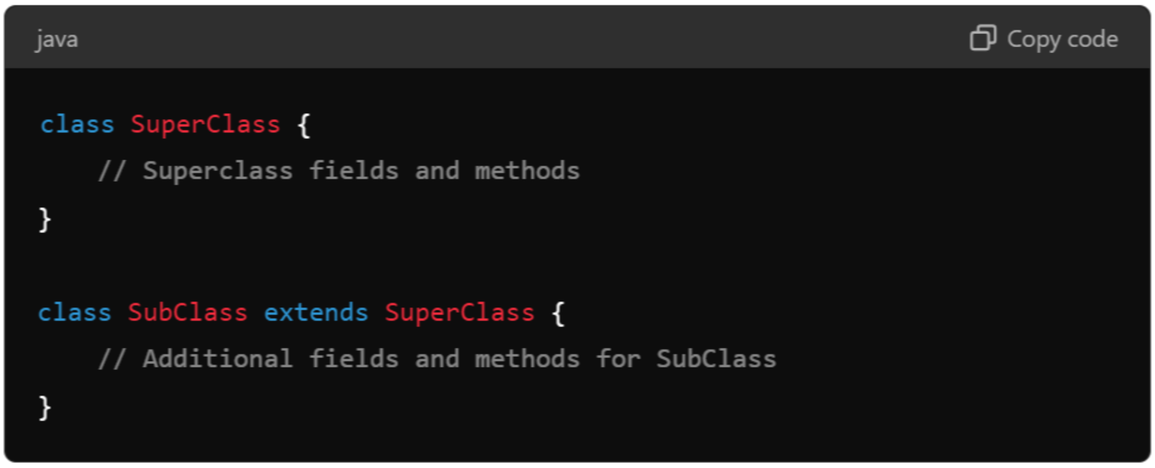
Java – MCA 2nd Internal Important Questions

1. What is Inheritance? Discuss different Inheritance Hierarchies. (10)

Inheritance is a **fundamental concept** in object-oriented programming (OOP) that allows a **new class** (known as a subclass or child class) to inherit **properties** and **behaviors** (**fields** and **methods**) from an **existing class** (known as a **superclass** or **parent** class). This promotes code reusability and establishes a natural hierarchical relationship between classes.

Syntax:



```java
class SuperClass {
    // Superclass fields and methods
}

class SubClass extends SuperClass {
    // Additional fields and methods for SubClass
}
```

Inheritance Hierarchies:

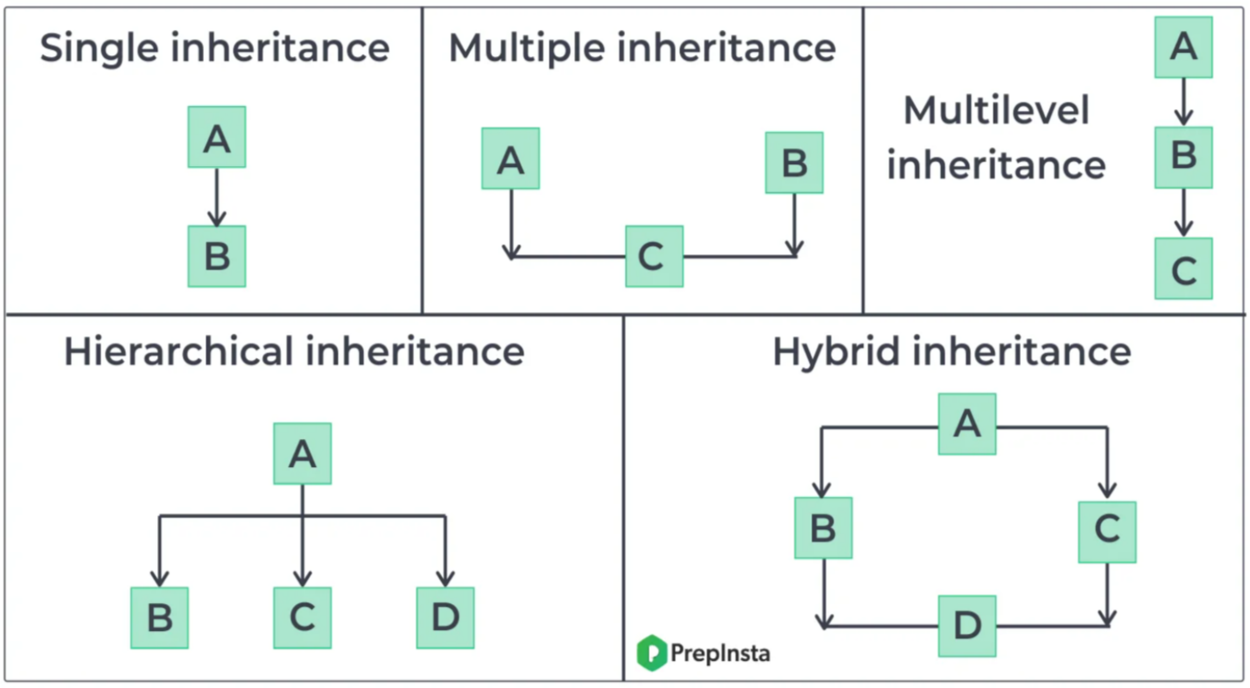
An **inheritance hierarchy** represents the relationships among classes in a multi-level manner, where classes are derived from other classes in a chain.

1. **Single Inheritance:** A subclass inherits from only one superclass.
2. **Multi-Level Inheritance:** A chain of inheritance where a class is derived from another derived class.
3. **Multiple Inheritance:** A class derived from more than one Super class.
4. **Hierarchical Inheritance:** Multiple subclasses inherit from a single superclass.

5. **Hybrid Inheritance:** A combination of more than one type of inheritance.

Note:

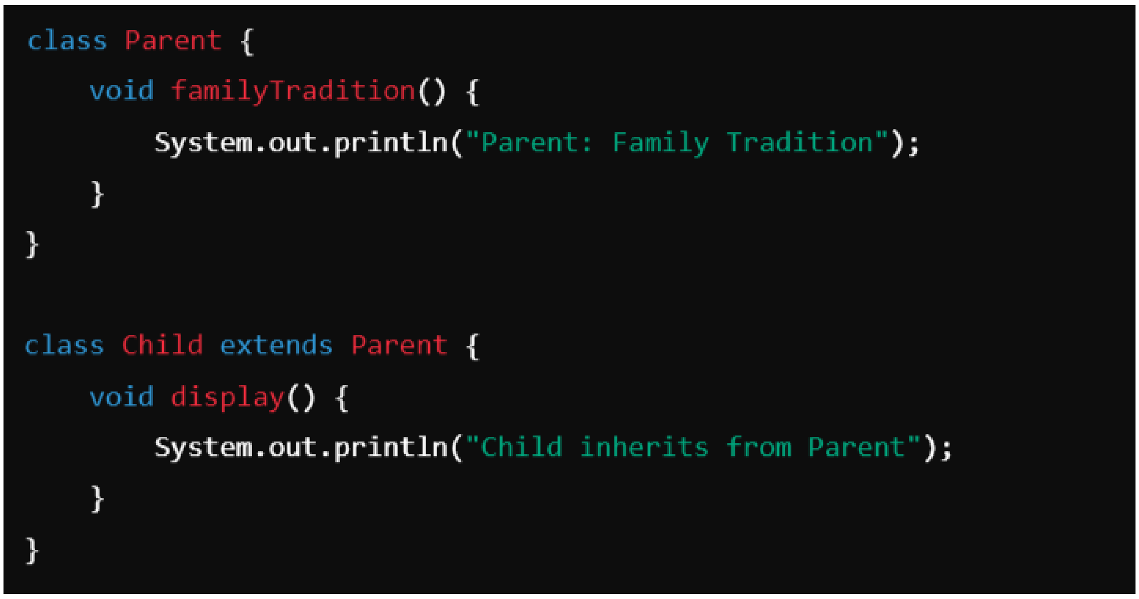
Java doesn't directly support multiple inheritance due to the **Diamond Problem**, but it can be achieved through interfaces.



