1. What is encapsulation?

Encapsulation is one of the four fundamental Object-Oriented Programming (OOP) principles. It refers to the concept of wrapping the data (variables) and the code (methods) that operate on the data into a single unit, usually a class. It also involves restricting direct access to some of an object's components, which is a means of preventing unintended interference and misuse of the data.

In Java, encapsulation is achieved using:

Private variables: to restrict access.

Public getters and setters: to provide controlled access to the variables

Benefits:

Improves code maintainability and flexibility.

Increases security of data.

Allows validation logic to be added in setter methods.

Example:

```
public class Student {

  private String name;




  public String getName() {

     return name;

  }
```

```
  public void setName(String name) {
      if(name != null && !name.isEmpty()) {
          this.name = name;
      }
  }
}
```

2. How are ArrayLists different from arrays?

Feature Array ArrayList

Size Fixed (defined at creation) Dynamic (can grow/shrink)

Type Can hold both primitive and objects Only holds objects

Performance Faster (no overhead) Slower due to resizing, boxing

Syntax Simple Requires use of methods

Flexibility Less flexible More flexible

Example:

```
int[] arr = new int[5]; // Array

ArrayList<Integer> list = new ArrayList<>(); // ArrayList
```

ArrayList is part of the Java Collections Framework and offers built-in methods like .add(), .remove(), .contains(), etc., which makes it

easier to work with compared to arrays.

3. How to sort an ArrayList?

You can sort an ArrayList using the Collections.sort() method. This method sorts the elements of the list in natural order (ascending for numbers, alphabetical for strings) or you can use a custom Comparator for custom sorting.

Example:

```
ArrayList<Integer> list = new ArrayList<>();

list.add(3);

list.add(1);

list.add(2);

Collections.sort(list); // Sorts in ascending order
```

Custom Sorting:

```
Collections.sort(list, Collections.reverseOrder()); // Descending order
```

For custom objects:

```
Collections.sort(studentList, (s1, s2) -> s1.getName().compareTo(s2.getName()));
```

4. What is constructor overloading?

Constructor overloading in Java means having multiple constructors in a class with different parameter lists. This allows the creation of objects in different ways, depending on the parameters passed.

Example:

```
public class Book {
```

```
  String title;
  int pages;
Book() {
    title = "Unknown";
    pages = 0;
  }
Book(String title) {
    this.title = title;
    this.pages = 100;
  }
Book(String title, int pages) {
    this.title = title;
    this.pages = pages;
  }
}
```

Benefits:

Increases flexibility.

Supports different ways of initializing an object.

Promotes code reuse.

5. How does garbage collection work in Java?

Garbage Collection (GC) is the process by which Java automatically deallocates memory by removing objects that are no longer in use, thereby preventing memory leaks.

Java uses an automatic garbage collector, which runs in the background and:

1. Identifies objects that are no longer reachable.

2. Reclaims the memory occupied by these objects.

How it works:

An object becomes eligible for GC when no live thread can access it.

Java uses algorithms like Mark and Sweep, Generational GC, etc.

Example:

```
Book b = new Book();

b = null; // Now eligible for garbage collection

System.gc(); // Suggests JVM to run GC
```

6. Why do we use getters and setters?

Getters and setters are methods used to read and write private variables from outside the class, providing controlled access to the fields.

Benefits:

Helps implement encapsulation.

Allows validation logic before setting values.

Makes the code easier to maintain and refactor.

Supports read-only or write-only fields.

Example:

```
private int age;

public int getAge() {

 return age;

}

public void setAge(int age) {

  if (age > 0) {

      this.age = age;

  }

}
```

7. What is a static variable?

A static variable is a variable that belongs to the class rather than to any specific object of that class. It is shared among all instances of that class.

Characteristics:

Initialized only once.

Shared by all instances.

Can be accessed using the class name.

Example:

```
public class Student {

  static int count = 0;



  Student() {

      count++;

  }

}
```

In this example, every time a Student object is created, the static variable count is incremented.

8. What is the use of final keyword?

The final keyword in Java is used to indicate that something cannot be changed. It can be applied to variables, methods, and classes.

Usage:

Final variable: value cannot be changed once assigned.

Final method: cannot be overridden.

Final class: cannot be inherited.

Examples:

final int x = 10;

x = 20; // Error: cannot change final variable

final class Shape { }

// class Circle extends Shape { } // Error: cannot extend final class

9. Difference between compile-time and runtime errors?

Feature Compile-Time Error Runtime Error

When it occurs During compilation During program execution

Detected by Compiler JVM

Examples Syntax errors, type mismatch NullPointerException, divide by zero

Fix Before running the program Requires debugging and exception handling

Example of Compile-Time Error:

int x = "hello"; // Type mismatch

Example of Runtime Error:

```
String s = null;

System.out.println(s.length()); // NullPointerException
```

---

10. What are access modifiers?

Access modifiers in Java define the scope of accessibility of classes, variables, methods, and constructors. Java has four types of access modifiers:

Modifier	Class	Package	Subclass	World
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No

default Yes Yes No No

private Yes No No No

Examples:

public int a; // Accessible fr

om anywhere

private int b; // Accessible only within the class

protected int c; // Accessible within package and subclasses

int d; // Default access, within same package