**Java Technical**

**1.what is encapsulation explain?**

Encapsulation is one of the four fundamental Object-Oriented Programming (OOP) concepts, the others being inheritance, polymorphism, and abstraction. It refers to the bundling of data (attributes or properties) and the methods (functions or procedures) that operate on the data into a single unit, often called a class in object-oriented programming languages.

The main idea behind encapsulation is to hide the internal details of an object and only expose what is necessary for the outside world to interact with it. This helps in achieving two key objectives:

1. **Data Hiding:** Encapsulation allows the internal details or state of an object to be hidden from the outside. Only the necessary features, such as methods or properties, are exposed. This helps prevent unintended interference and modification of an object's internal state.
2. **Abstraction:** Encapsulation also involves abstracting the complex internal workings of an object, providing a simplified and clear interface for interacting with the object. Users of the class don't need to know how the methods are implemented; they only need to know what the methods do and how to use them.
3. package javaprac;
4. public class EmployeeDetails {
5. private int emp\_id;
6. private String emp\_name;
7. private double emp\_salary;
8. private String emp\_mail\_id;
10. public int getEmpId()
11. {
12. return this.emp\_id;
13. }
14. public String getEmpName()
15. {
16. return this.emp\_name;
17. }
18. public double getEmpSalary()
19. {
20. return this.emp\_salary;
21. }
22. public String getEmpMail()
23. {
24. return this.emp\_mail\_id;
25. }
26. public void setEmpId(int empid)
27. {
28. emp\_id = empid;
29. }
30. public void setEmpName(String empname)
31. {
32. emp\_name = empname;
33. }
34. public void setEmpSalary(double empsalary)
35. {
36. emp\_salary = empsalary;
37. }
38. public void setEmpmail(String empmail)
39. {
40. emp\_mail\_id = empmail;
41. }
42. }
43. public class EncapsulationTest {
44. public static void main(String[] args) {
45. // **TODO** Auto-generated method stub
46. EmployeeDetails e1 = new EmployeeDetails();
47. e1.setEmpName("Samala Ritesh dhyan");
48. e1.setEmpId(1);
49. e1.setEmpmail("riteshdhyan2696@gmail.com");
50. e1.setEmpSalary(50000);
51. System.***out***.println("Employee 1 details : ");
52. System.***out***.println(e1.getEmpId()+" "+e1.getEmpName()+" "+e1.getEmpSalary()+" "+e1.getEmpMail());
53. System.***out***.println("----------------------------------------------------------------------------");
54. EmployeeDetails e2 = new EmployeeDetails();
55. e2.setEmpName("Nandhyala karthik Reddy");
56. e2.setEmpId(2);
57. e2.setEmpmail("nandhyalakarthikreddy@gmail.com");
58. e2.setEmpSalary(90000);
59. System.***out***.println("Employee 2 details : ");
60. System.***out***.println(e2.getEmpId()+" "+e2.getEmpName()+" "+e2.getEmpSalary()+" "+e2.getEmpMail());
61. }
62. }

**2.Explain about serialization in java?**

Serialization in Java is a process of converting the state of an object into a byte stream, which can be easily persisted or transmitted over a network. The reverse process, converting a byte stream back into an object, is called deserialization. Serialization is commonly used for data storage, network communication, and object persistence.

Java provides a mechanism for serialization through the **java.io.Serializable** interface. When a class implements this interface, it indicates that its instances can be serialized. The Java Virtual Machine (JVM) provides default serialization and deserialization mechanisms, but developers can also customize this process by implementing specific methods in the serializable class.

package javaprac;

import java.io.\*;

// A serializable class

class Person implements Serializable {

private static final long ***serialVersionUID*** = 1L; // for version control

private String name;

private int age;

// Constructor

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Getter methods (optional)

public String getName() {

return name;

}

public int getAge() {

return age;

}

}

public class SerializationExample {

public static void main(String[] args) {

// Create an instance of the serializable class

Person person = new Person("John Doe", 30);

// Serialization

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("D:\\programs\\person.ser"))) {

oos.writeObject(person);

System.***out***.println("Serialization: Object has been serialized and saved to person.ser");

} catch (IOException e) {

e.printStackTrace();

}

// Deserialization

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("D:\\programs\\person.ser"))) {

Person deserializedPerson = (Person) ois.readObject();

System.***out***.println("Deserialization: Object has been deserialized");

System.***out***.println("Name: " + deserializedPerson.getName());

System.***out***.println("Age: " + deserializedPerson.getAge());

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

In this example:

1. The **Person** class implements the **Serializable** interface.
2. An instance of the **Person** class (**person**) is created.
3. Serialization: The object is serialized and saved to a file named "person.ser" using **ObjectOutputStream**.
4. Deserialization: The object is read back from the file using **ObjectInputStream**, and its values are printed.

