# Java OOPs Concepts

1. Classes, Objects and Java Interfaces
2. Constructors
3. Object Class
4. Abstraction
5. Encapsulation
6. Inheritance
7. Interfaces and Inheritance
8. Polymorphism
9. Packages
10. Functional Interface
11. Marker Interface
12. Comparator Interface

# Classes, Objects and Java Interfaces

In Java, classes and objects are basic concepts of Object Oriented Programming (OOPs) that are used to represent real-world concepts and entities. The class represents a group of objects having similar properties and behavior, or in other words, we can say that a class is a blueprint for objects, while an object is an instance of a class. For example, the animal type Dog is a class while a particular dog named Tommy is an object of the Dog class. We will discuss Java classes and objects and how to implement them in our program.

## Difference Between Java Classes and Objects

The table below demonstrates the difference between classes and objects in Java:

| Class | Object |
|---|---|
| Class is the blueprint of an object. It is used to create objects. | An object is an instance of the class. |
| No memory is allocated when a class is declared. | Memory is allocated as soon as an object is created. |
| A class is a group of similar objects. | An object is a real-world entity such as a book, car, etc. |
| Class is a logical entity. | An object is a physical entity. |
| A class can only be declared once. | Objects can be created many times as per the requirement. |
| An example of class can be a car. | Objects of the class car can be BMW, Mercedes, Ferrari, etc. |

## Java Classes

A class in Java is a set of objects that share common characteristics and common properties. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named X is an object.

### Properties of Java Classes

- Class is not a real-world entity. It is just a template or blueprint, or a prototype from which objects are created.
- Class does not occupy memory.
- A class is a group of variables of different data types and a group of methods.
- A Class in Java can contain:
    1. Data member
    2. Method
    3. Constructor
    4. Nested Class
    5. Interface

### Class Declaration in Java

```
access_modifier class <class_name> {
    data member;
    method;
    constructor;
    nested class;
    interface;
 }
```

### Components of Java Classes

In general, class declarations can include these components, in order:

- **Modifiers:** A class can be public or has default access (Refer this for details).
- **Class keyword:** Class keyword is used to create a class.
- **Class name:** The name should begin with an initial letter (capitalized by convention).
- **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body:** The class body is surrounded by braces, { }.

Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects. There are various types of classes that are used in real-time applications such as nested classes, anonymous classes and lambda expressions.

**Example 1:** Here, the below Java code demonstrates the basic use of class in Java.

```
class Student {
    int id;
    String n;

    public static void main(String args[]) {
        Student s1 = new Student();
        System.out.println(s1.id);
        System.out.println(s1.n);
    }
}
Output
0
null
```

**Example 2:** Here, the below Java code demonstrates creating an object using the newInstance() method.

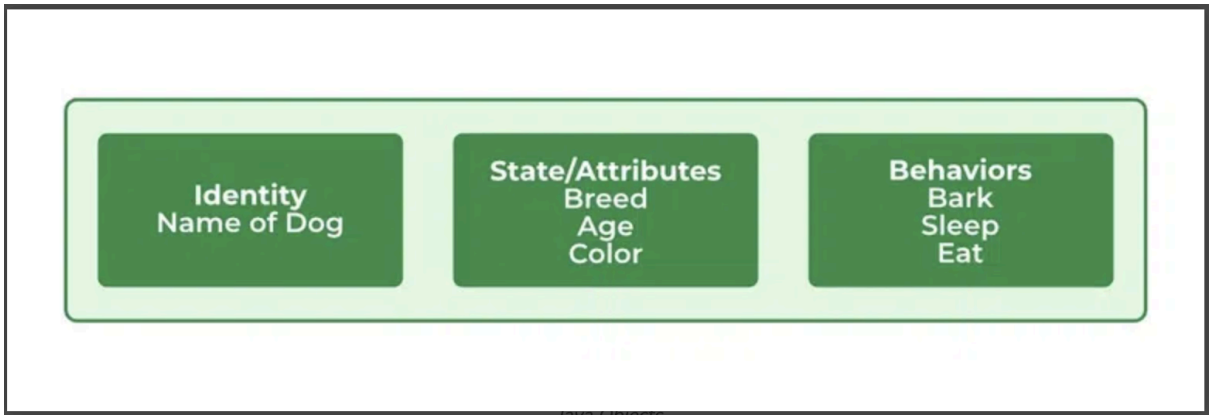
```
// Creation of Object Using new Instance
class Main {
    String n = "Pathshala360";

    public static void main(String[] args) {

        try {
            // Correcting the class name to match "Main"
            Main<?> c = Class.forName("Main");
            // Creating an object of the main class using reflection
            Main o = (Main) c.getDeclaredConstructor().newInstance();
            // Print and display
            System.out.println(o.n);
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        catch (InstantiationException e) {
            e.printStackTrace();
        }
        catch (IllegalAccessException e) {
            e.printStackTrace();
        }
        catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

Output
Pathshala360
```

## Java Objects

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of:

- **State:** It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object with other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

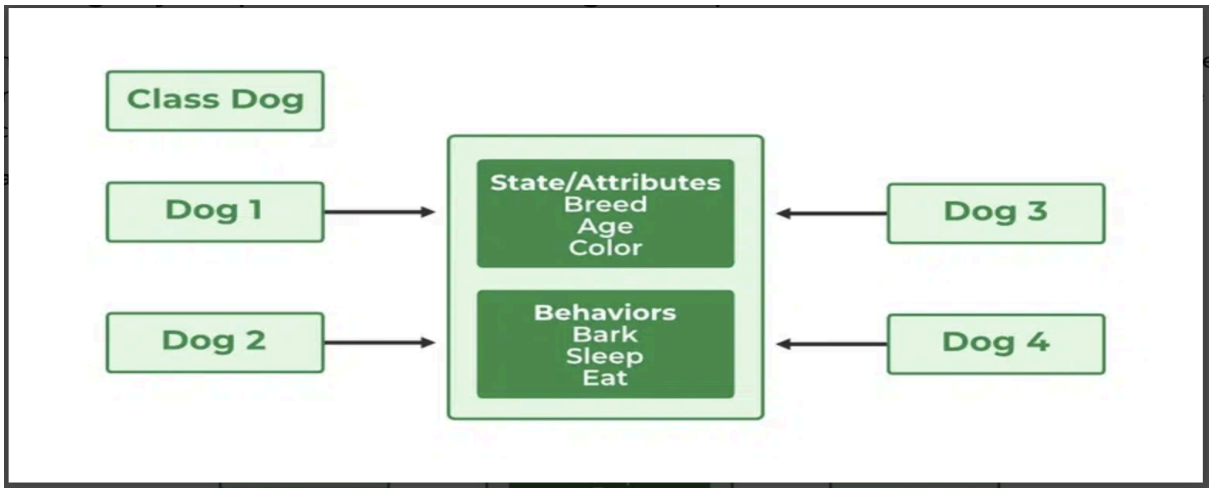
**Example of an object:** Dog




**Declaring Objects (Also called instantiating a Class)**

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

**Example:**



**Initializing a Java Object**

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

**Example:**

```
public class Dog {
    String name;
    String breed;
    int age;
    String color;

    // Constructor Declaration of Class
    public Dog(String name, String breed, int age, String color){
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }

    public String getName() {
      return this.name;
    }

    public String getBreed() {
      return this.breed;
    }

    public int getAge() {
      return this.age;
    }

    public String getColor() {
      return this.color;
    }

    @Override
    public String toString() {
        return ("Name is: " + this.getName()
                + "\nBreed, age, and color are: "
                + this.getBreed() + "," + this.getAge()
                + "," + this.getColor());
    }
```

```
public static void main(String[] args) {
        Dog tuffy = new Dog("tuffy", "papillon", 5, "white");
        System.out.println(tuffy.toString());
    }
}

Output
Name is: tuffy
Breed, age, and color are: papillon,5,white
```

**Initialize Object by using Method/Function**

// Java Program to initialize Java Object by using method/function

```
public class Main {
    static String name;
    static float price;

    static void set(String n, float p) {
        name = n;
        price = p;
    }

    static void get() {
        System.out.println("Software name is: " + this.name);
        System.out.println("Software price is: " + this.price);
    }

    public static void main(String args[]) {
        Main.set("Visual studio", 0.0f);
        Main.get();
    }
}

Output
Software name is: Visual studio
Software price is: 0.0
```

**Ways to Create an Object of a Class**

There are four ways to create objects in Java. Although the new keyword is the primary way to create an object, the other methods also internally rely on the new keyword to create instances.

**1. Using new Keyword**

It is the most common and general way to create an object in Java.

```
// creating object of class Test
Test t = new Test();
```

**2. Using Reflection (Dynamic Class Loading)**

Reflection is a powerful feature in Java that allows a program to inspect and modify its own structure and behavior at runtime.

```
// Assume Student class exists (or show its definition)
class Student {
    public Student() { // Ensure a public no-arg constructor exists
    }
}

public class Main {
    public static void main(String[] args) {
        try {
            Class<?> c = Class.forName("Student");
            Student s2 =
(Student)c.getDeclaredConstructor().newInstance();

            System.out.println("Object created: " + s2);
        } catch (ClassNotFoundException e) {
            System.err.println("Class not found!");
        } catch (NoSuchMethodException e) {
            System.err.println("No default constructor!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

**Note:** Reflection is used in frameworks like Spring for dependency injection.

### 3. Using clone() method

The clone() method is present in the Object class. It creates and returns a copy of the object.

```
// creating object of class Test
Test t1 = new Test();
 // creating clone of above object
 Test t2 = (Test)t1.clone();
```

**Example:**

```
// Creation of Object Using clone() method
class Main implements Cloneable {

    // Method 1
    @Override
    protected Object clone() throws CloneNotSupportedException
    {
        // Super() keyword refers to parent class
        return super.clone();
    }

    String name = "Pathshala360";

    public static void main(String[] args) {
        Main o1 = new Main();

        // Try block to check for exceptions
        try {
            Main o2 = (Main) o1.clone();
            System.out.println(o2.name);
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}

Output
Pathshala360
```

**Creating Multiple Objects by one type only (A good practice)**
In real-time, we need different objects of a class in different methods. Creating a number of references for storing them is not a good practice and therefore we declare a static reference variable and use it whenever required. In this case, the wastage of memory is less. The objects that are not referenced anymore will be destroyed by the Garbage Collector of Java.

**Example:**

```
Test test = new Test();
test = new Test();
```

In the inheritance system, we use a parent class reference variable to store a subclass object. In this case, we can switch into different subclass objects using the same referenced variable.

**Example:**

```
class Animal {}

class Dog extends Animal {}

class Cat extends Animal {}

public class Test {
   // using Dog object
   Animal obj = new Dog();
   // using Cat object
   obj = new Cat();
}
```

**Anonymous Objects in Java**

Anonymous objects are objects that are instantiated without storing their reference in a variable. They are used for one-time operations (e.g., method calls) and are discarded immediately after use.

- No reference variable: Cannot reuse the object.
- Created & used instantly: Saves memory for short-lived tasks.
- Common in event handling (e.g., button clicks).

**Example:**

```
// Creating an anonymous object of a custom class and calling a method
new MyClass().doSomething();

// Passing an anonymous object as an argument to a method
someMethod(new AnotherClass());

// Using an anonymous object in a print statement
System.out.println("Area: " + new Circle(5).calculateArea());
```

## Java Interfaces

An Interface in Java programming language is defined as an abstract type used to specify the behaviour of a class. An interface in Java is a blueprint of a behaviour. A Java interface contains static constants and abstract methods.

**Key Properties of Interface**:

The interface in Java is a mechanism to achieve abstraction.

- By default, variables in an interface are public, static, and final.
- It is used to achieve abstraction and multiple inheritance in Java.
- It supports loose coupling (classes depend on behavior, not implementation).
- In other words, interfaces primarily define methods that other classes must implement.
- An interface in Java defines a set of behaviours that a class can implement, usually representing an IS-A relationship, but not always in every scenario.

**Example:** This example demonstrates how an interface in Java defines constants and abstract methods, which are implemented by a class.

```
import java.io.*;
interface testInterface {
    final int a = 10;
    void display();
}

// Class implementing interface
class TestClass implements testInterface {
    // Implementing the capabilities of Interface
    public void display(){
      System.out.println("Test");
    }
}

class Main {
    public static void main(String[] args){
        TestClass t = new TestClass();
        t.display();
        System.out.println(t.a);
    }

}

Output
Test
10
```

**Note:** In Java, the abstract keyword applies only to classes and methods, indicating that they cannot be instantiated directly and must be implemented. When we decide on a type of entity by its behaviour and not via attribute we should define it as an interface.

**Syntax**:

```
interface InterfaceName {
    // Constant fields (public static final by default)
    int CONSTANT = 10;

    // Abstract method (public abstract by default)
    void methodName();

    // Default method (JDK 8+)
    default void defaultMethod() {
        System.out.println("Default implementation");
    }

    // Static method (JDK 8+)
    static void staticMethod() {
        System.out.println("Static method in interface");
    }

    // Private method (JDK 9+)
    private void privateMethod() {
        System.out.println("Private helper method");
    }

