensione del software.

#### ▼ classi interne (in una classe ne dichiari un'altra) come la istanzio? cast col padre nel main

Le classi interne in Java sono classi che sono definite all'interno di un'altra classe. Ci sono quattro tipi di classi interne in Java: classi interne non statiche, classi interne statiche, classi locali e classi anonime.

Per istanziare una classe interna non statica, è necessario innanzitutto creare un'istanza della classe esterna, e quindi utilizzare questa istanza per creare un'istanza della classe interna. Ad esempio:

```
public class ClasseEsterna {  
    public class ClasseInterna {  
        public void metodo() {  
            System.out.println("Sono il metodo della classe interna.");  
        }  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        // istanziamo la classe esterna  
        ClasseEsterna esterna = new ClasseEsterna();  
  
        // istanziamo la classe interna utilizzando l'istanza della classe esterna  
        ClasseEsterna.ClasseInterna interna = esterna.new ClasseInterna();  
  
        // chiamiamo il metodo della classe interna  
        interna.metodo();  
    }  
}
```



```
}
```

Per quanto riguarda il cast con la classe padre, supponiamo di avere una classe interna non statica come la seguente:

```
public class ClasseEsterna {  
    public class ClasseInterna extends Animale {  
        public void metodo() {  
            System.out.println("Sono il metodo della classe interna.");  
        } } }  
  
public class Animale {  
    public void mangia() {  
        System.out.println("L'animale sta mangiando.");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        // istanziamo la classe esterna  
        ClasseEsterna esterna = new ClasseEsterna();  
  
        // istanziamo la classe interna utilizzando l'istanza della classe esterna  
        ClasseEsterna.ClasseInterna interna = esterna.new ClasseInterna();  
  
        // upcasting: la classe interna viene convertita in un animale  
        Animale animale = (Animale) interna;  
  
        // ora è possibile chiamare il metodo "mangia" dell'oggetto "animale"  
        animale.mangia();  
    }  
}
```

In questo esempio, la classe interna estende la classe `Animale`, quindi è possibile eseguire un upcasting dell'oggetto della classe interna alla classe `Animale`.

In questo modo è possibile utilizzare l'oggetto della classe interna come se fosse un oggetto della classe `Animale`, ma con accesso anche ai metodi della classe interna.

#### ▼ Polimorfismo

Il polimorfismo è uno dei concetti fondamentali della programmazione orientata agli oggetti. Si riferisce alla capacità di un oggetto di assumere più di una forma.

In altre parole, un oggetto può essere visto e utilizzato come un oggetto di un tipo specifico, ma allo stesso tempo può essere visto e utilizzato come un oggetto di un tipo diverso.

Esso si basa sull'ereditarietà e sulla sostituzione: un oggetto di una classe derivata può essere assegnato a una variabile di una classe base, perché il tipo derivato è un tipo del tipo base.

In questo modo, il codice può essere scritto in modo più generico e riutilizzabile, poiché le classi derivate possono sostituire le classi base senza dover modificare il codice che le utilizza.

Il polimorfismo può essere implementato in Java attraverso l'uso di classi e metodi astratti, l'implementazione di interfacce e l'utilizzo di ereditarietà.

Ad esempio, una classe base può definire un metodo generico che viene implementato in modo diverso dalle classi derivate, consentendo loro di fornire un comportamento specifico ma mantenendo la stessa interfaccia.

#### ▼ Espressioni lambda (classi interne e interfacce funzionali)

Le espressioni lambda sono una caratteristica introdotta in Java 8 per semplificare la scrittura di codice che richiede l'implementazione di interfacce funzionali.

In pratica, una espressione lambda è una funzione anonima che può essere utilizzata in modo simile a un'implementazione di un'interfaccia funzionale. L'idea è quella di fornire un modo più conciso e leggibile per definire comportamenti che possono essere passati come argomenti a un metodo o assegnati a una variabile.

Le espressioni lambda sono costituite da una lista di argomenti separati da virgolette, una freccia (`->`) e un'espressione o un blocco di istruzioni. Ad esempio:

```
(int x, int y) -> x + y
```

Questa espressione lambda definisce una funzione che prende due argomenti di tipo `int` e restituisce la loro somma.

