OBJECT ORIENTED PROGRAMMING BASICS

INTERVIEW QUESTIONS

1 . **What is a class?**

In Java, a class is a blueprint or template for creating objects. It defines the structure and behavior of objects that can be created based on it. A class contains data members (fields) and methods that operate on those data members. The data members represent the attributes or properties of the objects, while the methods represent the actions or behaviors that the objects can perform.

2 . **What is an object?**

An object is an instance of a class in Java. It's a concrete, real-world entity that is created based on the blueprint provided by a class. Objects have their own unique state and behavior.

```
// Define a simple

class Person {

String name;    int

age;

}

public class ObjectExample {

public static void main(String[]

args) {

      // Create an object of the Person class

      Person person1 = new Person();


     // Set the state of the object

person1.name = "Alice";

 person1.age = 30;


      // Access and display the object's state
```

```
        System.out.println("Name: " + person1.name);

        System.out.println("Age: " + person1.age);

    }

}
```

3 . What is state of an object?

The state of an object refers to the values of its attributes or data members. It represents the current condition or characteristics of the object. For example, if you have a `Car` class, the state of a `Car` object would include attributes like its color, make, model, and current speed.

```
class Car {

    String color;

    String make;

    String model;

    int currentSpeed;


    // Constructor to initialize the Car object

    public Car(String color, String make, String model) {

        this.color = color;

        this.make = make;

        this.model = model;

        this.currentSpeed = 0; // Initial speed is 0

    }

}
```

4 . What is behavior of an object?

The behavior of an object refers to the actions or operations that the object can perform. These behaviors are defined by the methods of the class from which the object is created. Using the `Car` class example again, behaviors of a `Car` object might include methods like "start," "accelerate," "brake," and "turn." These methods define what the `Car` object can do.

```
class Car {
  String color;
  String make;
  String model;
  int currentSpeed;

  // Constructor and state initialization

  // Method to start the car
  public void start() {
    System.out.println("Starting the car.");
  }

  // Method to accelerate the car
  public void accelerate() {
    currentSpeed += 10;
    System.out.println("Accelerating. Current speed: " + currentSpeed + " mph");
  }

  // Method to brake the car
  public void brake() {
    currentSpeed -= 5;
    System.out.println("Braking. Current speed: " + currentSpeed + " mph");
  }

  // Method to turn the car
  public void turn(String direction) {
    System.out.println("Turning " + direction);
  }
}
```

5. What is the super class of every class in Java?

The superclass of every class in Java is the`Object` class. All classes implicitly or explicitly inherit from the `Object` class.

6. Explain about toString method ?

The `toString` method in Java is a method defined in the `Object` class, which can be overridden by other classes. It is used to return a string representation of an object. By default, the `toString` method returns a string that consists of the class name, an "@" symbol, and the object's hash code. Developers often override this method to provide a more meaningful and human-readable string representation of their objects.

7. What is the use of equals method in Java?
The `equals` method in Java is used to compare the content or value equality of two objects. It allows you to determine whether two objects are equal based on their attributes or data rather than their memory references. For example, consider a `Person` class with name and age attributes. Overriding the `equals` method allows us to compare two `Person` objects based on their name and age attributes for content equality.

```
class Person {
  String name;
  int age;

  // Constructor and state initialization

  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true; // Same object reference
    }
    if (obj == null || getClass() != obj.getClass()) {
      return false; // Different classes or null
    }
    Person person = (Person) obj; // Cast to the correct type
    return age == person.age && name.equals(person.name); // Compare attributes
  }
}
```

8 . What are the important things to consider when implementing equals method?

When implementing the `equals` method in Java, consider the following important aspects:

Reflexivity: Ensure that an object is equal to itself; that is, `x.equals(x)` should return `true`.

Symmetry: If `x.equals(y)` returns `true`, then `y.equals(x)` should also return `true`.

Transitivity: If `x.equals(y)` and `y.equals(z)` both return `true`, then `x.equals(z)` should return `true`.

Consistency: The result of `equals` should not change if the objects remain unchanged. In other words, multiple invocations of `equals` should return the same result as long as the objects have not been modified.

Handling `null`: Implement proper handling of null references, such as returning `false` if either object being compared is null.

Use `instanceof`: Use the `instanceof` operator to check if the object being compared is of the correct type before performing any attribute-based comparisons.

Override `hashCode`: If you override `equals`, also override the `hashCode` method to ensure that equal objects produce the same hash code.

Consistent with Business Logic: Ensure that the `equals` method reflects the relevant business logic for equality in your application domain.

9 . What is the Hashcode method used for in Java?

The `hashCode` method in Java is used to generate a unique integer value (hash code) for an object. It's primarily used for optimizing the performance of data structures and algorithms that rely on the efficient storage and retrieval of objects.

Here's the main purpose of the `hashCode` method:

Hash-Based Data Structures: Many data structures in Java, such as `HashMap`, `HashSet`, and `Hashtable`, use hash codes to store and retrieve objects quickly. When you insert an object into one of these collections, the `hashCode` method is called to determine the object's storage location. Later, when you want to find or remove an object, its `hashCode` is used to quickly identify the potential location, making lookups more efficient.