1. Single Inheritance:

Single inheritance means **one class** inherits from **only one** parent class.

Example:

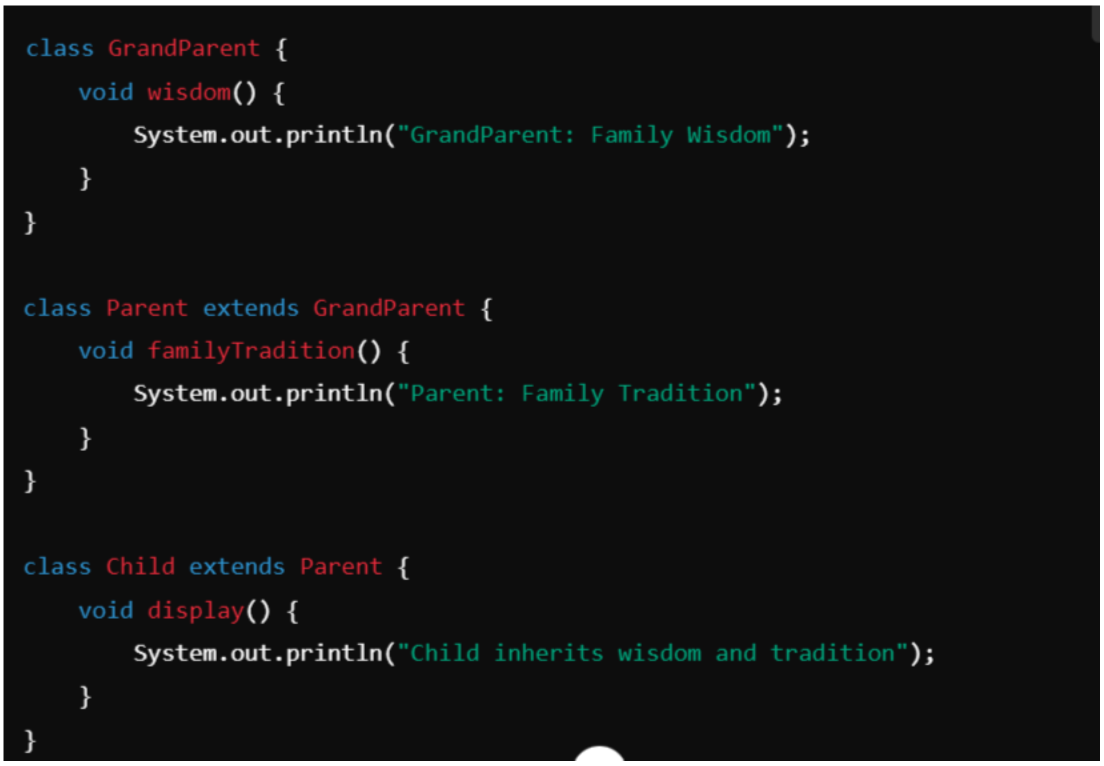


```java
class Parent {
    void familyTradition() {
        System.out.println("Parent: Family Tradition");
    }
}

class Child extends Parent {
    void display() {
        System.out.println("Child inherits from Parent");
    }
}
```

2. Multilevel Inheritance

Multilevel inheritance occurs when a class inherits from a class, which in turn inherits from another class. It forms a chain of inheritance.

Example:

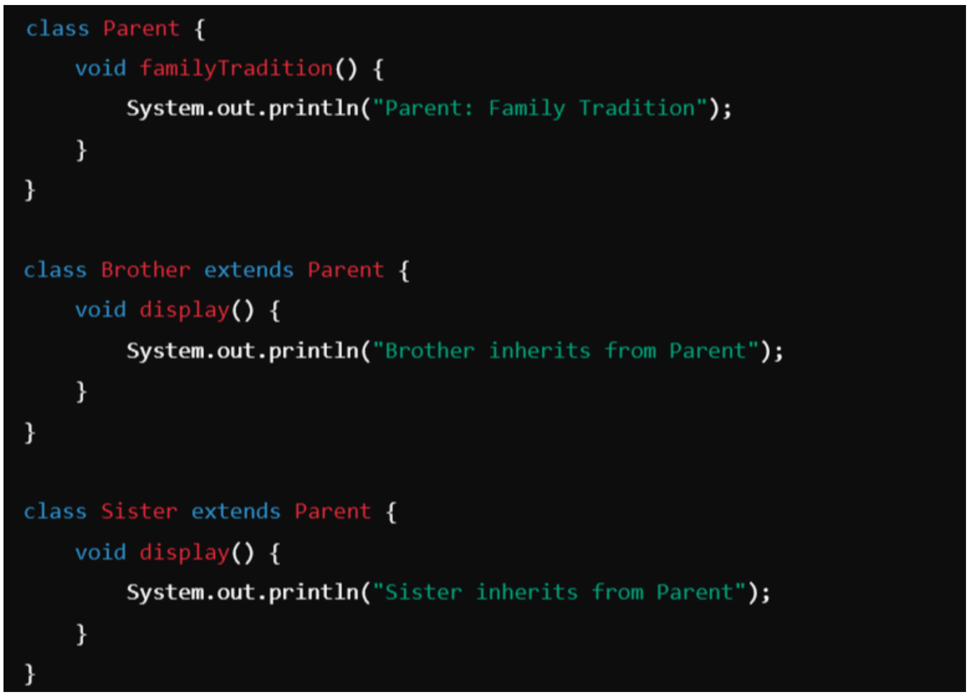


```java
class GrandParent {
    void wisdom() {
        System.out.println("GrandParent: Family Wisdom");
    }
}

class Parent extends GrandParent {
    void familyTradition() {
        System.out.println("Parent: Family Tradition");
    }
}

class Child extends Parent {
    void display() {
        System.out.println("Child inherits wisdom and tradition");
    }
}
```

3. Hierarchical Inheritance

Hierarchical inheritance occurs when **multiple classes inherit from a single parent class**.

Example:



```java
class Parent {
    void familyTradition() {
        System.out.println("Parent: Family Tradition");
    }
}

class Brother extends Parent {
    void display() {
        System.out.println("Brother inherits from Parent");
    }
}

class Sister extends Parent {
    void display() {
        System.out.println("Sister inherits from Parent");
    }
}
```

4. Hybrid Inheritance :

Hybrid inheritance is a mix of two or more types of inheritance. Java doesn’t support multiple inheritance directly through classes, but you can achieve it using **interfaces**.

