Module-2:

MULTIPLE INHERITANCE

PRAMOD NAIK

Module-2: MULTIPLE INHERITANCE

Inheritance: Inheritance hierarchies, super and subclasses, member access rules, super keyword,

Preventing inheritance: final classes and methods, the object class and its methods.

Polymorphism: dynamic binding, method overriding abstract classes and methods.

Inheritance:

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.

Key Points:

Super Class (Parent Class):

The class from which properties and methods are inherited.

Example: Animal might be a superclass.

Sub Class (Child Class):

The class that inherits from another class. It can add its own properties and methods, or override the inherited ones.

Example: Dog might be a subclass of Animal.

PRAMOD NAIK

Syntax:

java

Copy code

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

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

PRAMOD NAIK

Example:

```java
// Superclass
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Subclass
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();  // Inherited method from Animal
        myDog.bark(); // Method of Dog class
    }
}
```

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:

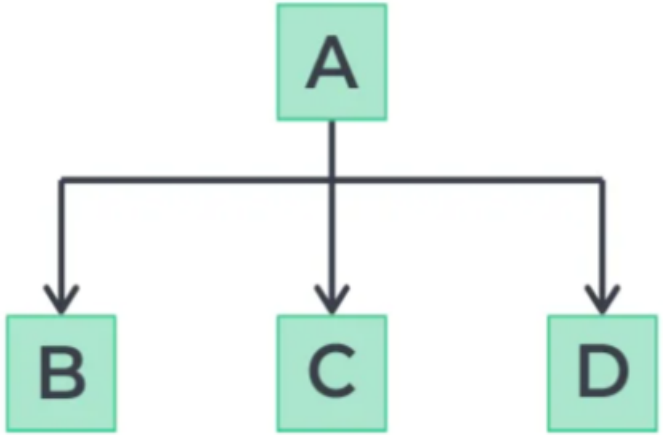
Java doesn't directly support multiple inheritance due to the **Diamond Problem**, but it can be achieved through interfaces.

Single inheritance

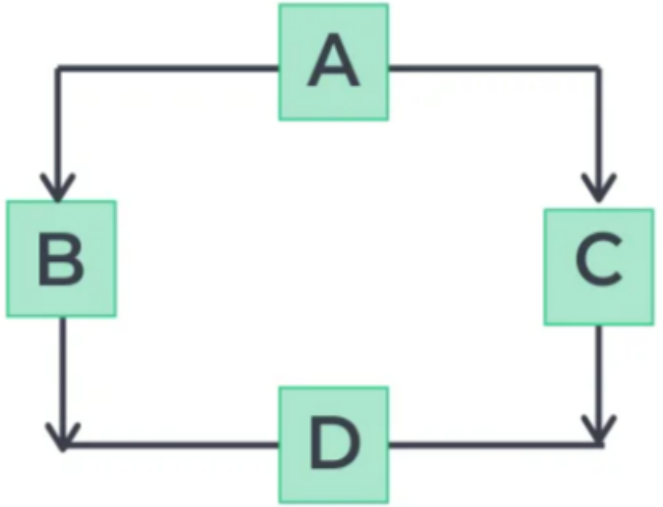
Multiple inheritance

Multilevel inheritance

Hierarchical inheritance



Hybrid inheritance



PRAMOD NAIK

Example:

```java
// Superclass
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}
```

```java
// Subclass 1
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}       You, 8 seconds ago • Uncommitted changes
```

```java
// Subclass 2
class Cat extends Animal {
    void meow() {
        System.out.println("The cat meows.");
    }
}
```

```java
// Subclass of Dog
class Puppy extends Dog {
    void play() {
        System.out.println("The puppy plays.");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Puppy myPuppy = new Puppy();
        myPuppy.eat();  // Inherited from Animal
        myPuppy.bark(); // Inherited from Dog
        myPuppy.play(); // Method of Puppy class
    }
}
```

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:

super: 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:

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

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

PRAMOD NAIK

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:

```
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.

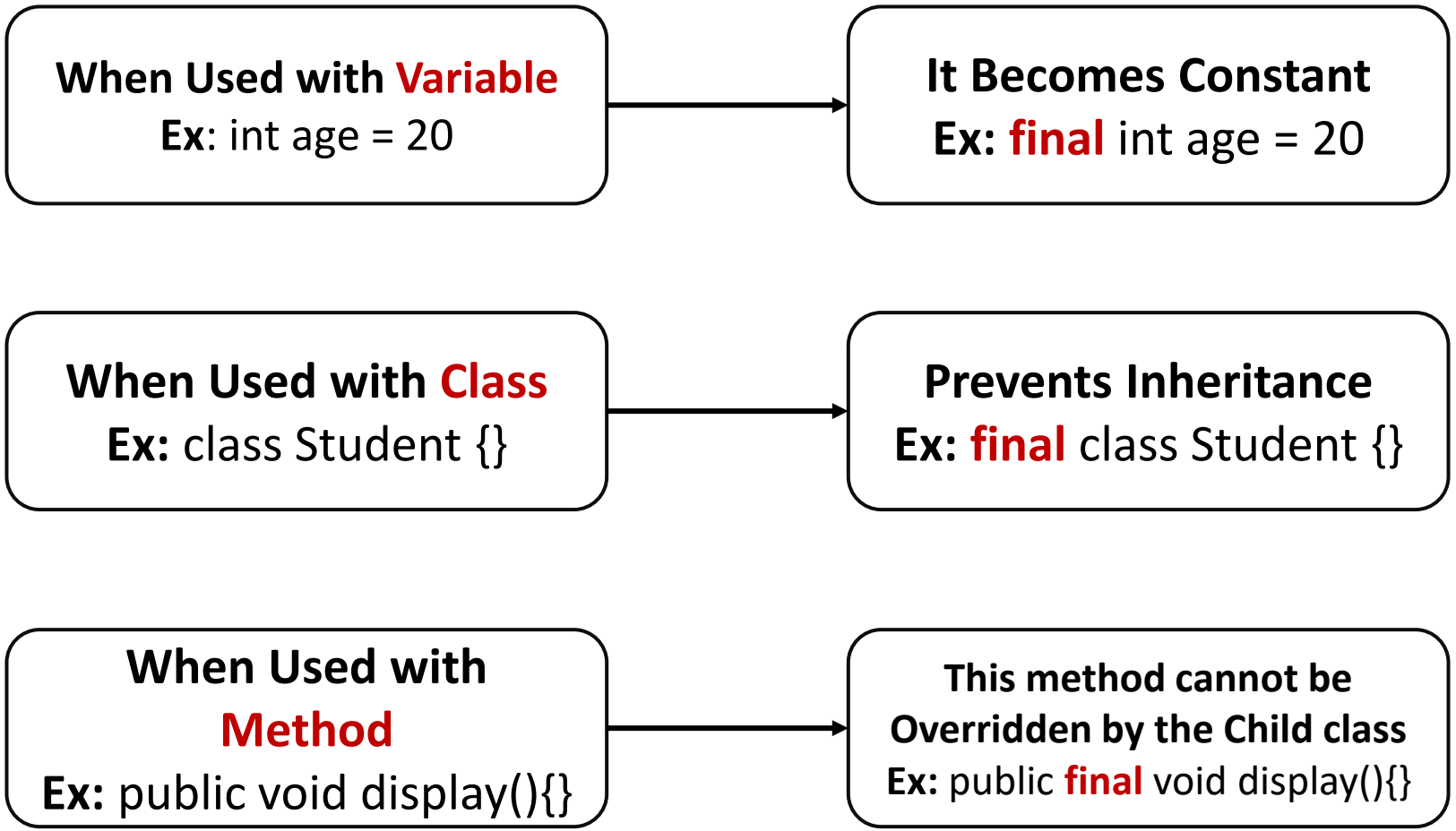
Example:

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

class Child extends Parent {
    int value = 20;

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

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.



PRAMOD NAIK

Preventing inheritance:

final Keyword:

In Java, the final keyword can be applied to **classes** and **methods** to restrict their usage or modification.

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:

```
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");
    // }

}
```

Object Class:

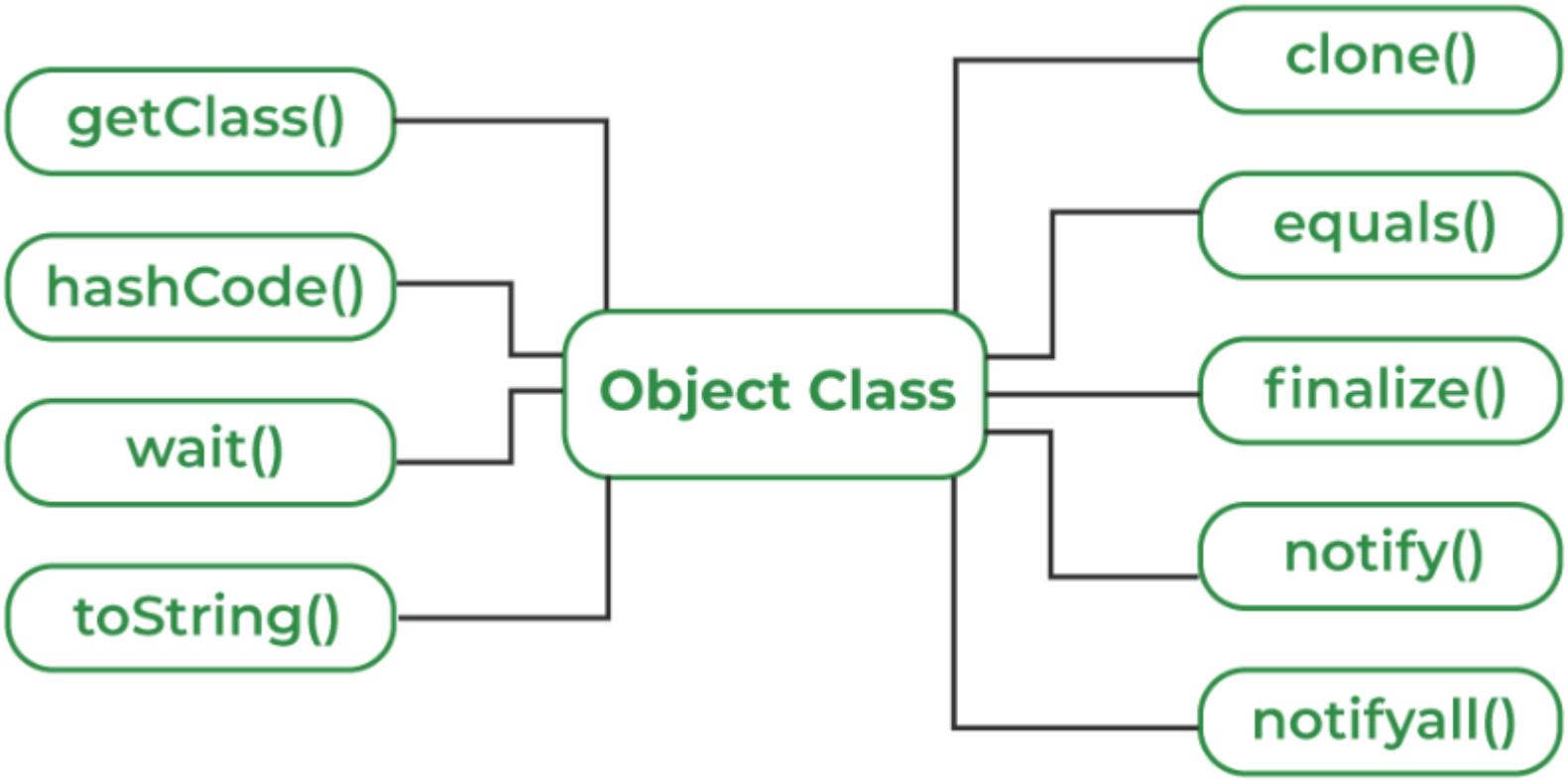
In Java, an Object is the root class of the Java class hierarchy. Every class in Java is implicitly a subclass of the Object class, either directly or indirectly. This means that all Java classes inherit the methods defined in the Object class.

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:



PRAMOD NAIK

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

Syntax:

```
public String toString()
```

Example:

```
class MyClass {
    int id;
    String name;

    MyClass(int id, String name) {
        this.id = id;
        this.name = name;
    }

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

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass(1, "Object 1");
        System.out.println(obj.toString());  // Output: MyClass{id=1, name='Object 1'}
    }
}
```

PRAMOD NAIK

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

Syntax:

```
public boolean equals(Object obj)
```

PRAMOD NAIK

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."