Ensuring Proper Functioning: The `hashCode` method is also used in conjunction with the `equals` method. When you override `equals`, you should also override `hashCode` to ensure that equal objects produce the same hash code. This is essential for consistency in hash-based data structures.

Here's a simplified example of how the `hashCode` method is used in a `HashMap`:

```
class Person {
    String name;
    int age;


    // Constructor and other methods


    @Override
    public int hashCode() {
        int prime = 31;
        int result = 1;
        result = prime * result + age;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }
}
```

In this example, the `hashCode` method combines the hash codes of the `name` and `age` attributes to create a unique hash code for each `Person` object. This ensures efficient storage and retrieval of `Person` objects in a `HashMap` while maintaining consistency with the `equals` method.

32 . Explain inheritance with examples .

Inheritance is one of the fundamental concepts in object-oriented programming (OOP) that allows a class (subclass or derived class) to inherit properties and behaviors (fields and methods) from another class (superclass or base class). This promotes code reuse and allows you to create a hierarchy of classes with shared attributes and behaviors.

Let's explain inheritance with an example in Java:

```
// Superclass or Base class
class Animal {
  String name;

  public Animal(String name) {
    this.name = name;
  }

  public void speak() {
    System.out.println(name + " makes a sound.");
  }
}

// Subclass or Derived class
class Dog extends Animal {
  public Dog(String name) {
    super(name); // Call the constructor of the superclass
  }

  @Override
  public void speak() {
    System.out.println(name + " barks."); // Overrides the speak method
  }

  public void wagTail() {
    System.out.println(name + " wags its tail.");
  }
}

public class InheritanceExample {
  public static void main(String[] args) {
    // Create an instance of the Dog class
    Dog myDog = new Dog("Buddy");

    // Access properties and methods inherited from Animal
    System.out.println("Name: " + myDog.name);
    myDog.speak(); // Calls the overridden speak method
```

```
        // Access methods specific to Dog
        myDog.wagTail();
    }
}
```

In this example:

- We have a `Animal` superclass with a `name` field and a `speak` method that prints a generic animal sound.
- We create a `Dog` subclass that extends `Animal`. The `Dog` class inherits the `name` field from `Animal` and overrides the `speak` method to provide a specific behavior for dogs.

In the `InheritanceExample` class, we create an instance of `Dog` called `myDog`. It can access both the `name` field and the `speak` method inherited from `Animal`, as well as the `wagTail` method specific to `Dog`.

33 . What is method overloading?

Method overloading is a feature in Java that allows you to define multiple methods in the same class with the same name but different parameter lists. In other words, you can have multiple methods with the same name in a class as long as they have a different number or type of parameters.

Key points about method overloading:

- Methods must have the same name but different parameter lists (number or types of parameters).
- Return types may or may not be the same.
- Overloaded methods are differentiated based on the number and types of their parameters.
- Method overloading is resolved at compile-time (static polymorphism).

Example of method overloading:

```
class Calculator {

    int add(int a, int b) {

        return a + b;

    }
    double add(double a, double b) {

        return a + b;

    }



    int add(int a, int b, int c) {

        return a + b + c;

    }

}
```

34 . What is method overriding?

Method overriding is a feature in Java that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. When a subclass overrides a method, it provides its own implementation of that method, which is used when the method is called on an instance of the subclass.

Key points about method overriding:

- The method in the subclass must have the same name, return type, and parameters (or a subtype of the parameter types) as the method in the superclass.
- Method overriding is used to provide a specialized behavior in the subclass.
- It is a fundamental concept of runtime polymorphism (dynamic method dispatch).
- The `@Override` annotation is often used to indicate that a method is intended to override a superclass method.

Example of method overriding:

```
class Animal {

    void makeSound() {

        System.out.println("Animal makes a sound.");

    }
}



class Dog extends Animal {

    @Override

    void makeSound() {

        System.out.println("Dog barks.");

    }

}
```

35 . Can super class reference variable can hold an object of sub class?
Yes, a superclass reference variable can hold an object of a subclass in Java, allowing you to access methods and fields specific to the subclass.

36 . Is multiple inheritance allowed in Java?
No, Java does not support multiple inheritance through class inheritance. However, it allows multiple inheritance through interfaces.

37 . What is an interface?
An interface in Java is a blueprint for a class that defines a set of abstract methods. It provides a contract for classes to implement specific behaviors.

38 . How do you define an interface?
To define an interface in Java, use the `interface` keyword followed by its name and a list of abstract method declarations. For example: `interface Drawable { void draw(); }`

39 . How do you implement an interface?
To implement an interface in Java, a class uses the `implements` keyword, followed by the interface name. The class must provide concrete implementations for all the methods declared in the interface. For example: `class Circle implements Drawable { /* Implement draw method */ }`

40 . Can you explain a few tricky things about interfaces?
Interfaces in Java can't contain instance variables; they are implicitly `public`, `static`, and `final`. Default methods allow backward compatibility when adding new methods to interfaces. Interfaces support multiple inheritance through implementing multiple interfaces. Interfaces define a contract for classes to implement specific behaviors. They are essential for achieving polymorphism and abstraction in Java.