**3.Explain about coupling?**

Coupling, in the context of software design and architecture, refers to the degree of interdependence or connection between different components or modules within a system. It measures how much one module knows about or relies on the internal details or implementation of another. Low coupling is generally considered a desirable characteristic in software design, as it leads to more modular, maintainable, and flexible code.

There are several types of coupling, indicating different levels of interdependence between modules:

1. **No Coupling (Independent):** Modules are entirely independent of each other. A change in one module does not affect any other module. This is an ideal scenario for maintainability and flexibility.
2. **Data Coupling:** Modules are connected by passing data between them. They communicate by sharing only necessary information, and they are not concerned with each other's internal workings.
3. **Control Coupling:** Modules share control information. One module influences the behavior of another by specifying what it should do.

**Real time example**

1. **No Coupling (Independent):**

Imagine the e-commerce application has a module responsible for handling user authentication. This module is entirely independent of the module responsible for processing orders. The order processing module doesn't need to know any details about how authentication is done, and changes to the authentication module don't affect order processing.

1. **Data Coupling:**

Now, let's say there is a module responsible for calculating shipping costs. It communicates with the order processing module by passing necessary order details (like the destination address, package weight, etc.). The order processing module and the shipping cost calculation module are only connected through the data they exchange.

1. **Control Coupling:**

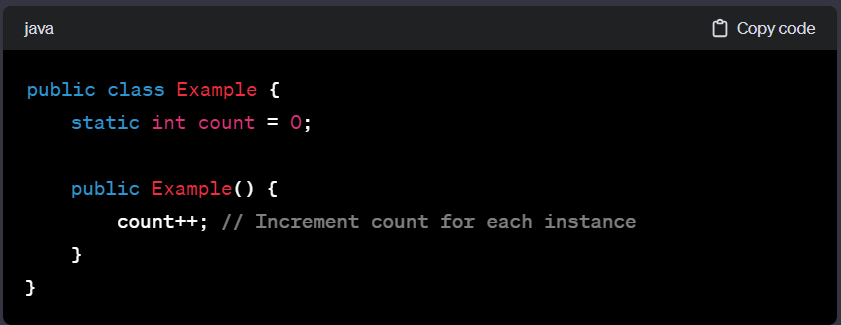
Suppose there's a module responsible for applying discounts to orders based on certain criteria. This module communicates with the order processing module, not just to get data, but also to influence the order processing logic. For example, it might instruct the order processing module to apply a specific discount based on a customer's loyalty status.

**4.explain about static keyword in java?**

In Java, the static keyword is used to declare members (fields, methods, and nested classes) that belong to the class itself, rather than to instances of the class. When a member is declared as static, it means that there is only one copy of that member shared by all instances of the class, and it can be accessed using the class name rather than an instance of the class.

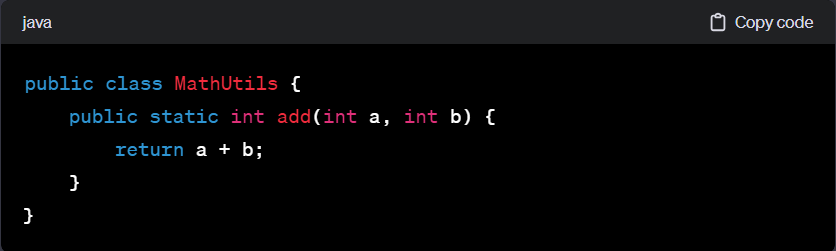
**Static Variables (Class Variables):**

* A static variable is shared by all instances of a class.
* It is declared using the **static** keyword and is often referred to as a class variable.
* Example:



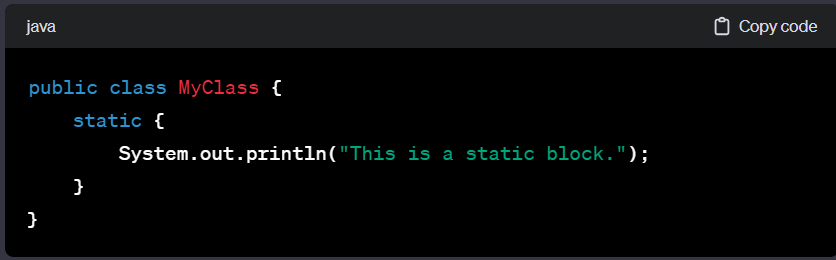
**Static Methods:**

* A static method belongs to the class rather than to any specific instance.
* It can be called using the class name, without creating an instance of the class.
* Example:



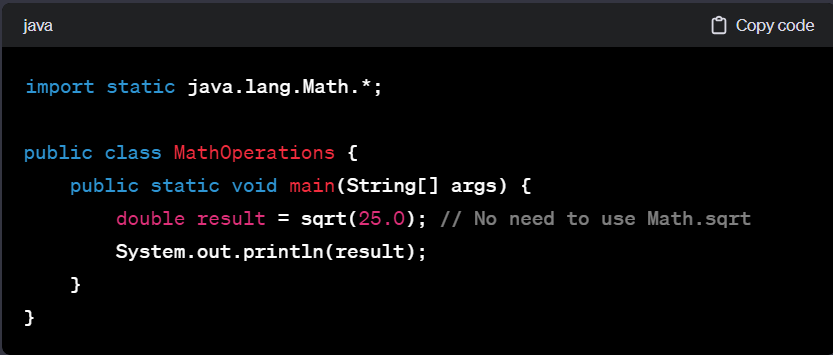
**Static Blocks:**

* A static block is used to initialize static variables or perform any one-time actions when the class is loaded.
* It is executed only once when the class is loaded into memory.
* Example:



**Static Import:**

* The **import static** statement allows you to access static members of a class without qualifying them with the class name.
* Example:



**5.finally vs final**

|  |  |
| --- | --- |
| **finally** | **Final** |
| 1.It is a block of statements. | 1.It is a keyword. |
| **2.**It is used in exception handling. Any statement which needs to be executed irrespective of whether an exception has occurred or not should be in the finally block. | **2.**It is applied for classes, variables and methods.   * Final class cannot be inherited. * Final variable cannot be updated. * Final method cannot be overrided. |
|  |  |

**6.explain about polymorphism with a real time example.**

Polymorphism, in the context of object-oriented programming, refers to the ability of a class to take on multiple forms. There are two types of polymorphism in Java: compile-time (method overloading) and runtime (method overriding). I'll provide an example of runtime polymorphism, which involves method overriding.

**Example: Animal Hierarchy**

Consider an example of an animal hierarchy where you have a base class **Animal** and two subclasses **Dog** and **Cat**. Each of these subclasses can make a distinct sound.

// Base class

class Animal {

public void makeSound() {

System.out.println("Some generic sound");

}

}

// Subclass 1

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Woof! Woof!");

}

public void fetch() {

System.out.println("Fetching the ball");

}

}

// Subclass 2

class Cat extends Animal {

@Override

public void makeSound() {

System.out.println("Meow!");

}

public void scratch() {

System.out.println("Scratching furniture");

}

}

public class PolymorphismExample {

public static void main(String[] args) {

Animal myDog = new Dog();

Animal myCat = new Cat();

// Polymorphic behavior

myDog.makeSound(); // Output: Woof! Woof!

myCat.makeSound(); // Output: Meow!

// Uncommenting the lines below would result in a compilation error

// myDog.fetch(); // Error: fetch() is not a method of the Animal class

// myCat.scratch(); // Error: scratch() is not a method of the Animal class

}

}

**7.exception types?**

In Java, exceptions are categorized into two main types: checked exceptions and unchecked exceptions.

1. **Checked Exceptions:**
   * These are exceptions that are checked at compile-time.
   * Any exception that extends **java.lang.Exception** but not **java.lang.RuntimeException** is a checked exception.
   * Checked exceptions must be either caught using a try-catch block or declared using the **throws** clause in the method signature.
   * Examples of checked exceptions include:
     + **IOException**
     + **ClassNotFoundException**
     + **SQLException**
     + **FileNotFoundException**
2. **Unchecked Exceptions (Runtime Exceptions):**
   * These are exceptions that are not checked at compile-time.
   * Any exception that extends **java.lang.RuntimeException** is an unchecked exception.
   * Unchecked exceptions do not need to be caught or declared, and the compiler does not enforce handling them.
   * Examples of unchecked exceptions include:
     + **ArithmeticException**
     + **NullPointerException**
     + **ArrayIndexOutOfBoundsException**

**8.**