Le espressioni lambda sono utilizzate soprattutto per implementare interfacce funzionali, ovvero interfacce che hanno un solo metodo astratto. Ad esempio, l'interfaccia funzionale `Runnable` ha un solo metodo astratto `run()`, che può essere implementato con una espressione lambda:

```
Runnable r = () -> System.out.println("Hello, world!");
```

In questo esempio, la espressione lambda viene assegnata a una variabile di tipo `Runnable`, il che significa che può essere passata come argomento a un metodo o eseguita in un thread.

#### ▼ GUI (tipi di layout)

GUI, acronimo di Graphical User Interface, è una interfaccia utente grafica che permette all'utente di interagire con il software attraverso oggetti grafici come pulsanti, caselle di testo, finestre di dialogo e così via.

In Java, la creazione di GUI può essere realizzata attraverso la libreria Swing, che mette a disposizione numerosi componenti grafici e layout manager per organizzare i componenti sulla finestra.

I layout manager sono utilizzati per determinare la posizione e le dimensioni dei componenti all'interno del container.

In Java, i layout manager più comuni sono il `BorderLayout` e il `GridBagLayout`.

Il `BorderLayout` divide il container in cinque aree: nord, sud, est, ovest e centro, e posiziona i componenti in base alla loro posizione relativa.

Ad esempio, un pulsante posizionato al centro occuperebbe la maggior parte dello spazio disponibile, mentre un pulsante posizionato a sud occuperebbe solo lo spazio necessario per visualizzare il testo.

Il `GridBagLayout` organizza i componenti in una griglia, in cui ogni cella può contenere un componente. Tuttavia, a differenza di altri layout manager a griglia, il `GridBagLayout` consente di specificare la posizione e le dimensioni dei componenti in modo flessibile e preciso, utilizzando un oggetto  `GridBagConstraints` che specifica i vincoli di posizionamento.

Oltre a questi layout manager, Java fornisce anche il `FlowLayout`, che organizza i componenti in una riga o in una colonna, a seconda della direzione specificata, e il `BoxLayout`, che organizza i componenti in una riga o in una colonna, ma consente di specificare le dimensioni preferite e massime dei componenti.



#### ▼ Eccezioni, controllate e non (Superclasse Throwable)

In Java, un'eccezione è un'istanza di una classe che rappresenta un'evenienza anomala che si verifica durante l'esecuzione di un programma.

Le eccezioni possono essere di due tipi: controllate e non controllate.

Le eccezioni controllate sono quelle che il compilatore Java richiede di essere gestite o dichiarate dal codice che le utilizza.

Ad esempio, se un metodo può sollevare un'eccezione di tipo `IOException`, il chiamante del metodo deve gestire l'eccezione o dichiararla nella sua firma del metodo.

Le eccezioni non controllate, al contrario, sono eccezioni che possono essere sollevate in qualsiasi momento durante l'esecuzione del programma e non richiedono una gestione obbligatoria.

Ad esempio, `NullPointerException` viene sollevata quando si tenta di accedere a un oggetto che ha il valore `null`.

Tutte le eccezioni in Java derivano dalla classe `Throwable`, che ha due sottoclassi dirette: `Error` e `Exception`.

La classe `Error` rappresenta problemi irrecuperabili, come la memoria esaurita, e non dovrebbe essere gestita dal codice dell'applicazione.

La classe `Exception`, invece, rappresenta problemi che possono essere gestiti dal codice dell'applicazione.

In generale, è buona pratica gestire le eccezioni in modo appropriato nel codice, per evitare crash imprevisti e per fornire un feedback utile all'utente in caso di errori. La gestione delle eccezioni può includere la registrazione degli errori, il ripristino dello stato dell'applicazione e il fornire all'utente le informazioni necessarie per risolvere il problema.