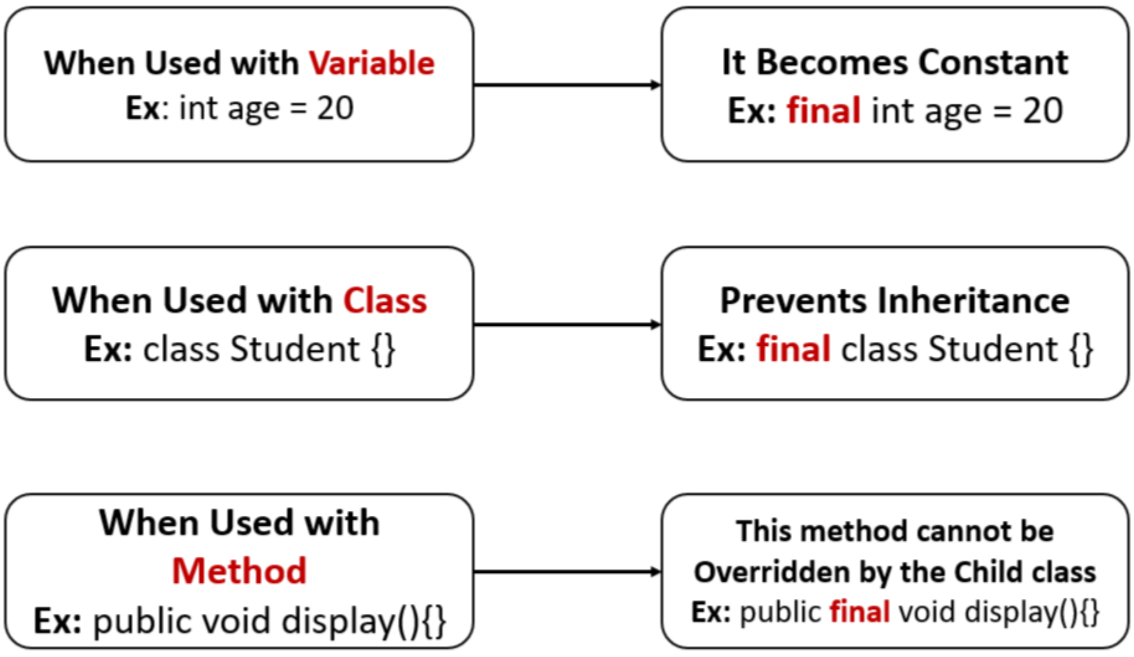
Example:

```java
// Grandparent class
class GrandParent {
    void familyWisdom() {
        System.out.println("GrandParent: Family wisdom passed down.");
    }
}

// Father class extends GrandParent
class Father extends GrandParent {
    void fatherTraits() {
        System.out.println("Father: Strong and responsible.");
    }
}

// Mother class extends GrandParent
class Mother extends GrandParent {
    void motherTraits() {
        System.out.println("Mother: Caring and nurturing.");
    }
}

// Child class extends Father
class Child extends Father {
    void childBehavior() {
        System.out.println("Child: Energetic and playful.");
    }
}

// Brother class extends Father
class Brother extends Father {
    void brotherTraits() {
        System.out.println("Brother: Protective of siblings.");
    }
}

// Sister class extends Mother
class Sister extends Mother {
    void sisterTraits() {
        System.out.println("Sister: Supportive and caring.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating instances and calling methods
        Child child = new Child();
        child.familyWisdom();  // Inherited from GrandParent
        child.fatherTraits();  // Inherited from Father
        child.childBehavior(); // Child's own method

        Brother brother = new Brother();
        brother.familyWisdom();  // Inherited from GrandParent
        brother.fatherTraits();  // Inherited from Father
        brother.brotherTraits(); // Brother's own method

        Sister sister = new Sister();
        sister.familyWisdom();  // Inherited from GrandParent
        sister.motherTraits();  // Inherited from Mother
        sister.sisterTraits();  // Sister's own method
    }
}
```

2. Explain the usage of final keyword with suitable example. (8)

final Keyword:

In Java, the keyword final is used to declare constants, restrict inheritance, and prevent method overriding or reassignment. It can be applied to **variables**, **methods**, and **classes**, each serving a specific purpose.



final Classes and Methods:

In Java, the final keyword can be used to mark **classes**, **methods**, and **variables**. When applied to a class or method, it has specific effects:

1. **final Classes:**

A **final class** is a class that cannot be **subclassed** or **extended**. This is useful when you want to prevent other classes from inheriting your class, ensuring that its implementation remains unchanged.

Syntax:

```
final class ClassName {
    // class body
}
```

Example:

```
final class Animal {
    String name;


    public void makeSound() {
        System.out.println(name + " is making a sound");
    }
}

// The following code will result in a compilation error
// class Dog extends Animal {
//     // Cannot extend Animal class
// }
```

2. final Methods:

A **final method** is a method that cannot be **overridden by subclasses**. This is useful when you want to ensure that a method's implementation remains unchanged in any subclass.

Syntax:

```
class MyClass {
    public final void myMethod() {
        // Method definition
    }
}
```

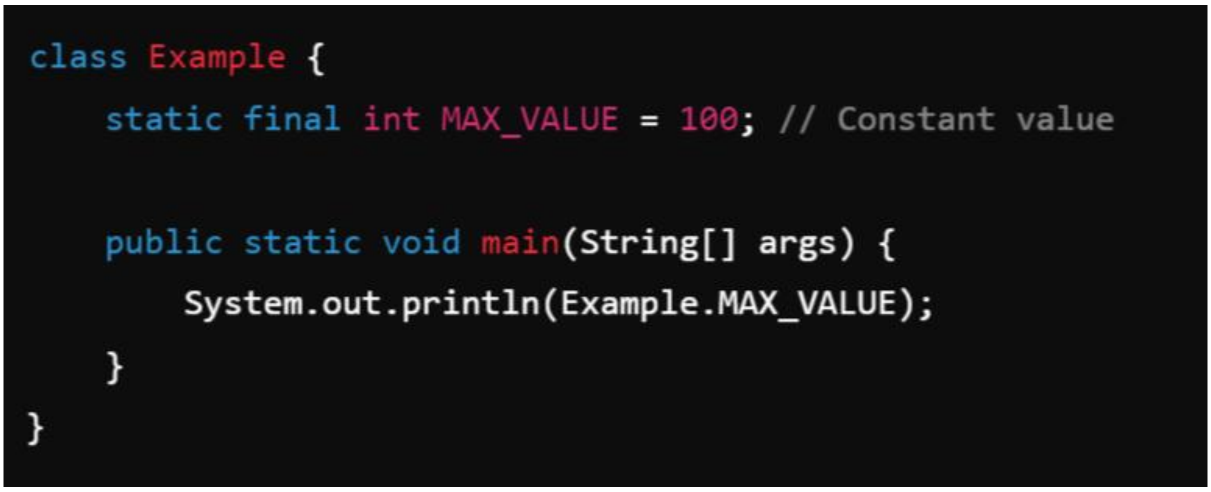
Example:

```java
public class Animal {
    String name;


    public final void makeSound() {
        System.out.println(name + " is making a sound");
    }
}

class Dog extends Animal{

    // Bellow Code will give Error: Cannot override the final method from Animal
    // public void makeSound() {
    //     System.out.println(name + " is making a sound");
    // }

}
```

3. **final Variable (Constant):**

   In Java, the final keyword is used to declare constants or make entities immutable. When a variable is declared as final, its value cannot be changed once it is assigned.