}
```

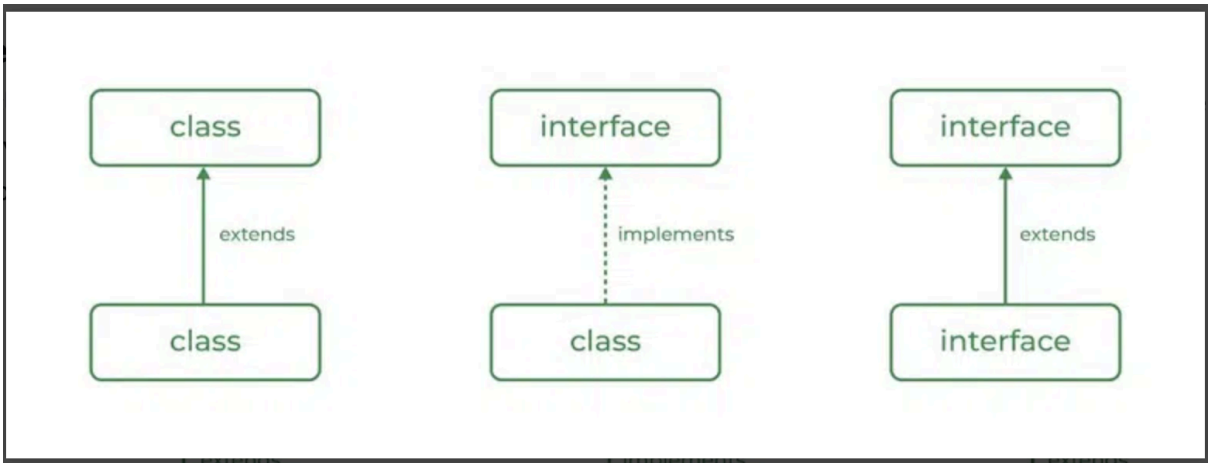
**Note:**

- Private methods can only be called inside default or static methods in the interface, not by implementing classes
- Static methods are also accessible via the 3 interface itself not through objects

To declare an interface, use the interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static, and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement the interface, use the implements keyword.

**Relationship Between Class and Interface**

A class can extend another class, and similarly, an interface can extend another interface. However, only a class can implement an interface, and the reverse (an interface implementing a class) is not allowed.




**Difference Between Class and Interface**

Although Class and Interface seem the same there are certain differences between Classes and Interface. The major differences between a class and an interface are mentioned below:

| Features | Class | Interface |
|---|---|---|
| Instantiation | In class, we can create an object. | In an interface, we can't create an object |
| Variables | Class can have instance variables | Variables are public static final (constants only). |
| Methods | Class can have concrete methods | In an interface, methods are abstract by default |
| Inheritance | It supports single inheritance | Supports multiple inheritance |
| Constructors | Can have constructors. | No constructors allowed. |
| Access Modifiers | Supports private, protected, public, default. | In an interface, all members are public by default |
| Keyword | Defined using class. | Defined using interface. |
| Default Methods | It does not support default methods | It supports default methods(JDK 8+) |

| | | |
|---|---|---|
| Static Methods | Can have static methods. | Supports static methods (JDK 8+) |
| Private Methods | Can have private methods. | Supports private methods (JDK 9+). |
| Main Method | Can have main() for execution. | Can have main() (since JDK 8, as static methods are allowed). |

**When to Use Class and Interface?**

- Use a Class when:
  - Use a class when you need to represent a real-world entity with attributes (fields) and behaviors (methods).
  - Use a class when you need to create objects that hold state and perform actions
  - Classes are used for defining templates for objects with specific functionality and properties.
- Use a Interface when:
  - Use an interface when you need to define a contract for behavior that multiple classes can implement.
  - Interface is ideal for achieving abstraction and multiple inheritance.

**Implementation:** To implement an interface, we use the keyword implements

Let's consider the example of Vehicles like bicycles, cars, and bikes sharing common functionalities, which can be defined in an interface, allowing each class (e.g., Bicycle, Car, Bike) to implement them in its own way. This approach ensures code reusability, scalability, and consistency across different vehicle types.

**Example:** This example demonstrates how an interface can be used to define common behavior for different classes (Bicycle and Bike) that implement the interface.

```
import java.io.*;

interface Vehicle {
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}
```

```java
// Class implementing vehicle interface
class Bicycle implements Vehicle {
    int speed;
    int gear;

    // Change gear
    @Override
    public void changeGear(int newGear){
        gear = newGear;
    }

    // Increase speed
    @Override
    public void speedUp(int increment){
        speed = speed + increment;
    }

    // Decrease speed
    @Override
    public void applyBrakes(int decrement){
        speed = speed - decrement;
    }

    public void printStates() {
        System.out.println("speed: " + speed
            + " gear: " + gear);
    }
}

// Class implementing vehicle interface
class Bike implements Vehicle {
    int speed;
    int gear;

    // Change gear
    @Override
    public void changeGear(int newGear){
        gear = newGear;
    }

    // Increase speed
    @Override
    public void speedUp(int increment){
        speed = speed + increment;
    }
```

```
    // Decrease speed
    @Override
    public void applyBrakes(int decrement){
        speed = speed - decrement;
    }

    public void printStates() {
        System.out.println("speed: " + speed
            + " gear: " + gear);
    }

}

class Main
{
    public static void main (String[] args)
    {

        // Instance of Bicycle(Object)
        Bicycle bicycle = new Bicycle();

        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);

        System.out.print("Bicycle present state : ");
        bicycle.printStates();

        // Instance of Bike (Object)
        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);

        System.out.print("Bike present state : ");
        bike.printStates();
    }
}

Output
Bicycle present state : speed: 2 gear: 2
Bike present state : speed: 1 gear: 1
```

# Java Constructors

In Java, constructors play an important role in object creation. A constructor is a special block of code that is called when an object is created. Its main job is to initialize the object, to set up its internal state, or to assign default values to its attributes. This process happens automatically when we use the "new" keyword to create an object.

**Characteristics of Constructors:**

- **Same Name as the Class:** A constructor has the same name as the class in which it is defined.
- **No Return Type:** Constructors do not have any return type, not even void. The main purpose of a constructor is to initialize the object, not to return a value.
- **Automatically Called on Object Creation:** When an object of a class is created, the constructor is called automatically to initialize the object's attributes.
- **Used to Set Initial Values for Object Attributes:** Constructors are primarily used to set the initial state or values of an object's attributes when it is created.

Now, let us look at a simple example to understand how a constructor works in Java.

**Example:** This program demonstrates how a constructor is automatically called when an object is created in Java.
// Java Program to demonstrate Constructor usage

```
import java.io.*;

class Main {

    Main() {
        super();
        System.out.println("Constructor Called");
    }

    public static void main(String[] args) {
        Main main = new Main();
    }
}

Output
Constructor Called
```

**Note:** It is not necessary to write a constructor for a class. It is because the Java compiler creates a default constructor (constructor with no arguments) if your class doesn't have any.

**Constructor vs Method in Java**
The below table demonstrates the key difference between Java Constructor and Java Methods.

| Features | Constructor | Method |
|---|---|---|
| Name | Constructors must have the same name as the class name | Methods can have any valid name |
| Return Type | Constructors do not return any type | Methods have the return type or void if it does not return any value. |
| Invocation | Constructors are called automatically with new keyword | Methods are called explicitly |
| Purpose | Constructors are used to initialize objects | Methods are used to perform operations |

Now let us come up with the syntax for the constructor being invoked at the time of object or instance creation.

```
class Test {
......

    // A Constructor

    Test() {

    }
.......
}

// We can create an object of the above class using the below statement.
This statement calls the above constructor.
Test obj = new Test();
```

The first line of a constructor is a call to super() or this(), (a call to a constructor of a superclass or an overloaded constructor), if you don't type in the call to super in your constructor the compiler will provide you with a non-argument call

to super at the first line of your code, the super constructor must be called to create an object:

**Note:** If you think your class is not a subclass it actually is, every class in Java is the subclass of a class object even if you don't say extends object in your class definition.

**Why Do We Need Constructors in Java**
Constructors play a very important role, it ensures that an object is properly initialized before use.

**What happens when we don't use constructors:**

- Objects might have undefined or default values.
- Extra initialization methods would be required.
- Risk of improper object state

Think of a Box. If we talk about a box class then it will have some class variables (say length, breadth, and height). But when it comes to creating its object (i.e Box will now exist in the computer's memory), then can a box be there with no value defined for its dimensions? The answer is No.
So, constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

**When Java Constructor is Called?**
Each time an object is created using a new() keyword, at least one constructor (it could be the default constructor) is invoked to assign initial values to the data members of the same class. Rules for writing constructors are as follows:

- The constructor of a class must have the same name as the class name in which it resides.
- A constructor in Java can not be abstract, final, static, or Synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

So, we have learned constructors are used to initialize the object's state. Like methods , a constructor also contains a collection of statements (i.e. instructions) that are executed at the time of object creation.

**Types of Constructors in Java**
Now is the correct time to discuss the types of the constructor, so primarily there are three types of constructors in Java are mentioned below:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor

**1. Default Constructor in Java**

A constructor that has no parameters is known as the default constructor. A default constructor is invisible. And if we write a constructor with no arguments, the compiler does not create a default constructor. Once you define a constructor (with or without parameters), the compiler no longer provides the default constructor. Defining a parameterized constructor does not automatically create a no-argument constructor, we must explicitly define it if needed. The default constructor can be implicit or explicit.

- **Implicit Default Constructor:** If no constructor is defined in a class, the Java compiler automatically provides a default constructor. This constructor doesn't take any parameters and initializes the object with default values, such as 0 for numbers, null for objects.
- **Explicit Default Constructor:** If we define a constructor that takes no parameters, it's called an explicit default constructor. This constructor replaces the one the compiler would normally create automatically. Once you define any constructor (with or without parameters), the compiler no longer provides the default constructor for you.

**Example:** This program demonstrates the use of a default constructor, which is automatically called when an object is created.

```
// Java Program to demonstrate Default Constructor
import java.io.*;

class Main {

    // Default Constructor
    Main() {
        System.out.println("Default constructor");

    }

    public static void main(String[] args) {
        Main hello = new Main();
    }
}

Output
Default constructor
```

**Note:** Default constructor provides the default values to the object like 0, null, false etc. depending on the type.

## 2. Parameterized Constructor in Java

A constructor that has parameters is known as a parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

**Example:** This program demonstrates the use of a parameterized constructor to initialize an object's attributes with specific values.

```
// Java Program for Parameterized Constructor
import java.io.*;

class Main {
    String name;
    int id;

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

class Test {
    public static void main(String[] args) {
        // This would invoke the parameterized constructor
        Main main = new Main("Test", 68);
        System.out.println("Name: " + main.name
                           + " and Id: " + main.id);
    }
}

Output
Name: Test and Id: 68
```

## 3. Copy Constructor in Java

Unlike other constructors, a copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

**Note:** Java does not provide a built-in copy constructor like C++. We can create our own by writing a constructor that takes an object of the same class as a parameter and copies its fields.

Example: This example demonstrates how a copy constructor can be used to create a new object by copying the values of another object's attributes.

```
// Java Program for Copy Constructor
import java.io.*;

class Student {
    String name;
    int id;

    // Parameterized Constructor
    Student(String name, int id){
        this.name = name;
        this.id = id;
    }

    // Copy Constructor
    Student(Student obj2)
    {
        this.name = obj2.name;
        this.id = obj2.id;
    }
}

class Test {
    public static void main(String[] args) {
        // This would invoke the parameterized constructor
        System.out.println("First Object");
        Student student1 = new Student("Test", 68);
        System.out.println("Name: " + student1.name
                           + " and Id: " + student1.id);

        System.out.println();

        // This would invoke the copy constructor
        Student student2 = new Student(student1);
        System.out.println("Copy Constructor used Second Object");
        System.out.println("Name: " + student2.name
                           + " and Id: " + student2.id);
    }
}

Output
First Object
Name: Test and Id: 68
Copy Constructor used Second Object
Name: Test and Id: 68
```

**Constructor Overloading**

This is a key concept in OOPs related to constructors is constructor overloading. This allows us to create multiple constructors in the same class with different parameter lists.

**Example:** This example demonstrates constructor overloading, where multiple constructors perform the same task (initializing an object) with different types or numbers of arguments.

```
import java.io.*;

class Main {
    // constructor with one argument
    Main(String name){
        System.out.println("Constructor with one "
                           + "argument - String: " + name);
    }
    // constructor with two arguments
    Main(String name, int age) {
        System.out.println("Constructor with two arguments: "
            + " String and Integer: " + name + " " + age);
    }
    // Constructor with one argument but with different type than
previous
    Main(long id){
        System.out.println("Constructor with one argument: "
            + "Long: " + id);
    }
}

class Test {
    public static void main(String[] args) {
        Main main = new Main("Test");
        Main main1 = new Main("Test1", 28);
        Main main2 = new Main(325614567);
    }
}

Output
Constructor with one argument - String: Test
Constructor with two arguments:  String and Integer: Test 28
Constructor with one argument: Long: 325614567
```

**Common Mistakes to Avoid**

Some common mistakes to avoid when working with constructors in Java are listed below:

- **Forgetting super() in Child Classes:** Always call the parent constructor (super()) if the parent class has no default constructor, or it will lead to compilation errors.
- **Excessive Work in Constructors:** Keep constructors simple and focused on initialization, avoiding heavy logic that slows down object creation.
- **Not Handling Null Checks:** Always validate parameters to avoid NullPointerException when constructing objects.

# Object Class in Java

Object class in Java 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 the Java Object class and if it extends another class then it is indirectly derived. The Object class provides several methods such as toString(),equals(), hashCode(), and many others. Hence, the Object class acts as a root of the inheritance hierarchy in any Java Program.

**Example:** Here, we will use the toString() and hashCode() methods to provide a custom string representation for a class.

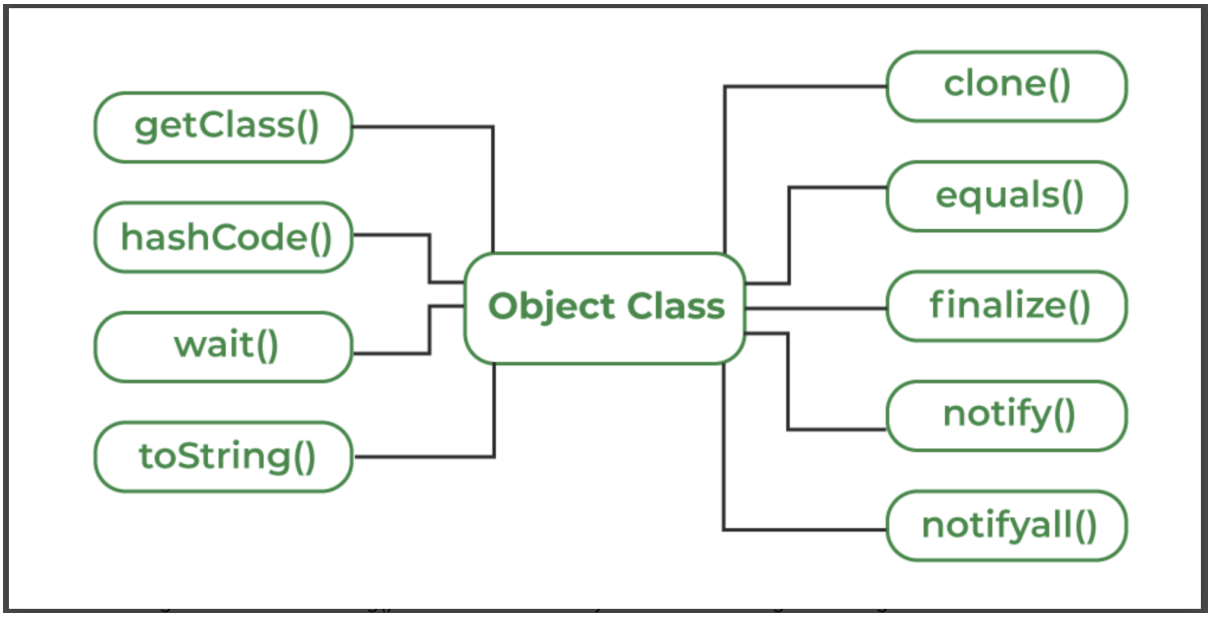
```
// Java Code to demonstrate Object class
class Person {
    String name;