#### ▼ Superclasse Object (toString, equals, hashmap)

`Object` è la superclasse di tutte le classi. Questo significa che tutte le classi estendono implicitamente `Object`

e che tutti gli oggetti in Java sono istanze di una classe che discende da `Object`. Questa classe fornisce un insieme di metodi di base che sono disponibili per tutte le classi in Java.

Uno dei metodi più utilizzati forniti dalla classe `Object` è `toString()`, che restituisce una rappresentazione in forma di stringa dell'oggetto.

Questo metodo è spesso sovrascritto dalle sottoclassi per restituire una rappresentazione più significativa dell'oggetto.

Un altro metodo importante fornito da `Object` è `equals()`, che viene utilizzato per confrontare due oggetti e determinare se sono uguali.

Questo metodo viene spesso sovrascritto dalle sottoclassi per fornire una definizione personalizzata di uguaglianza.

La classe `Object` fornisce anche il metodo `hashCode()`, che viene utilizzato in combinazione con la classe `HashMap` per la creazione di mappe hash.

Una mappa hash è una struttura dati che associa chiavi a valori, e il metodo `hashCode()` viene utilizzato per calcolare un valore numerico univoco per ogni oggetto che viene utilizzato come chiave nella mappa.

#### ▼ Stream (differenza tra for ENHANCED “con i due punti” e stream)

In Java, uno stream è una sequenza di elementi che può essere elaborata in modo efficiente e conciso.

Gli stream sono disponibili a partire da Java 8 e sono progettati per semplificare la manipolazione di collezioni di dati.

Una differenza fondamentale tra gli stream e i cicli `for` tradizionali, come il `for-each` loop, è che gli stream sono basati su un paradigma di elaborazione dichiarativo invece di uno imperativo.

Con un ciclo `for`, l'elaborazione dei dati è specificata esplicitamente nel codice, mentre con uno stream, l'elaborazione è definita attraverso una serie di operazioni da eseguire sugli elementi della sequenza.

Le operazioni sugli stream possono essere concatenate per formare una "pipeline" di elaborazione che può essere eseguita in un'unica passata sulla sequenza.

Ad esempio, si possono filtrare gli elementi della sequenza in base a determinati criteri, quindi eseguire una mappatura degli elementi in nuovi valori e infine ridurre i risultati a un singolo valore.

Inoltre, gli stream supportano l'elaborazione parallela, il che significa che possono essere eseguiti su più thread per sfruttare i processori multi-core.

Ciò può portare a un miglioramento delle prestazioni quando si elaborano grandi quantità di dati.

In confronto, i cicli `for` tradizionali, come il `for-each` loop, richiedono più codice per eseguire operazioni simili, non offrono la stessa flessibilità nella manipolazione dei dati e non supportano l'elaborazione parallela in modo nativo.

#### ▼ Come vengono rappresentati in memoria gli array bidimensionali?

Gli array bidimensionali sono rappresentati in memoria come una sequenza di elementi contigui, disposti in righe e colonne.

Ad esempio, un array bidimensionale di dimensioni  $n \times m$  è rappresentato come una sequenza di  $n$  righe, ognuna contenente  $m$  elementi.

Ogni elemento dell'array è identificato da una coppia di indici `[i][j]`, dove `i` indica la riga e `j` indica la colonna dell'elemento.

In memoria, gli elementi dell'array bidimensionale sono memorizzati in modo consecutivo.

Ciò significa che gli elementi di una riga vengono memorizzati in sequenza, seguiti dagli elementi della riga successiva e così via.

In particolare, gli elementi dell'array sono memorizzati in modo contiguo in una regione di memoria allocata dinamicamente, chiamata "heap".

La posizione di ciascun elemento viene calcolata in base alla posizione dell'elemento precedente nell'array, il tipo di dati e la dimensione dell'array.

#### ▼ Array tridimensionali

```
int[][][] array3d = new int[3][4][5];
```

In questo esempio, abbiamo creato un array tridimensionale `array3d` con una dimensione di  $3 \times 4 \times 5$ .

Ciò significa che abbiamo 3 matrici, ognuna delle quali ha 4 righe e 5 colonne.