   Example: IN the below example we have a constant or final variable named MAX_VALUE whose content cannot be changed (Re-Initialized) once initialized.



```java
class Example {
    static final int MAX_VALUE = 100; // Constant value

    public static void main(String[] args) {
        System.out.println(Example.MAX_VALUE);
    }
}
```

3. Explain the usage of super keyword with suitable example. (8)

super Keyword:

The super keyword in Java is used in three main contexts:

1. Accessing the **Parent Class Constructor**
2. Accessing **Parent Class Methods**
3. Accessing **Parent Class Fields**

Overall, super is mainly used to **differentiate between members of a parent class and the current class,** ensuring the correct fields, methods, or constructors are accessed.

Note: The super keyword is used to refer to the immediate parent class. However, super can **only be used within a method or a constructor**. You cannot use super directly in the class body.

1. Accessing the Parent Class Constructor:

It is used to call a constructor of the parent class from a subclass. This is typically done to **initialize the parent class's fields when an instance of the subclass is created**.

Example:

```java
class Parent {
    Parent() {
        System.out.println("Parent Constructor");
    }
}

class Child extends Parent {
    Child() {
        super();  // Calls the Parent class constructor
        System.out.println("Child Constructor");
    }
}
```

2. Accessing Parent Class Methods:

super can be used to call a method from the parent class that has been overridden in the child class.

Example:

```java
class Parent {
    void display() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    void display() {
        super.display();  // Calls the Parent class method
        System.out.println("Child method");
    }
}
```

3. Accessing Parent Class Fields:

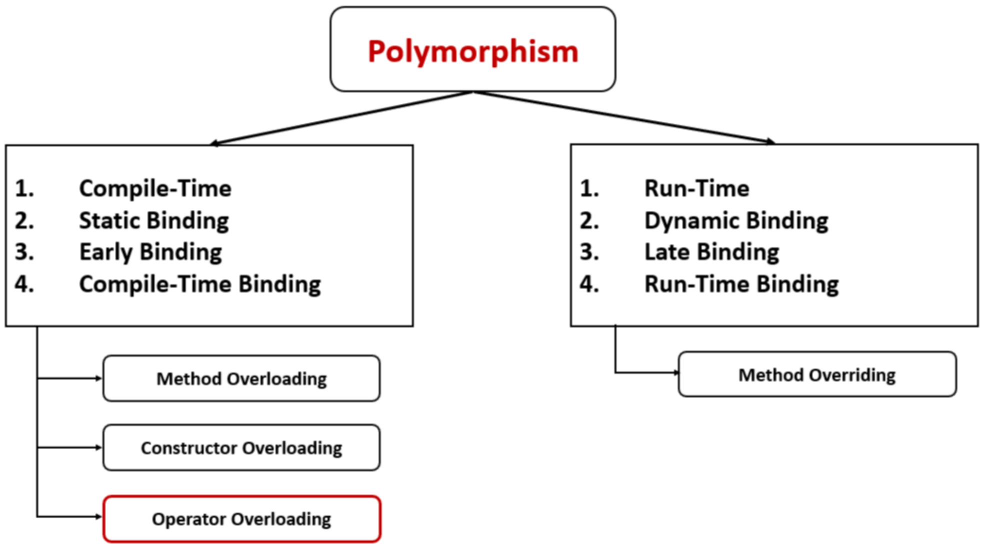
If a field in a subclass hides a field in its superclass, super can be used to refer to the superclass's field.

Example:

```java
class Parent {
    int value = 10;
}

class Child extends Parent {
    int value = 20;

    void showValue() {
        System.out.println(super.value);
// Accesses the Parent class field
    }
}
```

4. What is method overloading and method overriding. Illustrate with a suitable example. (10)

Polymorphism is one of the important pillars of OOP’s Principle. So, Polymorphism can be achieved using Method Overloading and Overriding.



1. Compile-Time Polymorphism (Static Binding or Early Binding):

Static binding, also known as early binding or compile-time binding, refers to the process where the method to be executed is determined at compile time rather than at runtime. This is in contrast to dynamic binding, which occurs at runtime.

Method Overloading and Constructor Overloading:

Method Overloading and Constructor Overloading in Java are concepts that allow **multiple methods** or **constructors to have the same name but different parameters** within the same class. They help improve code readability and flexibility by enabling the same method or constructor to perform different tasks based on the input parameters.

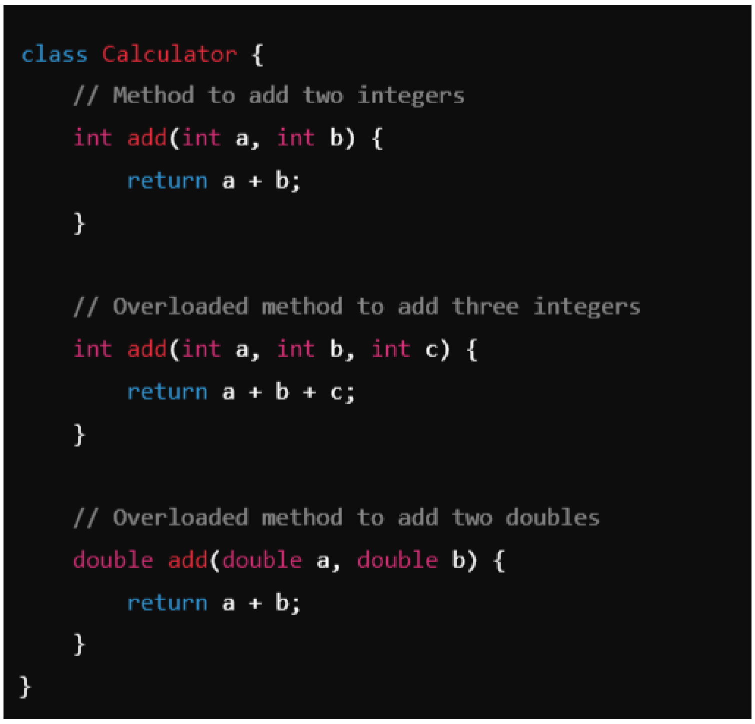
1. Method Overloading:

Definition: Method overloading occurs when two or more methods in the same class have the same name but differ in the number or type of parameters.

Key Points:

- Methods must have different parameter lists (either in **number**, **type**, or **order of parameters**).
- The return type can be the same or different, but it doesn't influence overloading.
- Overloading enhances code readability and reusability.