    // Constructor
    public Person(String name) {
        this.name = name;
    }

    // Override toString() for a custom string representation
    @Override
    public String toString() {
        return "Person{name:'" + n + "'}";
    }

    public static void main(String[] args) {
        Person p = new Person("Test");

        // Custom string representation
        System.out.println(p.toString());

        // Default hashcode value
        System.out.println(p.hashCode());
    }
}

Output
Person{name:'Test'}
321001045
```

**Explanation:** In the above example, we override the toString() method to provide a custom string representation of the Person class and use the hashCode() method to display the default hashcode value of the object.

**Object Class Methods**

The Object class provides multiple methods which are as follows:

- toString() method
- hashCode() method
- equals(Object obj) method
- finalize() method
- getClass() method
- clone() method




**1. toString() Method**

The toString() provides a String representation of an object and is used to convert an object to a String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object.

**Note:** Whenever we try to print any Object reference, then internally toString() method is called.

**Example:**

```
public class Student {
    public String toString() {
        return "Student object";
    }
}
```

**Explanation:** The toString() method is overridden to return a custom string representation of the Student object.

### 2. hashCode() Method

For every object, JVM generates a unique number which is a hashcode. It returns distinct integers for distinct objects. A common misconception about this method is that the hashCode() method returns the address of the object, which is not correct. It converts the internal address of the object to an integer by using an algorithm. The hashCode() method is native because in Java it is impossible to find the address of an object, so it uses native languages like C/C++ to find the address of the object.

**Use of hashCode() method:**

It returns a hash value that is used to search objects in a collection. JVM(Java Virtual Machine) uses the hashcode method while saving objects into hashing-related data structures like HashSet, HashMap, Hashtable, etc. The main advantage of saving objects based on hash code is that searching becomes easy.

**Note:** Override of hashCode() method needs to be done such that for every object we generate a unique number. For example, for a Student class, we can return the roll no. of a student from the hashCode() method as it is unique.

**Example:**

```
public class Student {
    int roll;

    @Override
    public int hashCode() {
        return roll;
    }
}
```

**Explanation:** The hashCode() method is overridden to return a custom hash value based on the roll of the Student object.

### 3. equals(Object obj) Method

The equals() method compares the given object with the current object. It is recommended to override this method to define custom equality conditions.

**Note:** It is generally necessary to override the hashCode() method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

**Example:**

```
public class Student {
    int roll;

    @Override
    public boolean equals(Object o) {
        if (o instanceof Student) {
            return this.roll == ((Student) o).roll;
        }
        return false;
    }
}
```

**Explanation:** The equals() method is overridden to compare roll between two Student objects.

**4. getClass() method**

The getClass() method returns the class object of "this" object and is used to get the actual runtime class of the object. It can also be used to get metadata of this class. The returned Class object is the object that is locked by static synchronized methods of the represented class. As it is final so we don't override it.

**Example:**

```
// Demonstrate working of getClass()
public class Test {
    public static void main(String[] args) {
        Object o = new String("Test");
        Class c = o.getClass();
        System.out.println("Class of Object o is: " + c.getName());
    }
}

Output
Class of Object o is: java.lang.String
```

**Explanation:** The getClass() method is used to print the runtime class of the "o" object.

**Note**: After loading a .class file, JVM will create an object of the type java.lang.Class in the Heap area. We can use this class object to get Class level information. It is widely used in Reflection

**5. finalize() method**

The finalize() method is called just before an object is garbage collected. It is called the Garbage Collector on an object when the garbage collector determines that there are no more references to the object. We should override the finalize() method to dispose of system resources, perform clean-up activities and minimize memory leaks. For example, before destroying the Servlet objects web container, always called the finalize method to perform clean-up activities of the session.

**Note:** The finalize method is called just once on an object even though that object is eligible for garbage collection multiple times.

**Example:**

```
// Demonstrate working of finalize()
public class Test {
    public static void main(String[] args) {

        Test t = new Test();
        System.out.println(t.hashCode());

        t = null;
        // calling garbage collector
        System.gc();
        System.out.println("end");
    }

    @Override
    protected void finalize(){
        System.out.println("finalize method called");
    }
}

Output
1510467688
end
finalize method called
```

**Explanation:** The finalize() method is called just before the object is garbage collected.

**6. clone() method**

The clone() method creates and returns a new object that is a copy of the current object.

**Example:**

```
public class Book implements Cloneable {
    private String t;

    public Book(String t) {
        this.t = t;
    }

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

**Explanation:** The clone() method is overridden to return a cloned copy of the Book object.

# Abstraction in Java

Abstraction in Java is the process of hiding the implementation details and only showing the essential details or features to the user. It allows to focus on what an object does rather than how it does it. The unnecessary details are not displayed to the user.

**Key features of abstraction:**

- Abstraction hides the complex details and shows only essential features.
- Abstract classes may have methods without implementation and must be implemented by subclasses.
- By abstracting functionality, changes in the implementation do not affect the code that depends on the abstraction.

**How to Achieve Abstraction in Java?**

Java provides two ways to implement abstraction, which are listed below:

- Abstract Classes (Partial Abstraction)
- Interface (100% Abstraction)

**Real-Life Example of Abstraction:**

The television remote control is the best example of abstraction. It simplifies the interaction with a TV by hiding all the complex technology. We don't need to understand how the tv internally works, we just need to press the button to change the channel or adjust the volume.

**Example:**

```
// Working of Abstraction in Java
abstract class TVRemote {
    abstract void turnOn();
    abstract void turnOff();
}

// Concrete class implementing the abstract methods
class TVRemoteImpl extends TVRemote {
    @Override
    void turnOn() {
        System.out.println("TV is turned ON.");
    }

    @Override
    void turnOff() {
        System.out.println("TV is turned OFF.");
    }
}
```

```
// Main class to demonstrate abstraction
public class Main {
    public static void main(String[] args) {
        TVRemote remote = new TVRemoteImpl();
        remote.turnOn();
        remote.turnOff();
    }
}

Output
TV is turned ON.
TV is turned OFF.
```

**Explanation:** In the above example, the "TVRemote" abstract class hides implementation details and defines the essential methods turnOn and turnOff. The TVRemoteImpl class provides specific implementations for these methods. The main class demonstrates how the user interacts with the abstraction without needing to know the internal details.

**Abstract Classes and Abstract Methods**

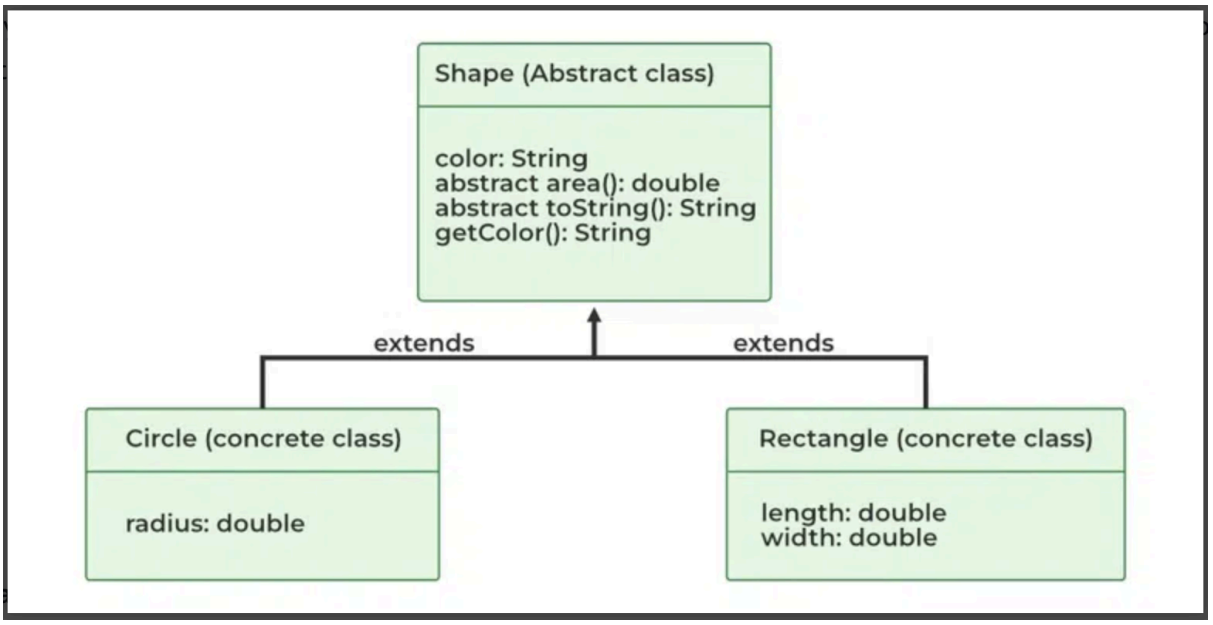
- An abstract class is a class that is declared with an abstract keyword.
- An abstract method is a method that is declared without implementation.
- An abstract class may have both abstract methods (methods without implementation) and concrete methods (methods with implementation).
- An abstract method must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.
- Any class that contains one or more abstract methods must also be declared with an abstract keyword.
- There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.
- An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

**Algorithm to Implement Abstraction**

- Determine the classes or interfaces that will be part of the abstraction.
- Create an abstract class or interface that defines the common behaviors and properties of these classes.
- Define abstract methods within the abstract class or interface that do not have any implementation details.
- Implement concrete classes that extend the abstract class or implement the interface.
- Override the abstract methods in the concrete classes to provide their specific implementations.
- Use the concrete classes to contain the program logic.

**When to Use Abstract Classes and Abstract Methods?**
There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

Consider a classic "shape" example, perhaps used in a computer-aided design system or game simulation. The base type is "shape" and each shape has a color, size, and so on. From this, specific types of shapes are derived(inherited)-circle, square, triangle, and so on — each of which may have additional characteristics and behaviors. For example, certain shapes can be flipped. Some behaviors may be different, such as when you want to calculate the area of a shape. The shape hierarchy shows both the similarities that all shapes share and the differences that make each one unique.



**Example:**

```
// Java program to illustrate the concept of Abstraction
abstract class Shape {
    String color;

    // these are abstract methods
    abstract double area();
    public abstract String toString();

    // abstract class can have the constructor
    public Shape(String color) {
        System.out.println("Shape constructor called");
        this.color = color;
    }

    // this is a concrete method
    public String getColor() { return color; }
}

class Circle extends Shape {
    double radius;

    public Circle(String color, double radius){
        // calling Shape constructor
        super(color);
        System.out.println("Circle constructor called");
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * Math.pow(radius, 2);
    }

    @Override
    public String toString(){
        return "Circle color is " + super.getColor()
            + "and area is : " + area();
    }
}

class Rectangle extends Shape {
    double length;
    double width;
```

```
    public Rectangle(String color, double length, double width) {
        // calling Shape constructor
        super(color);
        System.out.println("Rectangle constructor called");
        this.length = length;
        this.width = width;
    }

    @Override
    double area() { return length * width; }

    @Override
    public String toString() {
        return "Rectangle color is " + super.getColor()
            + "and area is : " + area();
    }
}

public class Test {
    public static void main(String[] args)
    {
        Shape s1 = new Circle("Red", 2.2);
        Shape s2 = new Rectangle("Yellow", 2, 4);

        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}

Output
Shape constructor called
Circle constructor called
Shape constructor called
Rectangle constructor called
Circle color is Redand area is : 15.205308443374602
Rectangle color is Yellowand area is : 8.0
```

**Example:** Let's see another example to understand abstraction in Java.

```
// Abstract Class declared
abstract class Animal {
    private String name;

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

    public abstract void makeSound();

    public String getName() {
      return name;
    }
}

// Abstract class
class Dog extends Animal {
    public Dog(String name) {
      super(name);
    }

    public void makeSound() {
        System.out.println(getName() + " barks");
    }
}

// Abstract class
class Cat extends Animal {
    public Cat(String name) {
      super(name);
    }

    public void makeSound() {
        System.out.println(getName() + " meows");
    }
}

// Driver Class
public class Main {

    // Main Function
    public static void main(String[] args) {
        Animal myDog = new Dog("ABC");
        Animal myCat = new Cat("XYZ");
```

```
        myDog.makeSound();
        myCat.makeSound();
    }
}

Output
ABC barks
XYZ meows
```

**Interface**

Interfaces are another method of implementing abstraction in Java. The key difference is that, by using interfaces, we can achieve 100% abstraction in Java classes. In Java or any other language, interfaces include both methods and variables but lack a method body. Apart from abstraction, interfaces can also be used to implement inheritance in Java.

**Implementation:** To implement an interface we use the keyword “implements” with class.

**Example:**

```
// Define an interface named Shape
interface Shape {
    double calculateArea(); // Abstract method for calculating the area
}

// Implement the interface in a class named Circle
class Circle implements Shape {
    private double r;    // radius

    // Constructor for Circle
    public Circle(double r) {
      this.r = r;
    }

    // Implementing the abstract method
    // from the Shape interface
    public double calculateArea() {
        return Math.PI * r * r;
    }
}
```

```
// Implement the interface in a class named Rectangle
class Rectangle implements Shape {
    private double length;
    private double width;

    // Constructor for Rectangle
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    // Implementing the abstract method from the Shape interface
    public double calculateArea() {
      return length * width;
    }
}

// Main class to test the program
public class Main {
    public static void main(String[] args) {
        // Creating instances of Circle and Rectangle
        Circle c = new Circle(5.0);
        Rectangle rect = new Rectangle(4.0, 6.0);
        System.out.println("Area of Circle: " + c.calculateArea());
        System.out.println("Area of Rectangle:"+ rect.calculateArea());
    }
}

Output
Area of Circle: 78.53981633974483
Area of Rectangle: 24.0
```

**Advantages of Abstraction**

- Abstraction makes complex systems easier to understand by hiding the implementation details.
- Abstraction keeps different part of the system separated.
- Abstraction maintains code more efficiently.
- Abstraction increases the security by only showing the necessary details to the user.