```java
class MyClass {
    int id;
    String name;

    MyClass(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @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);
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass(1, "Object 1");
        MyClass obj2 = new MyClass(1, "Object 1");
        System.out.println(obj1.equals(obj2));  // Output: true
    }
}
```

PRAMOD NAIK

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:

```java
class MyClass {
    int id;
    String name;

    MyClass(int id, String name) {
        this.id = id;
        this.name = name;
    }

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

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass(1, "Object 1");
        System.out.println(obj.hashCode());  // Output: A unique integer hash code
    }
}
```

PRAMOD N

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:

```
class MyClass implements Cloneable {
    int id;
    String name;

    MyClass(int id, String name) {
        this.id = id;
        this.name = name;
    }

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

public class Main {
    public static void main(String[] args) {
        try {
            MyClass obj1 = new MyClass(1, "Object 1");
            MyClass obj2 = (MyClass) obj1.clone();
            System.out.println(obj1.equals(obj2));  // Output: true (based on custom equals implementation)
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

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:

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

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj = null;
        System.gc();  // Requesting JVM to run Garbage Collector
    }
}
```

6. getClass(): Returns the **runtime class** of the object.

In Java, the getClass() method is used to obtain the runtime class of an object. The runtime class of an object refers to the **actual class type of the object** as it exists during the execution of the program, not necessarily the type as known at compile time.

For example, if you have a variable declared as Object but it actually references an instance of String, calling getClass() on that variable will return the Class object representing String. This can be useful for reflection, debugging, or when you need to perform operations based on the exact type of the object at runtime.

Example-1:

```
public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass(1, "Object 1");
        System.out.println(obj.getClass());  // Output: class MyClass
    }
}
```

Example-2:

```
Object obj = new String("Hello");
Class<?> clazz = obj.getClass();
System.out.println(clazz.getName()); // Output: java.lang.String
```

7. 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).

8. notifyAll(): Wakes up **all threads** that are waiting on this object's monitor (**Internal Lock**). This lock, or monitor, is a mechanism that ensures that only **one thread** can access a synchronized block or method on that object at a time.

Syntax: `public final void notifyAll()`

1. When a thread calls notifyAll() on an object, all threads that are currently waiting on that object's monitor (using the wait() method) are awakened.
2. These threads do not immediately resume execution; they must re-acquire the lock on the object before they can proceed. Since only one thread can hold the lock at a time, the awakened threads will compete to acquire it.
3. Like notify(), notifyAll() must be called from a synchronized context, meaning the thread must have the monitor (lock) of the object.

Example:

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

PRAMOD NAIK

9. **wait()** and **wait(long timeout):**

Causes the **current thread** to **wait** until another thread invokes **notify()** or **notifyAll()** on this object. So the wait() method will make the current thread **pause its execution** until another thread **signals it to continue.**

Syntax:

```
public final void wait() throws InterruptedException
```

```
public final void wait(long timeout) throws InterruptedException
```

Example:

```
class Example {
    synchronized void demo() throws InterruptedException {
        wait();
    }
}
```

The **wait(1000)** method is called inside demo(), which tells the **current thread** to **wait** for up to **1,000 milliseconds** (1 second).

This method causes the **current thread to release the lock** on the object (in this case, the instance of Example), and the thread enters a waiting state.

Polymorphism:

In Polymorphism **Poly means many, morphism means form** together it says **More than One Form**.

Polymorphism allows **objects of different classes** to be treated as **objects of a common superclass** (through Method Overriding or Dynamic/ Late Binding). It is mainly achieved through **method overriding** (runtime polymorphism) and **method overloading** (compile-time polymorphism).

Important:

1. **Method** & **Constructor Overloading** & **Operator Overloading** (**compile-time or Static polymorphism**)
2. **Method Overriding** (through Inheritance) (**runtime or Dynamic polymorphism**)

Polymorphism

1. Compile-Time
2. Static Binding
3. Early Binding
4. Compile-Time Binding

- Method Overloading
- Constructor Overloading
- Operator Overloading

1. Run-Time
2. Dynamic Binding
3. Late Binding
4. Run-Time Binding

- Method Overriding

PRAMOD NAIK

1. Static Binding (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:

```
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;
    }
}
```

PRAMOD NAIK

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:

```
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;
    }
}
```

PRAMOD NAIK

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:

```
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();

    }
}
```

How to Prevent Method Overriding

PRAMOD NAIK

Data Abstraction

Abstract Classes and Methods

PRAMOD NAIK

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.

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:

```
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:

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

```
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

PRAMOD NAIK