Example: In this example we can see that the method add is overloaded 3 times based on num of parameters passed to it. So we have 3 methods with same name but different num of arguments.



```java
class Calculator {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method to add two doubles
    double add(double a, double b) {
        return a + b;
    }
}
```

2. Run-Time Polymorphism: Dynamic Binding (Late Binding):

Dynamic binding in Java refers to the process by which a **method call** is **resolved to the appropriate method implementation at runtime**, **rather than at compile time.** This concept is crucial for achieving polymorphism in object-oriented programming.

Implementation:

Method overriding:

Method overriding in Java is a feature that allows a subclass to provide a specific implementation of a method that is **already defined in its superclass**. This is used to achieve runtime polymorphism and enable the subclass to customize or enhance the behavior of the inherited method.

Key Points of Method Overriding:

1. **Same Method Signature:** The overriding method in the subclass must have the same name, return type, and parameters as the method in the superclass.
2. **Annotation:** The **@Override** annotation is often used above the method in the subclass to indicate that it is overriding a method from its superclass (though it is **optional**).
3. **Access Modifiers:** The access level of the overriding method **cannot be more restrictive** than the overridden method.
4. **Instance Methods Only:** Only instance methods (**non-static methods**) can be overridden. Static methods are not overridden but are hidden instead.

Example:

Note: To achieve a Method Overriding **there should be a Inheritance**.

In the below example we can see we have a Parent or Super cass Animal and a Child class Dog. Dog class is Overridden the method sound() which is present in the Animal class which is a Parent.

```java
class Animal{

    public void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal{

    @Override
    public void sound() {
        System.out.println("Inside Dog!");
    }

}


class MainClass {

    public static void main(String[] args) {

        Dog d = new Dog();
        d.sound();

    }
}
```

Note: How to Prevent Method Overriding --- Use **final** Keyword.

5. Explain the significance of the **Object** class in Java and Explain any 5 methods of Object class with proper syntax and example. (10)

Object Class:

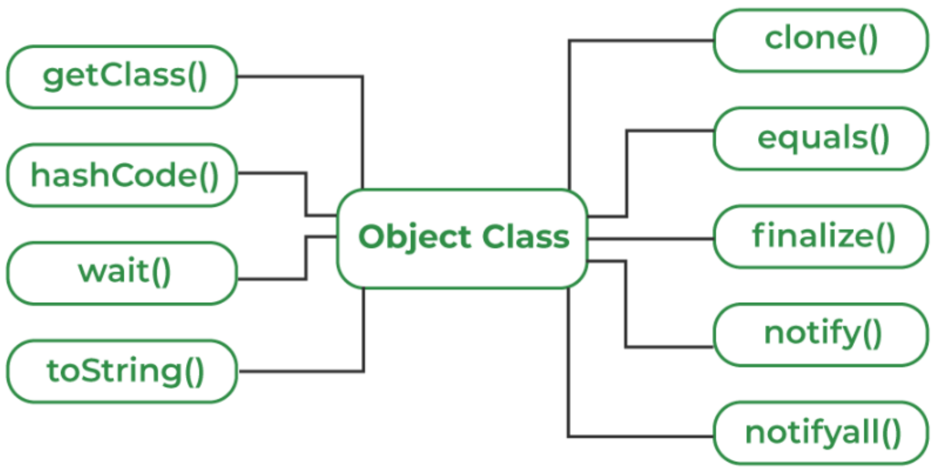
In Java, an Object is the root class of the Java class hierarchy. Every class in Java is implicitly a subclass of the Object class, either directly or indirectly. This means that all Java classes inherit the methods defined in the Object class.

When I say "**every class in Java is implicitly a subclass of the Object class**," I mean that no matter what class you create in Java, it **automatically inherits from the Object class**, even if you don’t explicitly specify a parent class. This inheritance happens by default.

Methods of Object Class:

Object class is present in **java.lang package**. Every class in Java is **directly** or **indirectly derived** from the Object class. If a class does not extend any other class then it is a direct child class of Object and if extends another class then it is indirectly derived. Therefore the **Object class methods** are available to all Java classes. Hence Object class acts as a root of the inheritance hierarchy in any Java Program.

Methods:



Note: All the methods should be overridden from the Parent Object class to our own class, For the exam perspective I have giving only the implementation of the Methods not the full code as in the notes.

1. **toString():** Returns a **string representation** of the object.

Syntax:

public String toString()

Example:

```
@Override
public String toString() {
    return "MyClass{id=" + id + ", name='" + name + "'}";
}
```

2. equals(Object obj): Compares **this** object with the **specified object** for equality.

Syntax:

public boolean equals(Object obj)

Example: The equals method in the above example is a custom implementation of the equals() method in Java. This method is used to compare two objects for equality, and it's typically overridden in classes where you want to define what it means for two objects to be considered "equal."

```
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        MyClass myClass = (MyClass) obj;
        return id == myClass.id && name.equals(myClass.name);
    }
```

3. hashCode(): Returns a **hash code value** for the object, which is used in hashing-based collections like HashMap.

A hash code value is an integer that is generated by the hashCode() method in Java. This value is used to uniquely represent an object in hashing data structures like HashMap, HashSet, and Hashtable.

Syntax:

public int hashCode()

Example:

```
    @Override
    public int hashCode() {
        return id * 31 + name.hashCode();
    }
```

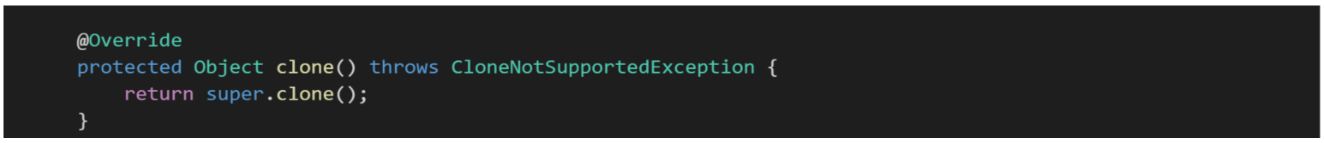
4. clone():

Purpose: Creates and returns a copy (clone) of the object. The class must **implement** the **Cloneable interface**.

Syntax:

```
protected Object clone() throws CloneNotSupportedException
```

Example:



```java
@Override
protected Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

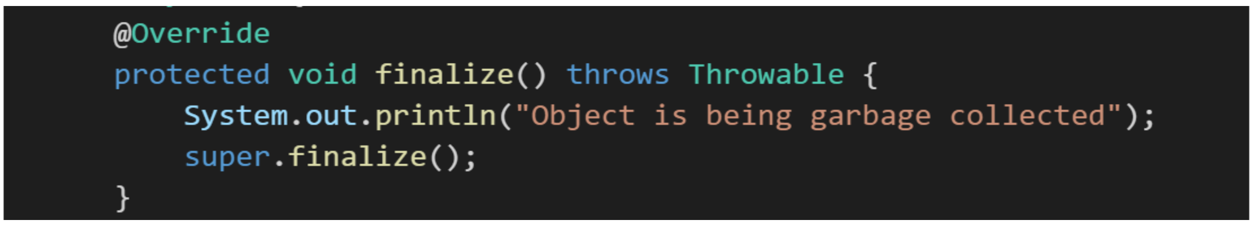
5. finalize():

The finalize() method in Java is a special method that the garbage collector calls before an object is removed from memory. It allows the object to perform any cleanup operations, such as releasing resources or closing files. However, it is **rarely used** because relying on finalize() can lead to unpredictable behavior. So this will be handled by the Garbage Collector Automatically.

Syntax:

protected void finalize() throws Throwable

Example:



```java
@Override
protected void finalize() throws Throwable {
    System.out.println("Object is being garbage collected");
    super.finalize();
}
```

6. notify() :

In Java, the notify() method is used in **multi-threaded programming** to **wake up** a single thread that is **waiting on the object's monitor** (lock). Here's a brief explanation:

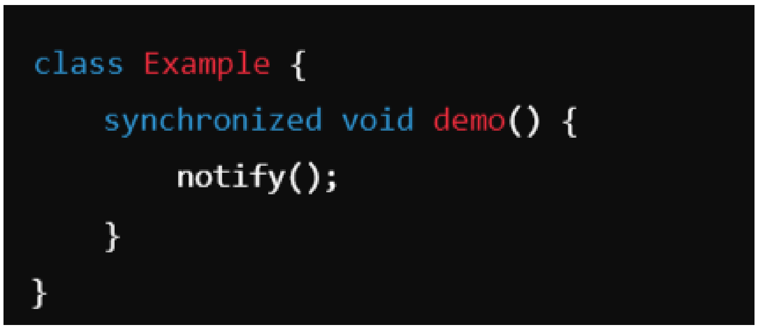
1. **Context:** When multiple threads are involved in a task, some threads might need to wait for certain conditions to be met before they can proceed. This is typically done using the wait() method, which causes a thread to wait until another thread notifies it that it can continue.

2. **Usage of notify():** The notify() method is called on an object to wake up one of the threads that is currently waiting on that object's monitor. Only one thread is awakened, and it is chosen by the JVM (Java Virtual Machine) in a somewhat random fashion if multiple threads are waiting.

Syntax:

public final void **notify**()

Example:



```
class Example {
    synchronized void demo() {
        notify();
    }
}
```

Synchronized Method (demo()):

The demo() method is marked as **synchronized**. This means that when a thread calls demo() on an instance of Example, it acquires the **lock** (monitor) on that instance before executing the method.

Only one thread can execute the demo() method at a time on the same object. If another thread tries to call demo() or any other synchronized method on the same object, it will block (wait) until the lock is released.

So here only one instance of Example will hold the control until and unless it calls the demo() method and inside it will execute the notify() method to release the lock, so that other objects can access it. notify() is used to wake up one thread that is waiting on the object’s lock (monitor).

6. Explain **abstract class** and **methods** with an example. (10)

Abstraction or Data Abstraction is one of the important Principles of OOP's. In java we can achieve this by using **abstract class** and **methods** and **Interface.**

Abstraction is the concept of **hiding the complex implementation details** and **showing only the essential features of an object**.

Abstract Classes and Methods:

1. Abstract Method:

An abstract method is a method declared **without a body**. It must be overridden in the subclass. The method signature includes the **abstract** keyword, and there is no method body.

Key Rules for Abstract Methods:

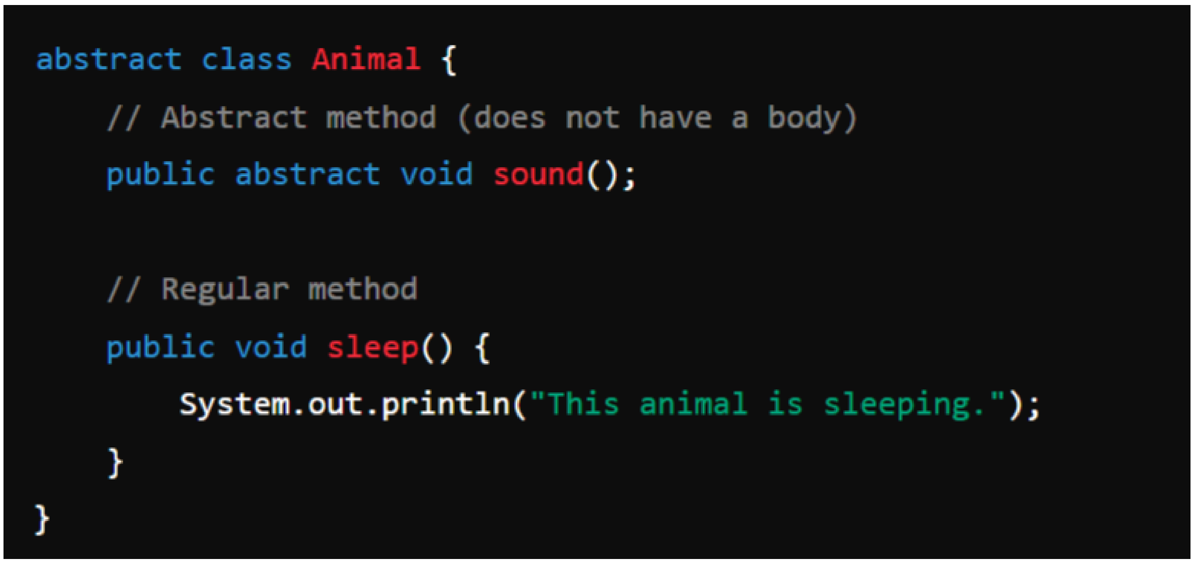
1. Must be declared in an abstract class.
2. Cannot have a method body.
3. Subclasses that extend the abstract class must override the abstract method.

2. Abstract Class:

An **Abstract Class** in object-oriented programming is a class that **cannot be instantiated directly**. It serves as a **blueprint for other classes**. Abstract classes are used when you want to define some common behavior (methods) that other classes should inherit and implement, but the abstract class itself should not be instantiated. In other words, an abstract class can have **abstract methods** (without implementation) and/or **concrete methods** (with implementation).

Example: In the below example we have a abstract method named sound() without any body, so a concrete or normal class cannot have a abstract method, so that's why the class is also made as abstract class.

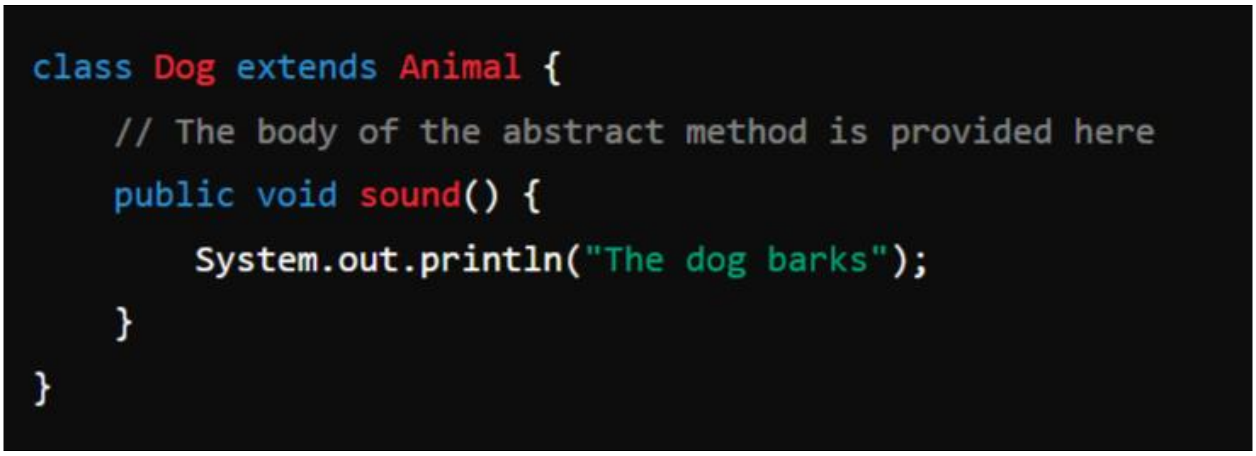
Now, whichever class inherits the Animal class, they should provide the method body.



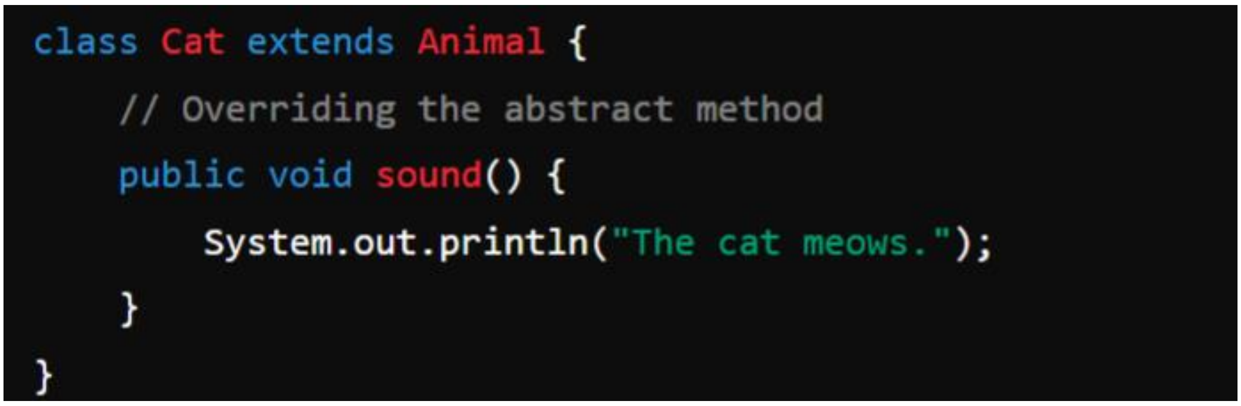
```java
abstract class Animal {
    // Abstract method (does not have a body)
    public abstract void sound();