**Disadvantages of Abstraction**

- It can add unnecessary complexity if overused.
- May reduce flexibility in implementation.
- Makes debugging and understanding the system harder for unfamiliar users.
- Overhead from abstraction layers can affect performance.

# Encapsulation in Java

In Java, encapsulation is one of the concrete concept of Object Oriented Programming (OOP) in which we bind the data members and methods into a single unit. Encapsulation is used to hide the implementation part and show the functionality for better readability and usability. The following are important points about encapsulation.

- **Better Code Management :** We can change data representation and implementation any time without changing the other codes using it if we keep method parameters and return values the same. With encapsulation, we ensure that no other code would have access to implementation details and data members.
- **Simpler Parameter Passing :** When we pass an object to a method, everything (associated data members and methods are passed along). We do not have to pass individual members.
- **getter and setter:** getter (display the data) and setter method ( modify the data) are used to provide the functionality to access and modify the data, and the implementation of this method is hidden from the user. The user can use this method, but cannot access the data directly.

**Example:** Below is a simple example of Encapsulation in Java.

```
// Java program demonstrating Encapsulation
class Programmer {
    private String name;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}

public class Main {
    public static void main(String[] args) {
        Programmer p = new Programmer();
        p.setName("Test");
        System.out.println("Name=> " + p.getName());
    }
}

Output
Name=> Test
```

**Explanation:** In the above example, we use the encapsulation and use getter (getName) and setter (setName) methods which are used to show and modify the private data. This encapsulation mechanism protects the internal state of the Programmer object and allows for better control and flexibility in how the name attribute is accessed and modified.

**Uses of Encapsulation in Java**

Using encapsulation in Java has many benefits:

- **Data Hiding:** The internal data of an object is hidden from the outside world, preventing direct access.
- **Data Integrity:** Only validated or safe values can be assigned to an object's attributes via setter methods.
- **Reusability:** Encapsulated code is more flexible and reusable for future modifications or requirements.
- **Security:** Sensitive data is protected as it cannot be accessed directly.

**Implementation of Java Encapsulation**

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

- In encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of its own class.
- A private class can hide its members or methods from the end user, using abstraction to hide implementation details, by combining data hiding and abstraction.
- Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.
- It is more defined with the setter and getter method.

**Example 1:** Here is another example of encapsulation in Java.

```
// Java program to demonstrate the Encapsulation.
class Person {

    // Encapsulating the name and age
    // only approachable and used using
    // methods defined
    private String name;
    private int age;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public int getAge() { return age; }

    public void setAge(int age) { this.age = age; }
}
```

```
// Driver Class
public class Main {

    // main function
    public static void main(String[] args) {
        // person object created
        Person p = new Person();
        p.setName("Sweta");
        p.setAge(25);

        // Using methods to get the values from the
        // variables
        System.out.println("Name: " + p.getName());
        System.out.println("Age: " + p.getAge());
    }
}

Output
Name: Sweta
Age: 25
```

**Explanation:** Here, the encapsulation is achieved by restricting direct access to the name and age fields of the Person class. These fields are marked as private and can only be accessed or modified through public getter and setter methods (getName(), setName(), getAge(), setAge()). This approach ensures data hiding and maintains control over the values of these fields.

**Example 2:** In this example, we use abstraction to hide the implementation and show the functionality.

```
class Area {
    private int l; // this value stores length
    private int b; // this value stores breadth

    Area(int l, int b){
        this.l = l;
        this.b = b;
    }

    public void getArea() {
        int area = l * b;
        System.out.println("Area: " + area);
    }
}
```

```
public class Main{
    public static void main(String[] args) {
        Area rect = new Area(2, 16);
        rect.getArea();
    }
}

Output
Area: 32
```

**Explanation**: In the above example, Area class and inside the class we create a constructor which takes two parameters l (length) and b (breath) and a getArea() method to calculate the area of the rectangle. This method shows the abstraction where the implementation (Calculating area) is hidden and functionality is showing (Area of rectangle).

**Example 3:** The program to access variables of the class is shown below:

```
// Java program to demonstrate Java encapsulation
class Encapsulate {
    private String name;
    private int roll;
    private int age;

    public int getAge() { return age; }

    public String getName() { return name; }

    public int getRoll() { return roll; }

    public void setAge(int newAge) { age = newAge; }

    public void setName(String newName) { name = newName; }

    public void setRoll(int newRoll) { roll = newRoll; }
}
```

```
// Main Class
public class Main {
    public static void main(String[] args) {
        Encapsulate o = new Encapsulate();

        // setting values of the variables
        o.setName("Test");
        o.setAge(19);
        o.setRoll(51);

        // Displaying values of the variables
        System.out.println("Name: " + o.getName());
        System.out.println("Age: " + o.getAge());
        System.out.println("Roll: " + o.getRoll());
    }
}

Output
Name:Test
Age: 19
Roll: 51
```

**Explanation:** Here, the class Encapsulate keeps its variables private, it simply means that they can not be accessed directly from outside the class. To get the values of these variables we need to use public methods. This way the data can only be accessed or changed through these methods.

**Example 4:** Let us go through another example, to understand the concept better.

```
// Java Program which demonstrate Encapsulation
class Account {
    private long accNo;
    private String name;
    private String email;
    private float amount;

    public long getAccNo() { return accNo; }
    public void setAccNo(long accNo) { this.accNo = accNo; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
    public float getAmount() { return amount; }
    public void setAmount(float amount) { this.amount = amount; }
}
```

```
public class Main {
    public static void main(String[] args) {
        Account acc = new Account();

        acc.setAccNo(90482098491L);
        acc.setName("ABC");
        acc.setEmail("abc@gmail.com");
        acc.setAmount(100000f);

        // Get values using getter methods
        System.out.println("Account Number: " + acc.getAccNo());
        System.out.println("Name: " + acc.getName());
        System.out.println("Email: " + acc.getEmail());
        System.out.println("Amount: " + acc.getAmount());
    }
}

Output
Account Number: 90482098491
Name: ABC
Email: abc@gmail.com
Amount: 100000.0
```

**Explanation:** Here, we are using encapsulation and hiding all the details inside the Account class. We can not access all the details from outside the class, to access all the details we need to create getter and setter methods so that we can easily get and update the details. In the main method, an Account object is created so that values can be set using the setter and get the value using getter.

**Advantages of Encapsulation**

The advantages of encapsulation are listed below:

- **Data Hiding:** Encapsulation is used to hide the details of a class. We do not need to worry about how it works. It just uses the setter methods to give values to the variables.
- **Data Integrity:** With the help of setter and getter methods we can ensure that the correct data is assigned to the variable.
- **Reusability:** Without the help of encapsulation the readability of the code increases and the changing of code becomes easier when we need to add new features in the code.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.
- **Freedom of advantages:** one of the major advantages of encapsulation is that it gives the programmer freedom to implement the details of a system. The only constraint on the programmer is to maintain the abstract interface that outsiders see.

**Disadvantages of Encapsulation**

The disadvantages of encapsulation are listed below:

- Sometimes encapsulation can make the code complex and hard to understand if we do not use it in the right way.
- It can make it more difficult to understand how the program works because some parts of the program are hidden.

# Inheritance in Java

Java Inheritance is a fundamental concept in OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well. Inheritance promotes code reusability, method overriding, and polymorphism, which makes the Java program more modular and efficient.

**Note:** In Java, inheritance is implemented using the extends keyword. The class that inherits is called the subclass (child class), and the class being inherited from is called the superclass (parent class).

**Why Use Inheritance in Java?**

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstraction where we do not have to provide all details, is achieved through inheritance. Abstraction only shows the functionality to the user.
- **that** Organizes classes in a structured manner, improving readability and maintainability.

**Key Terminologies Used in Java Inheritance**

- **Class:** Class is a set of objects that share common characteristics/ behavior and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- **Super Class/Parent Class:** The class whose features are inherited is known as a superclass(or a base class or a parent class).
- **Sub Class/Child Class:** The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Extends Keyword:** This keyword is used to inherit properties from a superclass.
- **Method Overriding:** Redefining a superclass method in a subclass.
- **Reusability**: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

**How Inheritance Works in Java?**

The extends keyword is used for inheritance in Java. It enables the subclass to inherit the fields and methods of the superclass. When a class extends another class, it means it inherits all the non-primitive members (fields and methods) of the parent class, and the subclass can also override or add new functionality to them.

**Note:** The extends keyword establishes an "is-a" relationship between the child class and the parent class. This allows a child class to have all the behavior of the parent class.

**Syntax:**
The syntax for inheritance in Java is listed below:

```
class ChildClass extends ParentClass {
    // Additional fields and methods
}
```

**Inheritance in Java Example**
In the below example of inheritance, class Bicycle is a base class, class MountainBike is a derived class that extends the Bicycle class and class Test is a driver class to run the program.

```
class Bicycle {
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed) {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement){
        speed -= decrement;
    }

    public void speedUp(int increment){
        speed += increment;
    }

    // toString() method to print info of Bicycle
    public String toString() {
        return ("No of gears are " + gear + "\n"
                + "speed of bicycle is " + speed);
    }
}

class MountainBike extends Bicycle {
    public int seatHeight;

    public MountainBike(int gear, int speed, int startHeight) {
```

```
        super(gear, speed);
        seatHeight = startHeight;
    }

    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

    @Override
    public String toString() {
        return (super.toString() + "\nseat height is "
                + seatHeight);
    }
}

public class Test {
    public static void main(String args[]) {
        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());
    }
}

Output
No of gears are 3
speed of bicycle is 100
seat height is 25
```

**Explanation:** In the above example, when an object of MountainBike class is created, a copy of all methods and fields of the superclass acquires memory in this object. That is why by using the object of the subclass we can also access the members of a superclass.

**Note:** During inheritance only the object of the subclass is created, not the superclass. For more, refer to Java Object Creation of Inherited Class.

**Example:** This program demonstrates inheritance in Java, where the Engineer class inherits the salary field from the Employee class and adds its own benefits field.

```
class Employee {
    int salary = 60000;
}

class Engineer extends Employee {
    int benefits = 10000;
}
```

```
// Driver Class
class Test {
    public static void main(String args[]) {
        Engineer E1 = new Engineer();
        System.out.println("Salary : " + E1.salary
                           + "\nBenefits : " + E1.benefits);
    }
}

Output
Salary : 60000
Benefits : 10000
```

**Note:** In practice, inheritance, and polymorphism are used together in Java to achieve fast performance and readability of code.

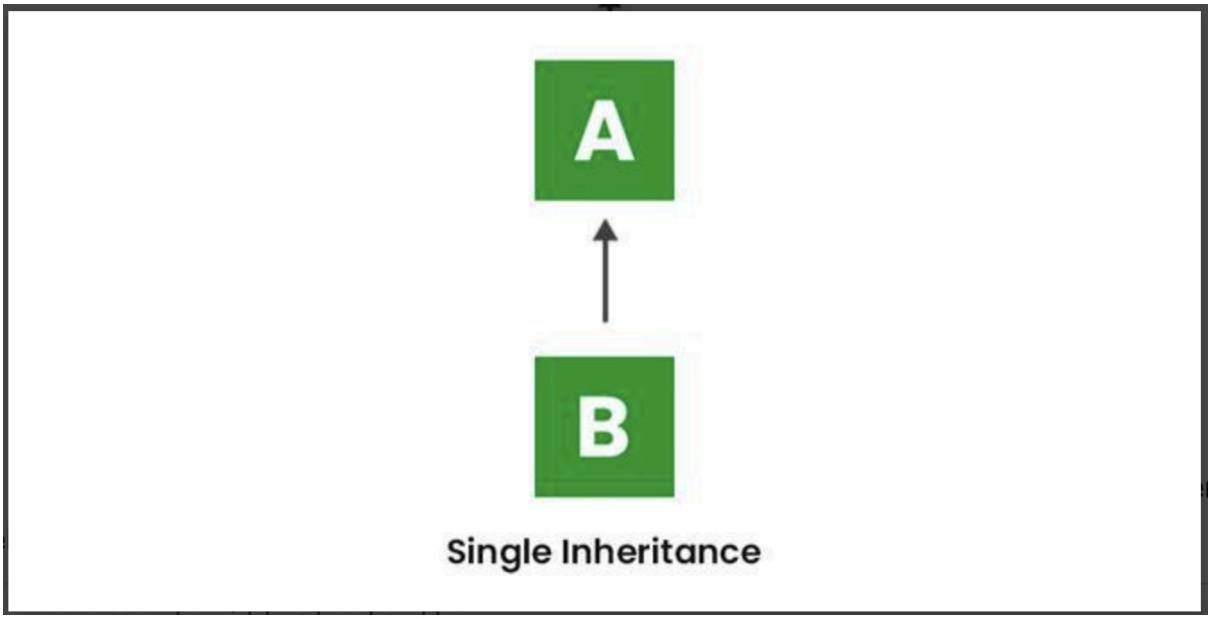
**Types of Inheritance in Java**

Below are the different types of inheritance which are supported by Java.

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance

**1. Single Inheritance**

In single inheritance, a subclass is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes, it is also known as simple inheritance. In the below figure, 'A' is a parent class and 'B' is a child class. The class 'B' inherits all the properties of the class 'A'.