Possiamo inizializzare gli elementi dell'array come segue:

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        for (int k = 0; k < 5; k++) {
            array3d[i][j][k] = i + j + k;
        }
    }
}
```

In questo esempio, abbiamo inizializzato ogni elemento dell'array con la somma degli indici delle tre dimensioni. Ad esempio, l'elemento `array3d[1][2][3]` avrà il valore 6 ( $1 + 2 + 3$ ).

L'accesso agli elementi dell'array tridimensionale può essere fatto utilizzando tre indici, come mostrato di seguito:

```
int value = array3d[1][2][3];
```

In questo esempio, stiamo accedendo all'elemento nella prima matrice, terza riga e quarta colonna, che avrà il valore 6 come indicato sopra.

## ▼ EREDITARIETÀ

L'ereditarietà in Java è un meccanismo attraverso il quale una classe può estendere un'altra classe esistente, ereditandone i metodi e le proprietà.

La classe che viene estesa viene chiamata "classe base" o "superclasse", mentre la classe che estende la classe base viene chiamata "classe derivata" o "sottoclasse".

In questo esempio, la classe `Studente` estende la classe `Persona`, ereditandone le proprietà e i metodi.

Possiamo quindi utilizzare tutte le proprietà e i metodi della classe `Persona` all'interno della classe `Studente`, senza doverli reimplementare, rendendo il codice più modulare e facile da mantenere (senza il rischio di errori di duplicazione del codice).

## ▼ Classe Astratta e Interfaccia (+ differenze)

Una **classe astratta** in Java è una classe che non può essere istanziata direttamente, ma viene utilizzata come classe base per creare altre classi derivate.

Una classe astratta può contenere metodi astratti, cioè metodi che non vengono implementati nella classe astratta ma devono essere implementati nelle classi derivate.

```
public abstract class Figura {
    public abstract double area(); }
```

**Un'interfaccia** in Java è simile a una classe astratta, ma contiene solo metodi astratti e costanti.

Le interfacce definiscono un contratto che una classe deve seguire se implementa quell'interfaccia. Le classi possono implementare più di una interfaccia, ma possono estendere solo una classe.

```
public interface Veicolo {
    void accelera(int velocita);
    void frena(); }
```

In questo esempio, l'interfaccia `Veicolo` definisce due metodi astratti `accelera()` e `frena()`.

Una classe che implementa l'interfaccia `Veicolo` deve fornire un'implementazione per questi metodi.

La **differenza principale** tra una classe astratta e un'interfaccia è che una classe astratta può avere variabili di istanza, costruttori e metodi non astratti, mentre un'interfaccia non può avere alcuno di questi.

Inoltre, una classe può estendere una sola classe astratta, ma può implementare più di una interfaccia.

## ▼ Stream Tokenizer

`StreamTokenizer` è una classe in Java che permette di analizzare un flusso di caratteri (come un file di testo) e di suddividerlo in token, cioè in parti più piccole che rappresentano parole, numeri, simboli, ecc.

Il `StreamTokenizer` legge i caratteri da un flusso di input (ad esempio un file) e li suddivide in token in base alle impostazioni specificate.

Il token può essere un numero (intero o floating point), una parola, un carattere, una stringa, un simbolo, ecc.

Ad esempio, il seguente codice legge un file di testo, conta il numero di parole e stampa il risultato:

```
import java.io.*;

public class ContaParole {
    public static void main(String[] args) {
        try {
            FileReader fileReader = new FileReader("testo.txt");
            StreamTokenizer tokenizer = new StreamTokenizer(fileReader);
            int count = 0;
            while (tokenizer.nextToken() != StreamTokenizer.TT_EOF) {
                if (tokenizer.ttype == StreamTokenizer.TT_WORD) {
                    count++;
                }
            }
            System.out.println("Numero di parole: " + count);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In questo esempio, `FileReader` viene utilizzato per leggere un file di testo e `StreamTokenizer` viene utilizzato per suddividere il file in token.

La variabile `count` viene incrementata ogni volta che viene trovata una parola (`tokenizer.ttype == StreamTokenizer.TT_WORD`).

Alla fine viene stampato il numero di parole trovate nel file.