    // Regular method
    public void sleep() {
        System.out.println("This animal is sleeping.");
    }
}
```

Usage:

Here we can see that the Dog and Cat both the class are child class to Animal, so they have provided the body to the abstract method sound() which is present in the Parent class Animal.



```java
class Dog extends Animal {
    // The body of the abstract method is provided here
    public void sound() {
        System.out.println("The dog barks");
    }
}
```



```java
class Cat extends Animal {
    // Overriding the abstract method
    public void sound() {
        System.out.println("The cat meows.");
    }
}
```

Rules for Abstract Classes:

1. Cannot be **instantiated.**
2. Can have **abstract methods.**
3. Can have **concrete methods.**
4. Subclasses **must override** abstract methods.
5. Subclasses can be **abstract.**

7. Write a short note on static keyword. (4)

In Java, the static keyword is used to define fields (variables) and methods that **belong to the class rather than to instances** of the class. This means that static members are **shared among all instances** of the class and can be accessed **without creating an instance of the class.**

1. Static Fields (Static Variables)

A static field is **shared among all instances of a class**. Unlike instance variables, which are unique to each object, static variables have the same value for all objects of the class. There is only one copy of the static variable in memory, regardless of how many instances of the class are created.

Key Points:

- Declared using the static keyword.
- Belongs to the class, not to any instance.

- Shared among all instances of the class.
- Can be accessed directly using the class name.

Example:

In this example:

- **count** is a static variable, shared across all instances of Counter.
- Every time a Counter object is created, count is incremented.
- The static method **displayCount**() is used to display the value of the count variable.

```java
class Counter {
    static int count = 0; // Static variable

    Counter() {
        count++; // Increment the static variable
    }

    public static void displayCount() {
        System.out.println("Count: " + count);
    }
}

public class Main {
    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        Counter c3 = new Counter();

        Counter.displayCount(); // Output: Count: 3
    }
}
```

2. Static Methods

A static method **belongs to the class, not to any instance of the class**. It can be called **without creating an object** of the class. Static methods can only access static fields and other static methods. They cannot access instance variables or instance methods directly.

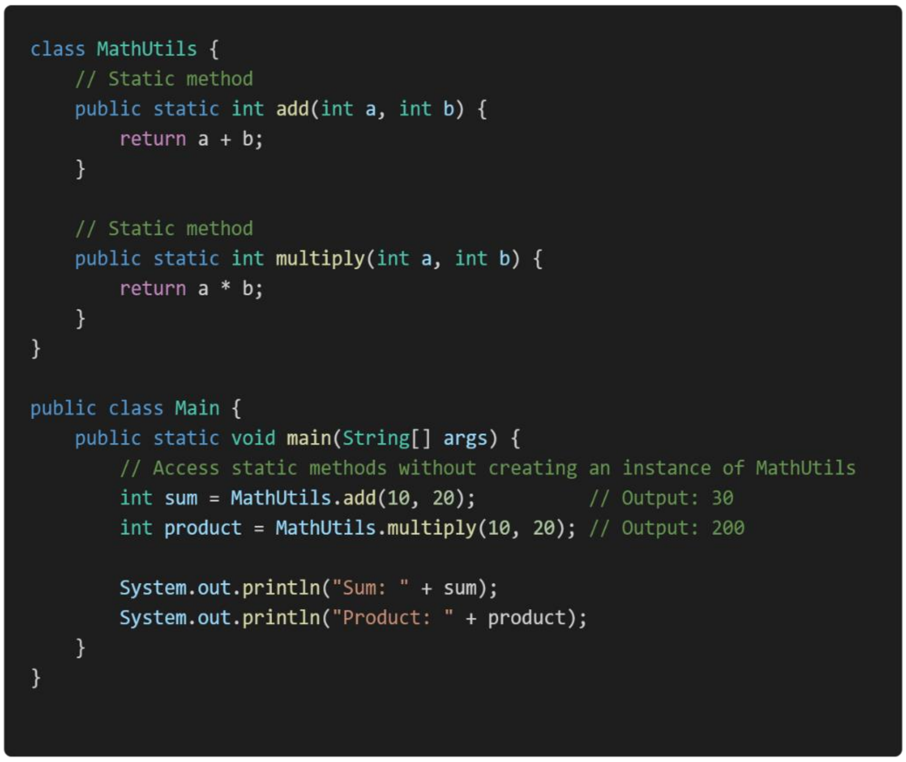
Key Points:

- Declared using the static keyword.
- Can be called without creating an instance of the class.
- Cannot access instance variables or instance methods directly.
- Useful for utility or helper methods (e.g., Math.pow()).

Example:

In this example:

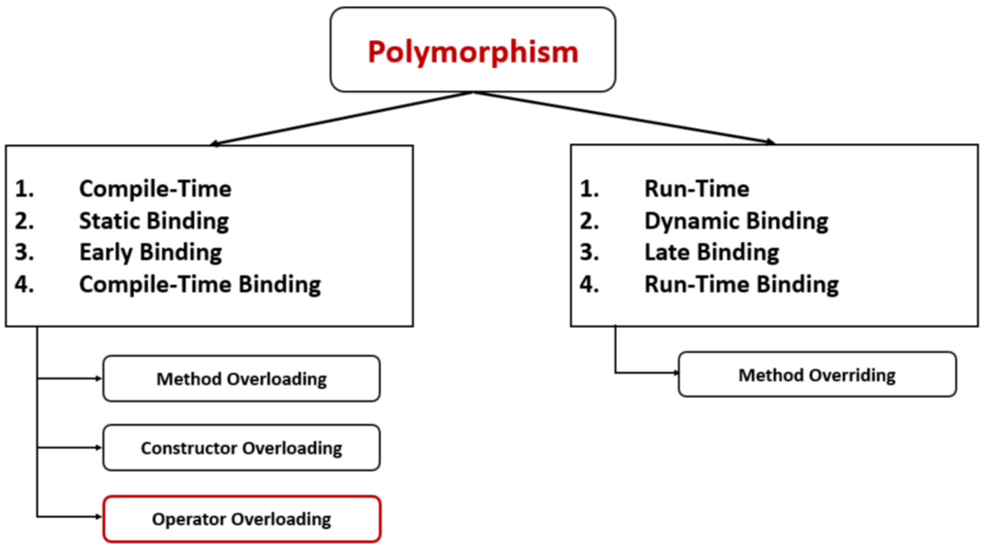
- The methods **add()** and **multiply()** are static, so they can be called **without creating an object** of MathUtils.
- The static methods are designed to perform operations that **don't rely on any instance state.**