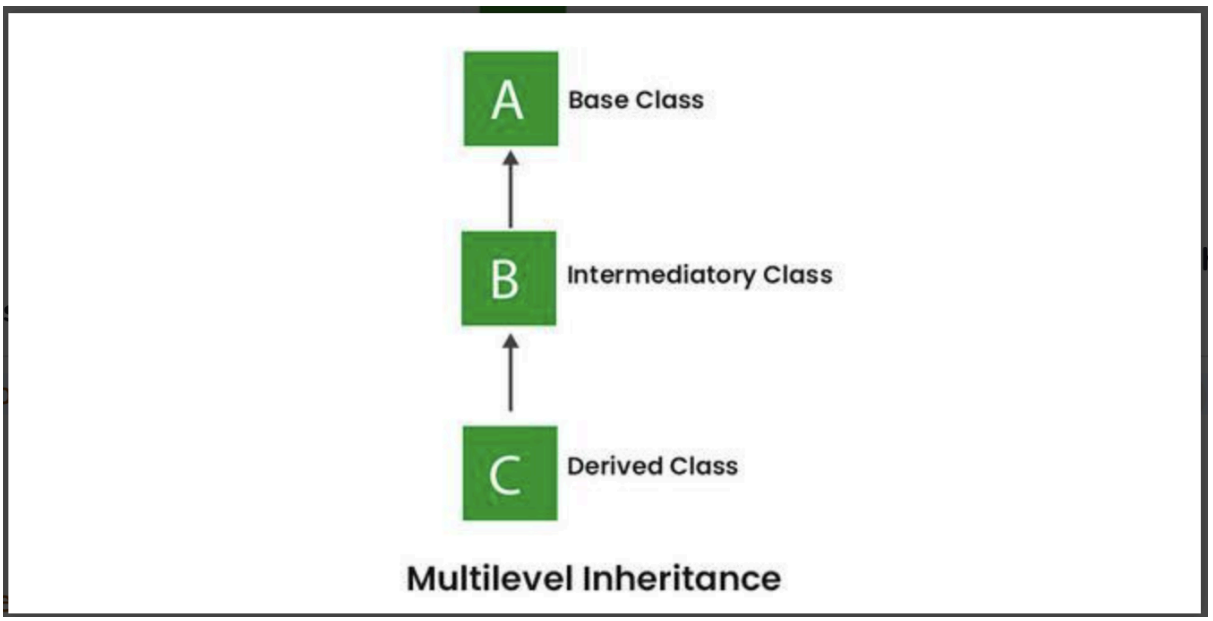


**Example:** This program demonstrates single inheritance in Java, where the Two class inherits the print() method from the One class and adds its own printFor() method.

```
class One {
    public void print() {
        System.out.println("Test");
    }
}

class Two extends One {
    public void printFor() { System.out.println("Purpose"); }
}

public class Main {
    public static void main(String[] args){
        Two two = new Two();
        two.print();
        two.printFor();
    }
}
Output
Test
Purpose
```

**2. Multilevel Inheritance**

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members if they are private.



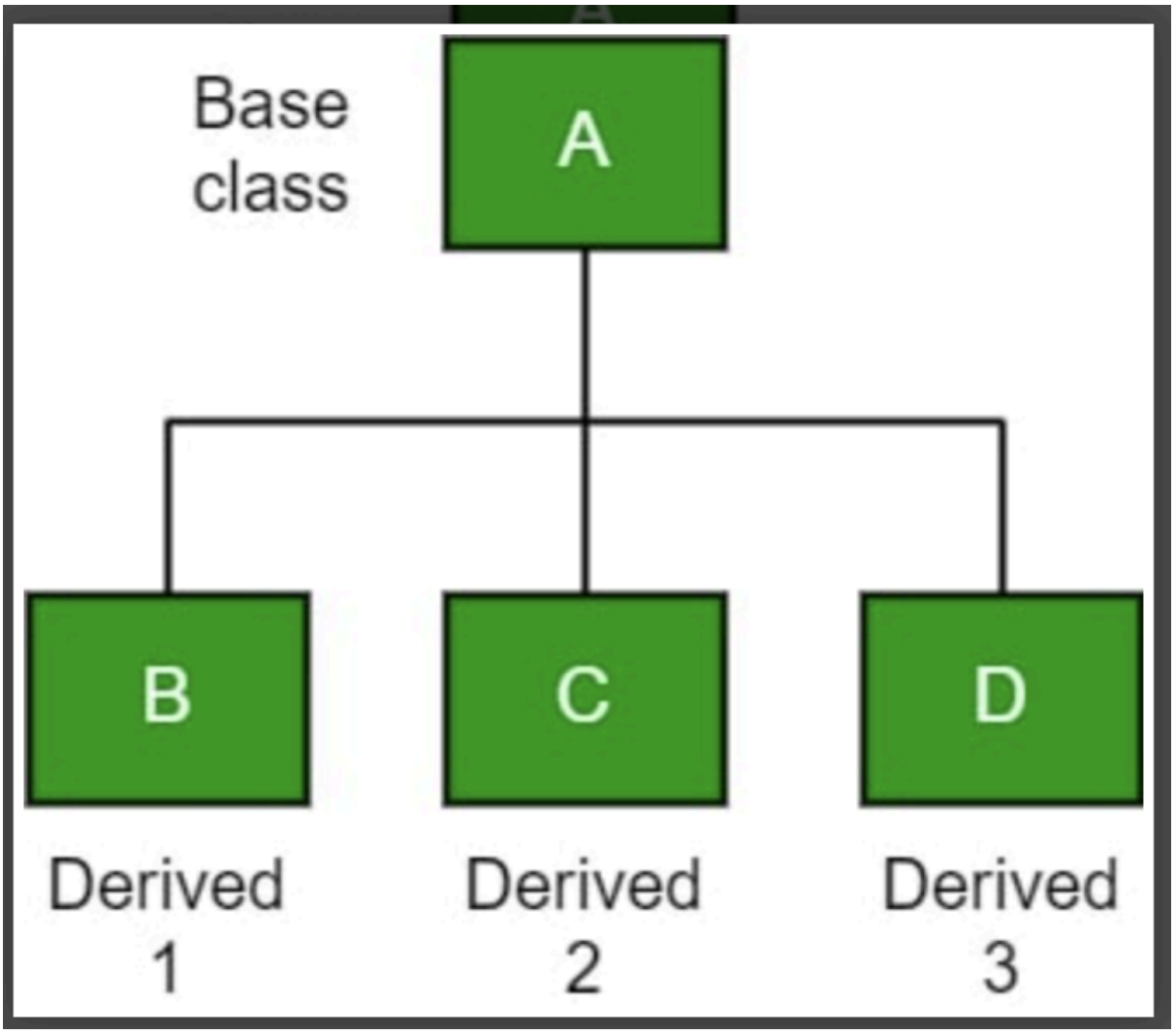

**Example:** This program demonstrates multilevel inheritance in Java, where class Three inherits from class Two, and class Two inherits from class One.

```
class One {
    public void print() {
        System.out.println("Test");
    }
}

class Two extends One {
    public void printFor() {
        System.out.println("Purpose");
    }
}

class Three extends Two {
    public void printLast() {
        System.out.println("Inheritance");
    }
}


public class Main {
    public static void main(String[] args) {
        Three g = new Three();
        g.print();
        g.printFor();
        g.printLast();
    }
}
```

```
Test
Purpose
Inheritance
```

### 3. Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.




**Example:** This example demonstrates Hierarchical inheritance where multiple classes (B, C, D) inherit from a single class (A), allowing them to share the same methods from class A.

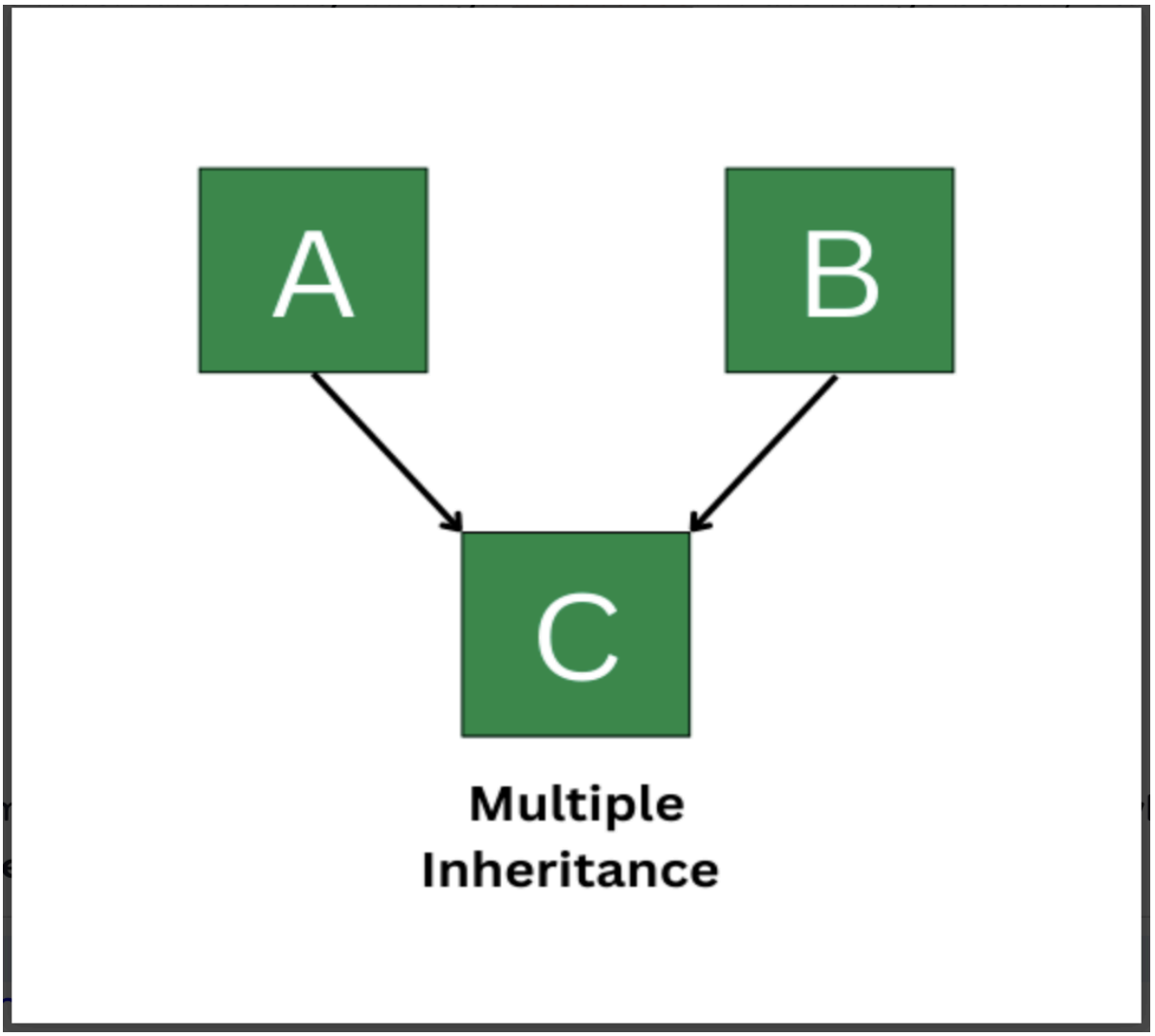
```
class A {
    public void printA() { System.out.println("Class A"); }
}
```

```
class B extends A {
    public void printB() { System.out.println("Class B"); }
}

class C extends A {
    public void printC() { System.out.println("Class C"); }
}

class D extends A {
    public void printD() { System.out.println("Class D"); }
}

public class Test {
    public static void main(String[] args){
        B objB = new B();
        objB.printA();
        objB.printB();

        C objC = new C();
        objC.printA();
        objC.printC();

        D objD = new D();
        objD.printA();
        objD.printD();
    }
}

Output
Class A
Class B
Class A
Class C
Class A
Class D
```

**4. Multiple Inheritance (Through Interfaces)**

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support

multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.




**Example:** This example demonstrates Multiple inheritance through interfaces, where a class can implement multiple interfaces to inherit methods from them.

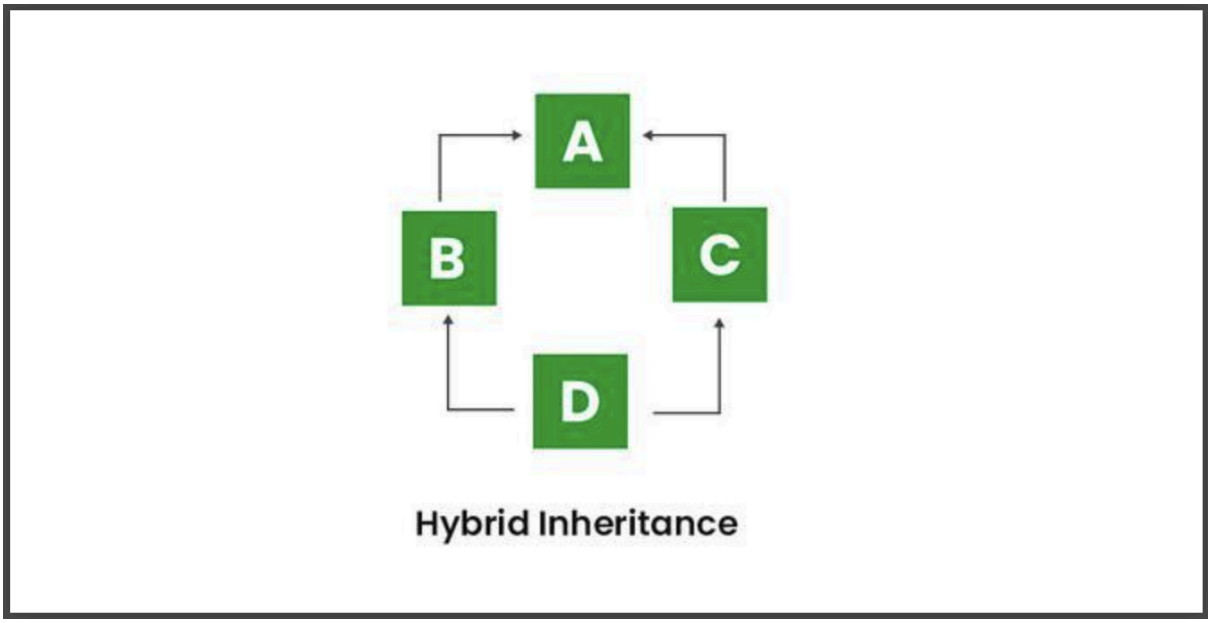
```
// Interface 1 that defines coding behavior
interface Coder {
```

```
    void writeCode();
}

// Interface 2 that defines testing behavior
interface Tester {
    void testCode();
}

// Class implementing both interfaces
class DevOpsEngineer implements Coder, Tester {
    @Override
    public void writeCode() {
        System.out.println("DevOps Engineer writes automation
scripts.");
    }

    @Override
    public void testCode() {
        System.out.println("DevOps Engineer tests deployment
pipelines.");
    }

    public void deploy() {
        System.out.println("DevOps Engineer deploys code to the
cloud.");
    }
}

public class Main {
    public static void main(String[] args) {
        DevOpsEngineer devOps = new DevOpsEngineer();

        devOps.writeCode();
        devOps.testCode();
        devOps.deploy();
    }
}

Output
DevOps Engineer writes automation scripts.
DevOps Engineer tests deployment pipelines.
DevOps Engineer deploys code to the cloud.
```

**5. Hybrid Inheritance**

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritance with classes, hybrid inheritance involving multiple

inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.
It is important to note that Hybrid inheritance does not necessarily require the use of Multiple Inheritance exclusively. It can be achieved through a combination of Multilevel Inheritance and Hierarchical Inheritance with classes, Hierarchical and Single Inheritance with classes. Therefore, it is indeed possible to implement Hybrid inheritance using classes alone, without relying on multiple inheritance types.