41 . Can you extend an interface?

No, you cannot extend an interface using the `extends` keyword. Instead, you use the `implements` keyword to declare that a class is implementing an interface.

42 . Can a class extend multiple interfaces?
Yes, a class can implement multiple interfaces in Java, allowing it to inherit and provide implementations for multiple sets of behaviors.

43 . What is an abstract class?
An abstract class in Java is a class that cannot be instantiated and may contain abstract methods (methods without a body). It serves as a blueprint for other classes and can also contain concrete methods. It is defined using the `abstract` keyword.

44 . When do you use an abstract class?
You use an abstract class when you want to create a common base class for other classes, but you don't want instances of the abstract class itself. Abstract classes are useful when you have some methods that should be common to all derived classes, but they may also have abstract methods that must be implemented by derived classes, ensuring consistency in the hierarchy.

45 . How do you define an abstract method?
To define an abstract method in Java, you declare a method in an abstract class or interface without providing an implementation. It's denoted using the `abstract` keyword and lacks a method body. Subclasses or implementing classes are required to provide concrete implementations.

46 . Compare abstract class vs interface?
Abstract classes can have both abstract and concrete methods, support constructors, and can have instance variables. Interfaces can only have abstract methods (before Java 8), support multiple inheritance, and cannot have instance variables. Abstract classes represent an is-a relationship, while interfaces represent a can-do relationship.

47 . What is a constructor?
A constructor in Java is a special method used to initialize objects of a class. It is automatically called when an object is created. Constructors can be used to set initial values for object attributes and perform other setup tasks.

48 . What is a default constructor?
A default constructor is a constructor with no parameters. If a class does not explicitly define any constructors, Java provides a default constructor with no arguments. This default constructor initializes instance variables to their default values and is called when an object is created without specifying constructor arguments.

50 . How do you call a super class constructor from a constructor?

You can call a superclass constructor from a constructor in a subclass using the `super` keyword. This is useful when you want to initialize the inherited attributes or perform additional setup tasks defined in the superclass's constructor.

Here's an example in Java:

```
class Animal {
    String name;


    // Superclass constructor
    public Animal(String name) {
        this.name = name;
        System.out.println("Animal constructor called.");
    }


    void speak() {
        System.out.println(name + " makes a sound.");
    }
}


class Dog extends Animal {
    // Subclass constructor
    public Dog(String name) {
        super(name); // Call the superclass constructor
        System.out.println("Dog constructor called.");
    }
```

```
  @Override

  void speak() {

    System.out.println(name + " barks.");

  }

}


public class SuperclassConstructorExample {

  public static void main(String[] args) {

    Dog myDog = new Dog("Buddy");

    myDog.speak();

  }

}
```

In this example:

- We have a superclass `Animal` with a constructor that initializes the `name` attribute.
- The `Dog` class is a subclass of `Animal`, and it has its own constructor.
- In the `Dog` constructor, we use `super(name);` to call the superclass constructor with the `name` parameter.
- When you create a `Dog` object, both the superclass (`Animal`) and subclass (`Dog`) constructors are called, and you can see the order in which they are invoked. This allows you to set up the object's state properly and reuse initialization logic from the superclass constructor.

52 . What is the use of this()?

Use of `this()`: `this()` is used to call another constructor within the same class. It is often used to avoid code duplication and ensure proper initialization of an object. For example, if a class has multiple constructors with different parameters, you can use `this()` to call one constructor from another.

53 . Can a constructor be called directly from a method?

Constructor Called from a Method: Yes, a constructor can be indirectly called from a method. This can happen when a method creates an instance of a class using the `new` keyword. The constructor is invoked when an object is created in this way.

```
class Car {
  String make;
```

```
    // Constructor to initialize the make of the car
    public Car(String make) {
        this.make = make;
    }
}

public class ConstructorFromMethodExample {
    public static void main(String[] args) {
        // Call a constructor directly from a method
        Car car1 = createCar("Toyota");
        Car car2 = createCar("Honda");

        // Print the make of the cars
        System.out.println("Car 1 Make: " + car1.make);
        System.out.println("Car 2 Make: " + car2.make);
    }

    // Method that creates a Car object and initializes it
    public static Car createCar(String make) {
        // Call the Car constructor to create and initialize a Car object
        return new Car(make);
    }
}
```

In this example:

- The `Car` class has a constructor that initializes the `make` attribute when a `Car` object is created.
- The `createCar` method is defined to create a `Car` object and initialize it with the specified make. It calls the `Car` constructor directly from within the method.
- In the `main` method, we call the `createCar` method twice to create two `Car` objects with different makes.

54 . Is a super class constructor called even when there is no explicit call from a sub class constructor?

Superclass Constructor Without Explicit Call: In Java, a superclass constructor is called even when there is no explicit call from a subclass constructor. If you don't explicitly call a superclass constructor using `super(...)`, the compiler inserts an implicit call to the default (no-argument) constructor of the superclass. If the superclass doesn't have a no-argument constructor, you must explicitly call one of its constructors using `super(...)`.