```java
class MathUtils {
    // Static method
    public static int add(int a, int b) {
        return a + b;
    }

    // Static method
    public static int multiply(int a, int b) {
        return a * b;
    }
}

public class Main {
    public static void main(String[] args) {
        // Access static methods without creating an instance of MathUtils
        int sum = MathUtils.add(10, 20);          // Output: 30
        int product = MathUtils.multiply(10, 20); // Output: 200

        System.out.println("Sum: " + sum);
        System.out.println("Product: " + product);
    }
}
```

8. Explain **method overloading** and **Constructor Overloading** with suitable examples. (10)

   Polymorphism is one of the important pillars of OOP's Principle. So, Polymorphism can be achieved using Method Overloading and Overriding.



1. Compile-Time Polymorphism (Static Binding or Early Binding):

Static binding, also known as early binding or compile-time binding, refers to the process where the method to be executed is determined at compile time rather than at runtime. This is in contrast to dynamic binding, which occurs at runtime.

Method Overloading and Constructor Overloading:

Method Overloading and Constructor Overloading in Java are concepts that allow **multiple methods** or **constructors to have the same name but different parameters** within the same class. They help improve code readability and flexibility by enabling the same method or constructor to perform different tasks based on the input parameters.

1. Method Overloading:

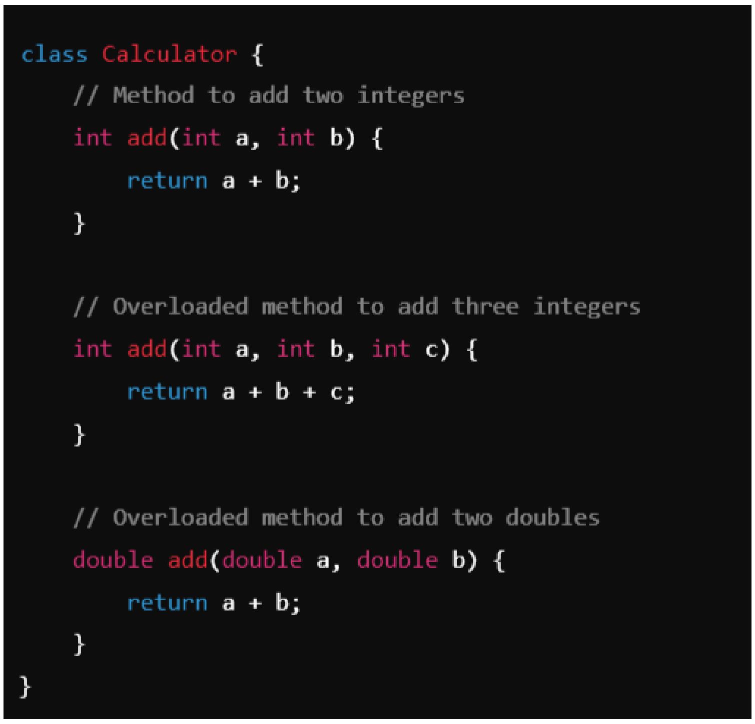
Definition: Method overloading occurs when two or more methods in the same class have the same name but differ in the number or type of parameters.

Key Points:

- Methods must have different parameter lists (either in **number**, **type**, or **order of parameters**).

- The return type can be the same or different, but it doesn't influence overloading.
- Overloading enhances code readability and reusability.

Example: In this example we can see that the method add is overloaded 3 times based on num of parameters passed to it. So we have 3 methods with same name but different num of arguments.



```
class Calculator {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method to add two doubles
    double add(double a, double b) {
        return a + b;
    }
}
```

2. Constructor Overloading:

Definition: Constructor overloading occurs when a class has **multiple constructors** with the **same name** (the class name) **but different parameter lists.**

Key Points:

- Allows creating objects in different ways depending on the arguments passed.
- Like method overloading, constructors must **differ in** the **number**, **type**, or **order of parameters**.

- It provides flexibility in initializing objects with different sets of data.

Example: Note: Constructer is a **special form of method**. So Constructor is indeed a method but it has some additional features.

```java
class Student {
    String name;
    int age;
    int id;

    // Constructor 1: No parameters, initializes with default values
    Student() {
        this.name = "Unknown";
        this.age = 0;
        this.course = "None";
        this.id = 0;
    }

    // Constructor 2: Initializes with name and age
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Constructor 4: Initializes with all properties
    Student(String name, int age, int id) {
        this.name = name;
        this.age = age;
        this.id = id;
    }
}
```

9. Explain the following:

a. **final**
b. **abstract**

final and abstract are the 2 important keywords used in Object Oriented Programming in java, each has its own functionalities.

Note: for this question answer is there in the above questions copy paste that.

final – final field, method, class

abstract – abstract class and method

10. What is **Method Overriding**? Explain how it allows Java to support **Run-Time Polymorphism** with an example. (10)

Note: for this question answer is there in the above questions copy paste that.

Run-Time Polymorphism: Dynamic Binding (Late Binding):
Dynamic binding in Java refers to the process by which a **method call** is **resolved to the appropriate method implementation at runtime**, **rather than at compile time.** This concept is crucial for achieving polymorphism in object-oriented programming.