**Java IS-A type of Relationship**
IS-A is a way of saying: This object is a type of that object. Let us see how the extended keyword is used to achieve inheritance.

```
public class SolarSystem {
}
public class Earth extends SolarSystem {
}
public class Mars extends SolarSystem {
}
public class Moon extends Earth {
}
```

Now, based on the above example, in Object-Oriented terms, the following are true:

- SolarSystem is the superclass of Earth class.
- SolarSystem is the superclass of Mars class.
- Earth and Mars are subclasses of SolarSystem class.
- Moon is the subclass of both Earth and SolarSystem classes.

```
class SolarSystem {
}
class Earth extends SolarSystem {
}
class Mars extends SolarSystem {
}
public class Moon extends Earth {
    public static void main(String args[])
    {
        SolarSystem s = new SolarSystem();
        Earth e = new Earth();
        Mars m = new Mars();

        System.out.println(s instanceof SolarSystem);
        System.out.println(e instanceof Earth);
        System.out.println(m instanceof SolarSystem);
    }
}

Output
true
true
true
```

**What Can Be Done in a Subclass?**
In subclasses we can inherit members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it (as in the example above, toString() method is overridden).
- We can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- We can declare new methods in the subclass that are not in the superclass.
- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

**Advantages of Inheritance in Java**

- **Code Reusability:** Inheritance allows for code reuse and reduces the amount of code that needs to be written. The subclass can reuse the properties and methods of the superclass, reducing duplication of code.
- **Abstraction:** Inheritance allows for the creation of abstract classes that define a common interface for a group of related classes. This promotes abstraction and encapsulation, making the code easier to maintain and extend.
- **Class Hierarchy:** Inheritance allows for the creation of a class hierarchy, which can be used to model real-world objects and their relationships.
- **Polymorphism:** Inheritance allows for polymorphism, which is the ability of an object to take on multiple forms. Subclasses can override the methods of the superclass, which allows them to change their behavior in different ways.

# Polymorphism in Java

Polymorphism in Java is one of the core concepts in object-oriented programming (OOP) that allows objects to behave differently based on their specific class type. The word polymorphism means having many forms, and it comes from the Greek words poly (many) and morph (forms), this means one entity can take many forms. In Java, polymorphism allows the same method or object to behave differently based on the context, specially on the project's actual runtime class.

**Key features of polymorphism:**

- **Multiple Behaviors:** The same method can behave differently depending on the object that calls this method.
- **Method Overriding:** A child class can redefine a method of its parent class.
- **Method Overloading:** We can define multiple methods with the same name but different parameters.

- **Runtime Decision:** At runtime, Java determines which method to call depending on the object's actual class.

**Real-Life Illustration of Polymorphism**
Consider a person who plays different roles in life, like a father, a husband, and an employee. Each of these roles defines different behaviors of the person depending on the object calling it.

**Example:** Different Roles of a Person

```
class Person {
    void role() {
        System.out.println("I am a person.");
    }
}
class Father extends Person {
    @Override
    void role() {
        System.out.println("I am a father.");
    }
}
public class Main {
    public static void main(String[] args) {
        Person p = new Father();
        p.role();
    }
}

Output
I am a father.
```

**Explanation:** In the above example, the Person class has a method role() that prints a general message. The Father class overrides role() to print a specific message. The reference of type Person is used to point to an object of type Father, demonstrating polymorphism at runtime. The overridden method in Father is invoked when role() is called.

**Why Use Polymorphism In Java?**
Using polymorphism in Java has many benefits which are listed below:

- **Code Reusability:** Polymorphism allows the same method or class to be used with different types of objects, which makes the code more usable.
- **Flexibility:** Polymorphism enables objects of different classes to be treated as objects of a common superclass, which provides flexibility in method execution and object interaction.
- **Abstraction:** It allows the use of abstract classes or interfaces, enabling you to work with general types (like a superclass or interface) instead of

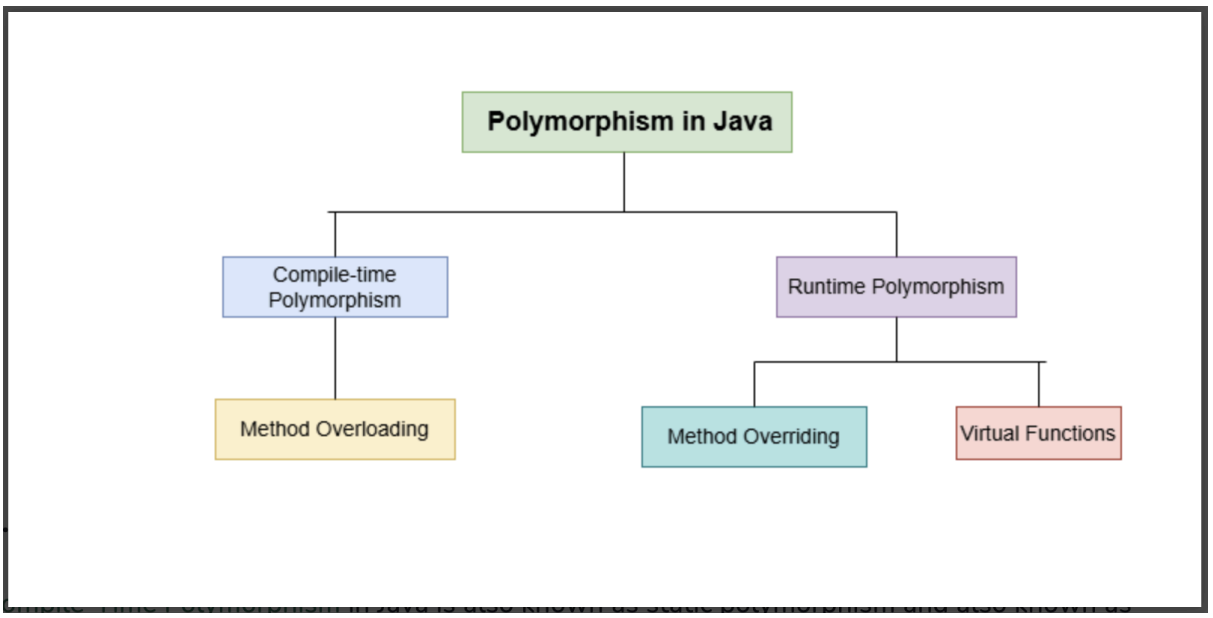
concrete types (like specific subclasses), thus simplifying the interaction with objects.

- **Dynamic Behavior:** With polymorphism, Java can select the appropriate method to call at runtime, giving the program dynamic behavior based on the actual object type rather than the reference type, which enhances flexibility.

**Types of Polymorphism in Java**

In Java Polymorphism is mainly divided into two types:

- Compile-Time Polymorphism (Static)
- Runtime Polymorphism (Dynamic)




**1. Compile-Time Polymorphism**

Compile-Time Polymorphism in Java is also known as static polymorphism and also known as method overloading. This happens when multiple methods in the same class have the same name but different parameters.

**Note**: But Java doesn't support the Operator Overloading.

**Method Overloading**

As we discussed above, Method overloading in Java means when there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

**Example**: Method overloading by changing the number of arguments

```
class Helper {
    static int Multiply(int a, int b) {
        return a * b;
    }

    static double Multiply(double a, double b) {
        return a * b;
    }
}

Class Main {
    public static void main(String[] args) {
        System.out.println(Helper.Multiply(2, 4));
        System.out.println(Helper.Multiply(5.5, 6.3));
    }
}

Output
8
34.65
```

**Explanation:** The Multiply method is overloaded with different parameter types. The compiler picks the correct method during compile time based on the arguments.

In Java, Method Overloading allows us to define multiple methods with the same name but different parameters within a class. This difference may include:

- The number of parameters
- The types of parameters
- The order of parameters

Method overloading in Java is also known as Compile-time Polymorphism, Static Polymorphism, or Early binding, because the decision about which method to call is made at compile time. When there are overloaded methods that accept both a parent type and a child type, and the provided argument could match either one, Java prefers the method that takes the more specific (child) type. This is because it offers a better match.

**Key features of Method Overloading**

- Multiple methods can share the same name in a class when their parameter lists are different.
- Overloading is a way to increase flexibility and improve the readability of code.
- Overloading does not depend on the return type of the method, two methods cannot be overloaded by just changing the return type.

Now, let us go through a simple example to understand the concept better:

**Example:** This example demonstrates method overloading by defining multiple sum() methods with different parameter types and counts.

```
public class Sum {
    public int sum(int x, int y) { return (x + y); }

    public int sum(int x, int y, int z) {
        return (x + y + z);
    }

    public double sum(double x, double y) {
        return (x + y);
    }

    public static void main(String args[]) {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}

Output
30
60
31.0
```

**Different Ways of Method Overloading in Java**

The different ways of method overloading in Java are mentioned below:

- Changing the Number of Parameters
- Changing Data Types of the Arguments
- Changing the Order of the Parameters of Methods

**1. Changing the Number of Parameters**

Method overloading can be achieved by changing the number of parameters while passing to different methods.

**Example:** This example demonstrates method overloading by changing the number of parameters in the multiply() method.

```
class Product {
    public int multiply(int a, int b) {
```

```
        int prod = a * b;
        return prod;
    }

    public int multiply(int a, int b, int c) {
        int prod = a * b * c;
        return prod;
    }
}

class Main {
    public static void main(String[] args) {
        Product ob = new Product();
        int prod1 = ob.multiply(1, 2);

        System.out.println(
            "Product of the two integer value: " + prod1);

        int prod2 = ob.multiply(1, 2, 3);

        System.out.println(
            "Product of the three integer value: " + prod2);
    }
}

Output
Product of the two integer value: 2
Product of the three integer value: 6
```

**2. Changing Data Types of the Arguments**

In many cases, methods can be considered overloaded if they have the same name but have different parameter types, methods are considered to be overloaded.

**Example:** This example demonstrates method overloading by changing the data types of parameters in the Prod() method.

```
class Product {
    public int Prod(int a, int b, int c) {
        int prod1 = a * b * c;
        return prod1;
    }

    public double Prod(double a, double b, double c) {
```

```
        double prod2 = a * b * c;
        return prod2;
    }
}

class Main {
    public static void main(String[] args) {
        Product obj = new Product();

        int prod1 = obj.Prod(1, 2, 3);
        System.out.println(
            "Product of the three integer value: " + prod1);

        double prod2 = obj.Prod(1.0, 2.0, 3.0);
        System.out.println(
            "Product of the three double value: " + prod2);
    }
}

Output
Product of the three integer value: 6
Product of the three double value: 6.0
```

**3. Changing the Order of the Parameters of Methods**

Method overloading can also be implemented by rearranging the parameters of two or more overloaded methods. For example, if the parameters of method 1 are (String name, int roll_no) and the other method is (int roll_no, String name) but both have the same name, then these 2 methods are considered to be overloaded with different sequences of parameters.

**Example:** This example demonstrates method overloading by changing the order of parameters in the StudentId() method.

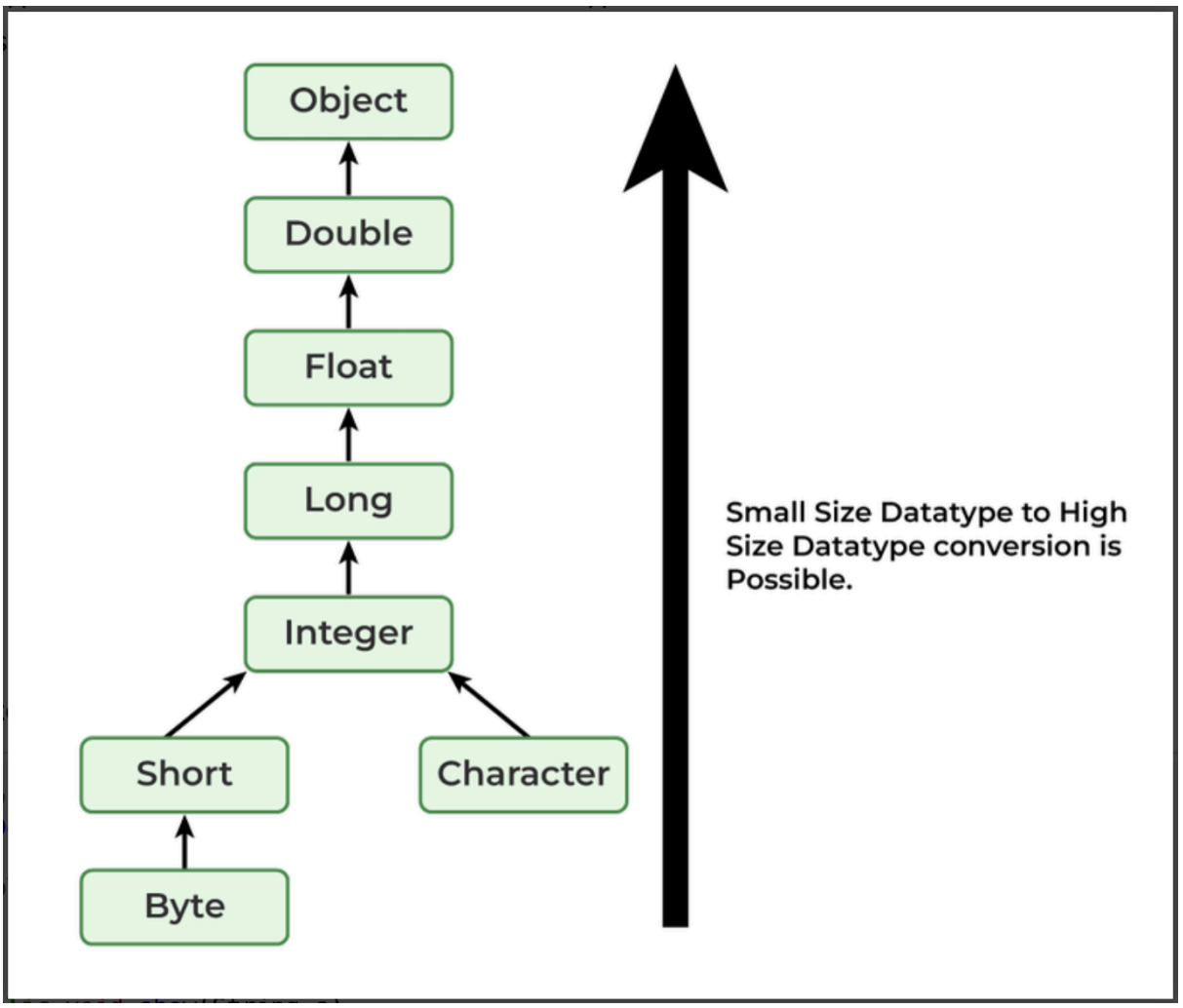
```
class Student {
    public void StudentId(String name, int rollNo) {
        System.out.println("Name: " + name + " "
                           + "Roll No: " + rollNo);
    }

    public void StudentId(int rollNo, String name) {
        System.out.println("Roll No: " + rollNo + " "
                           + "Name: " + name);
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Student obj = new Student();

        obj.StudentId("Test", 1);
        obj.StudentId(2, "Test");
    }
}

Output
Name: Test Roll No: 1
Roll No: 2 Name: Test
```

**Case1: Automatic promotion in overloading**

If no method matches exactly, priority-wise, the compiler takes these steps:

- Type Conversion but to a higher type (in terms of range) in the same family (for example, byte -> int)
- Type conversion to the next higher family. Suppose if there is no long data type available for an int data type, then it will search for the float data type.
- If no suitable method is found, compilation error.




Let's take an example to clarify the concept:

```
class Demo {

    public void show(int x) {
        System.out.println("In int" + x);
    }
    public void show(String s) {
        System.out.println("In String" + s);
```

```
    }
    public void show(byte b) {
        System.out.println("In byte" + b);
    }
}

class UseDemo {

    public static void main(String[] args) {
        byte a = 25;
        Demo obj = new Demo();
        obj.show(a);
        obj.show("hello");
        obj.show(250);
        obj.show('A');
        obj.show("A");
        obj.show(7.5);
    }
}

Output:

./UseDemo.java:46: error: no suitable method found for show(double)
        obj.show(7.5);
           ^
    method Demo.show(int) is not applicable
      (argument mismatch; possible lossy conversion from double to int)
    method Demo.show(String) is not applicable
      (argument mismatch; double cannot be converted to String)
    method Demo.show(byte) is not applicable
      (argument mismatch; possible lossy conversion from double to byte)
1 error
```

**Case 2: Overloaded methods exact match will always get high priority**

```
Class Test {
    public void methodOne(String s) {
        System.out.println("String version");
    }
    public void methodOne(Object o) {
        System.out.println("Object version");
```

```
    }

    public static void main(String[] args) {
        Test t=new Test();
        t.methodOne("arun");
        t.methodOne(new Object());
        t.methodOne(null);
    }
}

Output:
String version
Object version
String version
```

**Note:**
While resolving overloaded methods exact match will always get high priority,
While resolving overloaded methods child class will get the more priority than parent class

**Case 3: Method overloading with respect to same data**

```
class Test{
    public void methodOne(String s) {
        System.out.println("String version");
    }

    public void methodOne(StringBuffer s) {
        System.out.println("StringBuffer version");
    }

    public static void main(String[] args){
        Test t=new Test();
        t.methodOne("Test"); //String version
        t.methodOne(new StringBuffer("Test")); //StringBuffer version
        t.methodOne(null); //CE:reference to methodOne() is ambiguous
    }
}
```

**Case 4: Method overloading with respect to ambiguous auto promotion**

```
class Test {
    public void methodOne(int i, float f) {
         System.out.println("int-float method");
    }
```

```
    public void methodOne(float f, int i) {
        System.out.println("float-int method");
    }

    public static void main(String[] args){
        Test t=new Test();
        t.methodOne(10, 10.5f); //int-float method
        t.methodOne(10.5f, 10); //float-int method
        t.methodOne(10, 10); //CE:reference to methodOne is ambiguous,
        t.methodOne(10.5f,10.5f); //C.E: cannot find symbol
        symbol : methodOne(float, float) location : class Test
    }
}
```

**Case 5:method overloading vs var-arg method**

```
class Test {
    public void methodOne(int i) {
        System.out.println("general method");
    }

    public void methodOne(int... i) {
        System.out.println("var-arg method");
    }

    public static void main(String[] args){
        Test t = new Test();
        t.methodOne(); //var-arg method
        t.methodOne(10, 20); //var-arg method
        t.methodOne(10); //general method
    }
}
```

**Note:**
In general the var-arg method will get less priority, that is if no other method matches then only the var-arg method will get a chance for execution; it is almost the same as the default case inside switch.

**Case 6: Method overloading vs method resolution**

```
class Animal {}
class Monkey extends Animal {}

class Test{
```

```
    public void methodOne(Animal a) {
        System.out.println("Animal version");
    }

    public void methodOne(Monkey m) {
        System.out.println("Monkey version");
    }

    public static void main(String[] args){
        Test t = new Test();
        Animal a = new Animal();
        t.methodOne(a); //Animal version

        Monkey m = new Monkey();
        t.methodOne(m); //Monkey version

        Animal a1 = new Monkey();
        t.methodOne(a1);//Animal version
    }
}
```

**Note:**
In overloading method resolution is always based on reference type and runtime object won't play any role in overloading.

**Advantages**
The advantages of method overloading is listed below:

- Method overloading improves the Readability and reusability of the program.
- Method overloading reduces the complexity of the program.
- Using method overloading, programmers can perform a task efficiently and effectively.
- Using method overloading, it is possible to access methods performing related functions with slightly different arguments and types.
- Objects of a class can also be initialized in different ways using the constructors.

**2. Runtime Polymorphism**
Runtime Polymorphism in Java known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding. Method overriding,

on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

**Method Overriding**

Method overriding in Java means when a subclass provides a specific implementation of a method that is already defined in its superclass. The method in the subclass must have the same name, return type, and parameters as the method in the superclass. Method overriding allows a subclass to modify or extend the behavior of an existing method in the parent class. This enables dynamic method dispatch, where the method that gets executed is determined at runtime based on the object's actual type.

**Example:** This program demonstrates method overriding in Java, where the Print() method is redefined in the subclasses (subclass1 and subclass2) to provide specific implementations.

```
class Parent {
    void print() {
        System.out.println("parent class");
    }
}

class subclass1 extends Parent {
    void print() {
      System.out.println("subclass1");
    }
}

class subclass2 extends Parent {
    void print() {
        System.out.println("subclass2");
    }
}

class Main {
    public static void main(String[] args) {
        Parent a;
        a = new subclass1();
        a.Print();



        a = new subclass2();
        a.Print();
    }
}
```

```
Output
subclass1
subclass2
```

**Explanation:** In the above example, when an object of a child class is created, then the method inside the child class is called. This is because the method in the parent class is overridden by the child class. This method has more priority than the parent method inside the child class. So, the body inside the child class is executed.

**Rules for Overriding:**

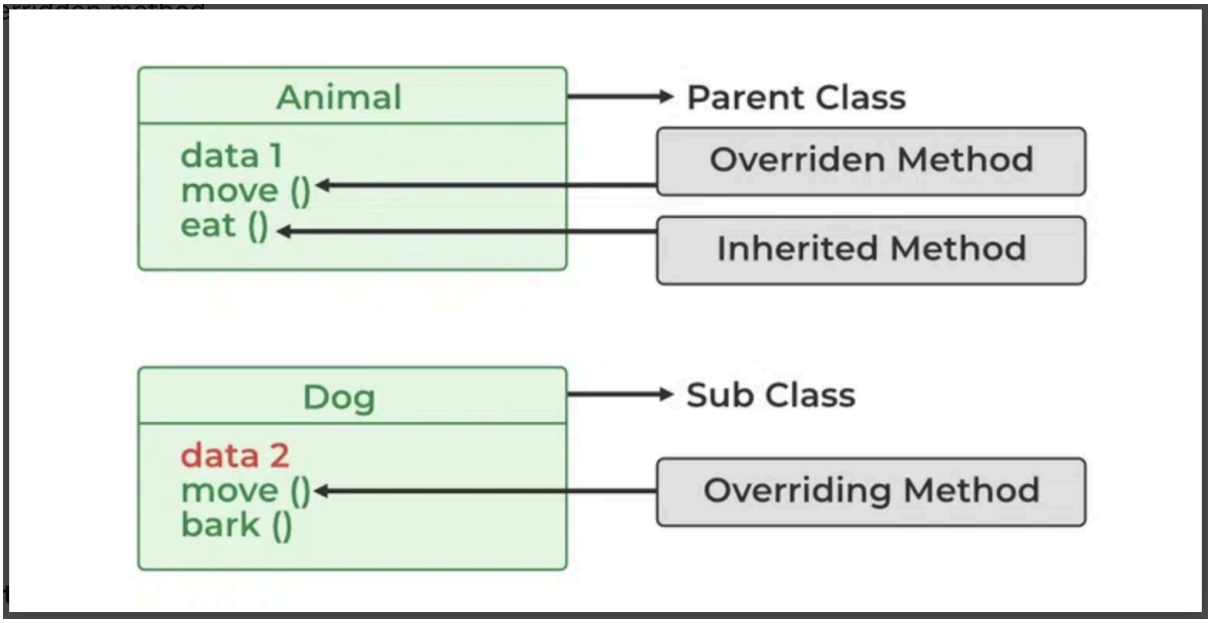
- Name, parameters, and return type must match the parent method.
- Java picks which method to run, based on the actual object type, not just the variable type.
- Static methods cannot be overridden.
- The @Override annotation catches mistakes like typos in method names.

**Example:** In the code below, Dog overrides the move() method from Animal but keeps eat() as it is. When we call move() on a Dog object, it runs the dog-specific version.

```
class Animal {
    void move() {
        System.out.println("Animals are moving.");
    }

    void eat() {
        System.out.println("Animals are eating.");
    }
}

class Dog extends Animal {
    @Override
    void move() {
        System.out.println("Dog is running.");
    }

    void bark() { System.out.println("Dog is barking."); }
}


public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.move();
```

```
        d.eat();
        d.bark();
    }
}

Output
Dog is running.
Animals are eating.
Dog is barking.
```

**Explanation:** The Animal class defines base functionalities like move() and eat(). The Dog class inherits from Animal and overrides the move() method to provide a specific behavior Dog is running. Both classes can access their own methods. When creating a Dog object, calling move() executes the overridden method.




**Note:** Method overriding is a key concept in Java that enables Run-time polymorphism. It allows a subclass to provide its specific implementation for a method inherited from its parent class.

**Example:** This example demonstrates runtime polymorphism in Java, where the show() method is overridden in the Child class, and the method called depends on the object type at runtime.

```
class Parent {
    void show() { System.out.println("Parent's show()"); }
}
```

```
class Child extends Parent {
    @Override
    void show(){
        System.out.println("Child's show()");
    }
}

class Main {
    public static void main(String[] args) {
        Parent obj1 = new Parent();
        obj1.show();

        Parent obj2 = new Child();
        obj2.show();
    }
}

Output
Parent's show()
Child's show()
```

**Explanation:** In this code, the Child class inherits from the Parent class and overrides the show() method, providing its own implementation. When a Parent reference points to a Child object, the Child's overridden show() method is executed at runtime, showcasing the principle of polymorphism in Java.

**Why Override Methods?**

The key reasons to use method overriding in Java are listed below:

- Change how a parent class method works in a subclass, for example, a Dog moves by running, while a Fish might swim.
- It enables polymorphism, that let one method call adapt to different objects at runtime.
- Reuse method names logically instead of creating new ones for minor changes.

**Rules for Java Method Overriding**

**1. Access Modifiers in overriding**

A subclass can make an overridden method more accessible, e.g., upgrade protected to public, but not less e.g., downgrade public to private. Trying to hide a method this way causes compiler errors.

**Example:** Below, Child legally overrides m2() with broader public access, but cannot override m1() because it's private in the parent.

```
class Parent {
    private void m1() {
        System.out.println("From parent m1()");
    }

    protected void m2() {
        System.out.println("From parent m2()");
    }
}

class Child extends Parent {
    private void m1() {
        System.out.println("From child m1()");
    }

    @Override
    public void m2() {
        System.out.println("From child m2()");
    }
}

class Main {
    public static void main(String[] args) {
        Parent P = new Parent();
        P.m2();

        Parent C = new Child();
        C.m2();
    }
}

Output
From parent m2()
From child m2()
```

**Explanation:** Here, the parent class is overridden by the subclass and from the output we can easily identify the difference.

### 2. Final Methods Cannot Be Overridden

If we don't want a method to be overridden, we declare it as final. Please see Using Final with Inheritance.

**Note:** If a method is declared as final, subclasses cannot override it.

**Example:** This example demonstrates that final methods in Java cannot be overridden in subclasses.

```
class Parent {
    final void show() {}
}

class Child extends Parent {
    // This would produce error
    void show() {}
}
```

### 3. Static Methods Cannot Be Overridden (Method Hiding)

When we define a static method with the same signature as a static method in the base class, it is known as method hiding.

- Subclass Instance method can override the superclass's Instance method but when we try to override the superclass static method gives a compile time error.
- Subclass Static method generates compile time when trying to override superclass Instance method subclass static method hides when trying to override superclass static method.

**Example:** This example demonstrates method hiding for static methods in Java, where a static method in the subclass hides the static method in the superclass, while non-static methods can be overridden.

```
class Parent {
    static void m1() {
        System.out.println("From parent static m1()");
    }

    void m2() {
        System.out.println( "From parent non - static(instance) m2() ");
    }
}


class Child extends Parent {
    static void m1() {
        System.out.println("From child static m1()");
    }

    @Override
```

```
    public void m2() {
        System.out.println("From child non - static(instance) m2()");
    }
}

class Main {
    public static void main(String[] args) {
        Parent obj1 = new Child();

        // here parents m1 called.
        // bcs static method can not overridden
        obj1.m1();

        // Here overriding works
        // and Child's m2() is called
        obj1.m2();
    }
}

Output
From parent static m1()
From child non - static(instance) m2()
```

**Note:** When a subclass defines a static method with the same signature as the parent's static method, it hides the parent method. The method called depends on the reference type, not the object type.

**4. Private Methods Cannot Be Overridden**

Private methods cannot be overridden as they are bonded during compile time. We cannot even override private methods in a subclass.

**Example:** This example demonstrates method hiding for private methods in Java, where a private method in the subclass does not override the private method in the superclass, and the subclass's public method overrides the superclass's public method.

```
class SuperClass {
    private void privateMethod() {
        System.out.println("It is a private method in SuperClass");
    }
```

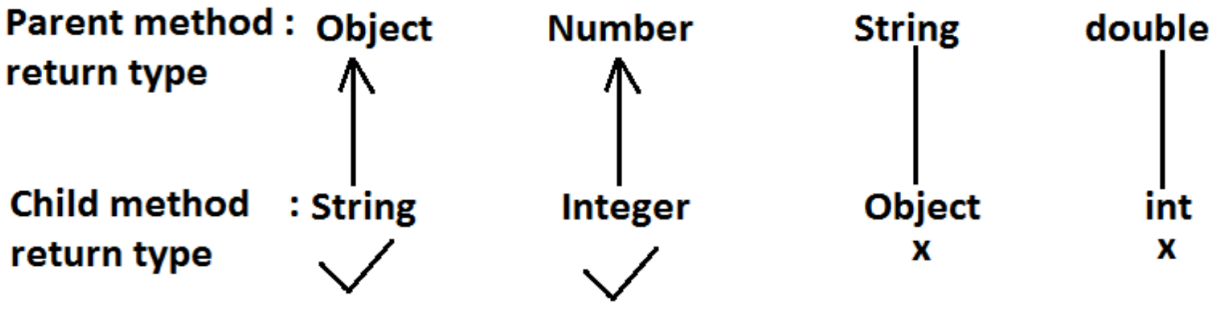
```
    public void publicMethod() {
        System.out.println("It is a public method in SuperClass");
        privateMethod();
    }
}

class SubClass extends SuperClass {
    private void privateMethod() {
        System.out.println("It is private method in SubClass");
    }

    public void publicMethod() {
        System.out.println("It is a public method in SubClass");
        privateMethod();
    }
}

public class Main {
    public static void main(String[] args) {
        SuperClass o1 = new SuperClass();
        o1.publicMethod();

        SubClass o2 = new SubClass();
        o2.publicMethod();
    }
}

Output
It is a public method in SuperClass
It is a private method in SuperClass
It is a public method in SubClass
It is a private method in SubClass
```

**5. Method Must Have the Same Return Type (or subtype)**

From Java 5.0 onwards it is possible to have different return types for an overriding method in the child class, but the child's return type should be a sub-type of the parent's return type. This phenomenon is known as the covariant return type. Covariant return type concept is applicable only for object types but not for primitives.

**Example:** This example demonstrates covariant return types in Java, where the subclass method overrides the superclass method with a more specific return type.

```
class SuperClass {
    public Object method() {
        System.out.println("This is the method in SuperClass");
        return new Object();
    }
}

class SubClass extends SuperClass {
    public String method() {
        System.out.println("This is the method in SubClass");
        return "Hello, World!";
    }
}

public class Main {
    public static void main(String[] args) {
        SuperClass obj1 = new SuperClass();
        obj1.method();
        SubClass obj2 = new SubClass();
        obj2.method();
    }
}

Output
This is the method in SuperClass
This is the method in SubClass
```




**6. Invoking Parent's Overridden Method Using super**

We can call the parent class method in the overriding method using the super keyword.

**Note:** Use super.methodName() to call the parent's version.

**Example**: This example demonstrates calling the overridden method from the subclass using super to invoke the superclass method in Java.

```
class Parent {
    void show() { System.out.println("Parent's show()"); }
}

class Child extends Parent {
    @Override
    void show() {
        super.show();
        System.out.println("Child's show()");
    }
}

class Main {
    public static void main(String[] args) {
        Parent o = new Child();
        o.show();
    }
}

Output
Parent's show()
Child's show()
```

**7. Overriding with respect to Var-arg methods:**
A var-arg method should be overridden with var-arg method only. If we are trying to override with a normal method then it will become overloading but not overriding.

**Example:**

```


class Parent {
    public void methodOne(int... i){
        System.out.println("parent class");
    }
}

class Child extends Parent {
    public void methodOne {
        System.out.println("child class");
    }
}
```

**Overriding and Constructor**
We cannot override the constructor as the parent and child class can never have a constructor with the same name (the constructor name must always be the same as the Class name).

**Exception-Handling in Overriding**
Below are two rules to note when overriding methods related to exception handling.

**Rule 1:** If the superclass overridden method does not throw an exception, the subclass overriding method can only throw the unchecked exception, throwing a checked exception will lead to a compile-time error.

**Example:** Below is an example of a Java program when the parent class method

```
class Parent {
    void m1() { System.out.println("From parent m1()"); }

    void m2() { System.out.println("From parent  m2()"); }
}

class Child extends Parent {
    @Override
    void m1() throws ArithmeticException {
        System.out.println("From child m1()");
    }

    @Override
    void m2() throws Exception {
        System.out.println("From child m2");
    }
}
```

**Explanation:** This example shows that the superclass overridden method does not throw an exception, the subclass overriding method can only throw the exception because the superclass does not declare the exception.

**Rule 2:** If the superclass overridden method does throw an exception, the subclass overriding method can only throw the same, subclass exception. Throwing parent exceptions in the Exception hierarchy will lead to compile time error. Also, there is no issue if the subclass overridden method does not throw any exception.

**Example:** This example demonstrates overriding when the superclass method does declare an exception.

```
class Parent {
    void m1() throws RuntimeException {
        System.out.println("From parent m1()");
    }
}

class Child1 extends Parent {
    @Override
    void m1() throws RuntimeException {
        System.out.println("From child1 m1()");
    }
}

class Child2 extends Parent {
    @Override
    void m1() throws ArithmeticException {
        System.out.println("From child2 m1()");
    }
}

class Child3 extends Parent {
    @Override
    void m1() {
        System.out.println("From child3 m1()");
    }
}

class Child4 extends Parent {
    @Override
     void m1() throws Exception {
        System.out.println("From child4 m1()");
    }
}




public class MethodOverridingExample {
    public static void main(String[] args) {
        Parent p1 = new Child1();
        Parent p2 = new Child2();
        Parent p3 = new Child3();
        Parent p4 = new Child4();
```

```
        try {
            p1.m1();
        } catch (RuntimeException e) {
            System.out.println("Exception caught: " + e);
        }

        try {
            p2.m1();
        } catch (RuntimeException e) {
            System.out.println("Exception caught: " + e);
        }

        try {
            p3.m1();
        } catch (Exception e) {
            System.out.println("Exception caught: " + e);
        }

        try {
            p4.m1();
        } catch (Exception e) {
            System.out.println("Exception caught: " + e);
        }
    }
}
```

**Overriding and Abstract Method**

Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise a compile-time error will be thrown.

**Overriding and Synchronized/strictfp Method**

The presence of a synchronized/strictfp modifier with the method has no effect on the rules of overriding, i.e. it's possible that a synchronized/strictfp method can override a non-synchronized/strictfp one and vice-versa.

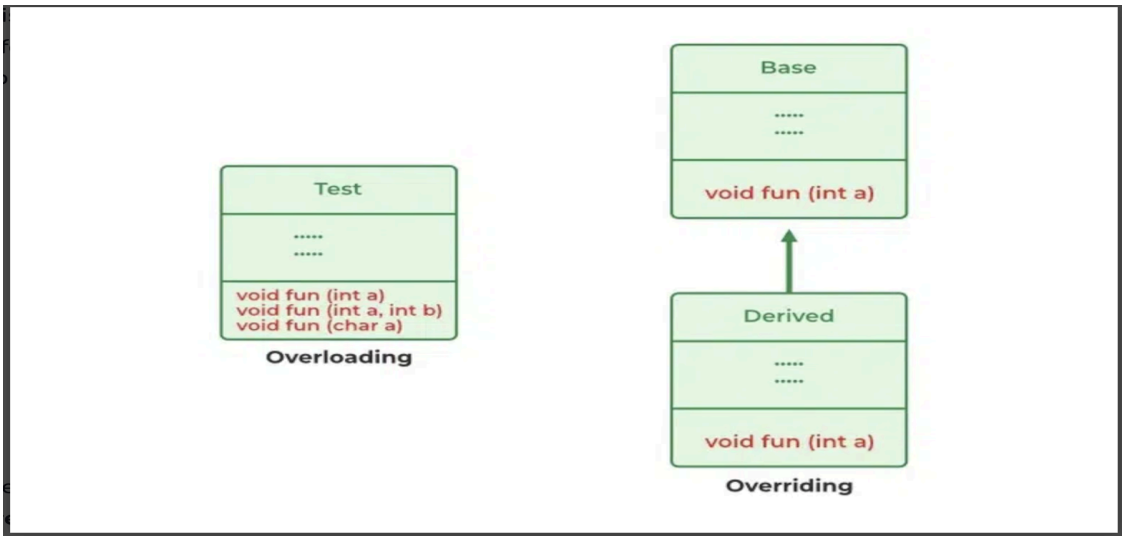
**Example:** This example demonstrates multi-level method overriding in Java, where a method is overridden across multiple levels of inheritance, and the method called is determined at runtime.

```
class Parent {
    void show() { System.out.println("Parent's show()"); }
}
```

```
class Child extends Parent {
    void show() { System.out.println("Child's show()"); }
}

class GrandChild extends Child {
    void show() {
        System.out.println("GrandChild's show()");
    }
}

class Main {
    public static void main(String[] args) {
        Parent o = new GrandChild();
        o.show();
    }
}

Output
GrandChild's show()
```

**Method Overriding vs Method Overloading**

This image demonstrates the difference between method overloading (same method name but different parameters in the same class) and overriding (same method signature in a subclass, replacing the superclass method).




**Method Overloading vs Method Overriding**

The table below demonstrates the difference between Method Overriding and Method Overloading.

| Features | Method Overriding | Method Overloading |
|---|---|---|

| | | |
|---|---|---|
| Definition | Method overriding is about the same signature in subclass. | Method overloading is about the same method name, different parameters. |
| Polymorphism | It is also known as Runtime polymorphism | It is also known as Compile Time polymorphism |
| Inheritance | Requires inheritance. | Can be in the same class or subclass |
| Return Type | Return type must be same | Return type can be different |
| Exceptions | Must follow overriding rules. | No restrictions. |

**Advantages of Polymorphism**

- Encourages code reuse.
- Simplifies maintenance.
- Enables dynamic method dispatch.
- Helps in writing generic code that works with many types.

**Disadvantages of Polymorphism**

- It can make it more difficult to understand the behavior of an object.
- This may cause performance issues, as polymorphic behavior may require additional computations at runtime.

# super() vs this()

The 1st line inside every constructor should be either super() or this() if we are not writing anything, the compiler will always generate super().

**Case 1:** We have to take super() (or) this() only in the 1st line of the constructor. If we are taking anywhere else we will get a compile time error.

**Example:** Call to super must be first statement in constructor

```
class Test {
    Test() {
        System.out.println("constructor");
        super();
    }
}

Output:
Compile time error.
```

**Case 2:** We can use either super() (or) this() but not both simultaneously.

**Example:** Call to this must be first statement in constructor

```
class Test {
    Test() {
        super();
        this();
    }
}
Output:
Compile time error.
```

**Case 3:** We can use super() (or) this() only inside the constructor. If we are using anywhere else we will get a compile time error.

**Example:** Call to super must be first statement in constructor

```
class Test {
    public void methodOne() {
         super();
    }
}
Output:
Compile time error.
```

**Note:** That is, we can call a constructor directly from another constructor only.

**Difference between super(), this() and super, this**

| super(), this() | super, this |
|---|---|
| These are constructors calls. | These are keywords |
| We can use these to invoke super class & current constructors directly. | We can use parent class and current class instance members. |
| We should use only inside constructors as the first line, if we are using outside constructors we will get compile time errors. | We can use anywhere (i.e., instance area) except static areas , otherwise we will get compile time errors . |

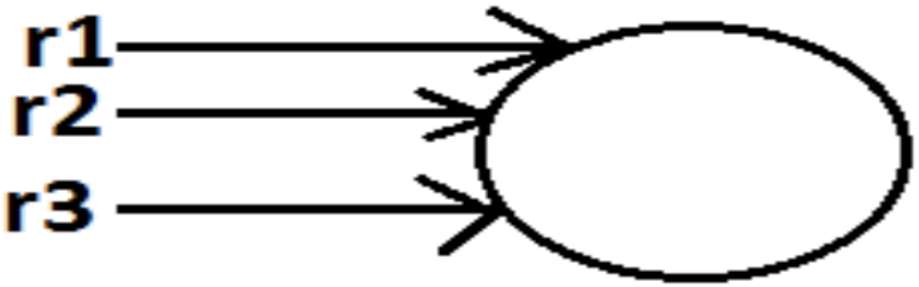
**Singleton classes :**
For any java class if we are allowed to create only one object such type of class is said to be singleton class.

**Example:**

- Runtime class
- ActionServlet
- ServiceLocator
- BusinessDelegate

```
Runtime r1 = Runtime.getRuntime();
//getRuntime() method is a factory method
Runtime r2 = Runtime.getRuntime();
Runtime r3 = Runtime.getRuntime();
.....................................................
.....................................................
System.out.println(r1==r2);//true
System.out.println(r1==r3);//true
```

**Diagram:**



**Advantage of Singleton class:**
If the requirement is same then instead of creating a separate object for every person we will create only one object and we can share that object for every required person we can achieve this by using singleton classes. That is the main advantages of singleton classes are Performance will be improved and memory utilization will be improved.

**Creation of our own singleton classes:**
We can create our own singleton classes for this we have to use private constructor, static variable and factory method.

**Example:**

```
class Test {
    private static Test t = null;

    private Test(){
    }

    public static Test getTest() {
        if(t == null) {
            t = new Test();
            return t;
        }
    }

}

class Client {
    public static void main(String[] args) {
        System.out.println(Test.getTest().hashCode());//1671711
        System.out.println(Test.getTest().hashCode());//1671711
        System.out.println(Test.getTest().hashCode());//1671711
        System.out.println(Test.getTest().hashCode());//1671711
    }
}
```

**Static Control Flow:**

- Identification of static members from top to bottom[1 to 6]
- Execution of static variable assignments and static blocks from top to bottom[7 to 12]
- Execution of main method[13 to 15]

## Static control flow :

**Example:**

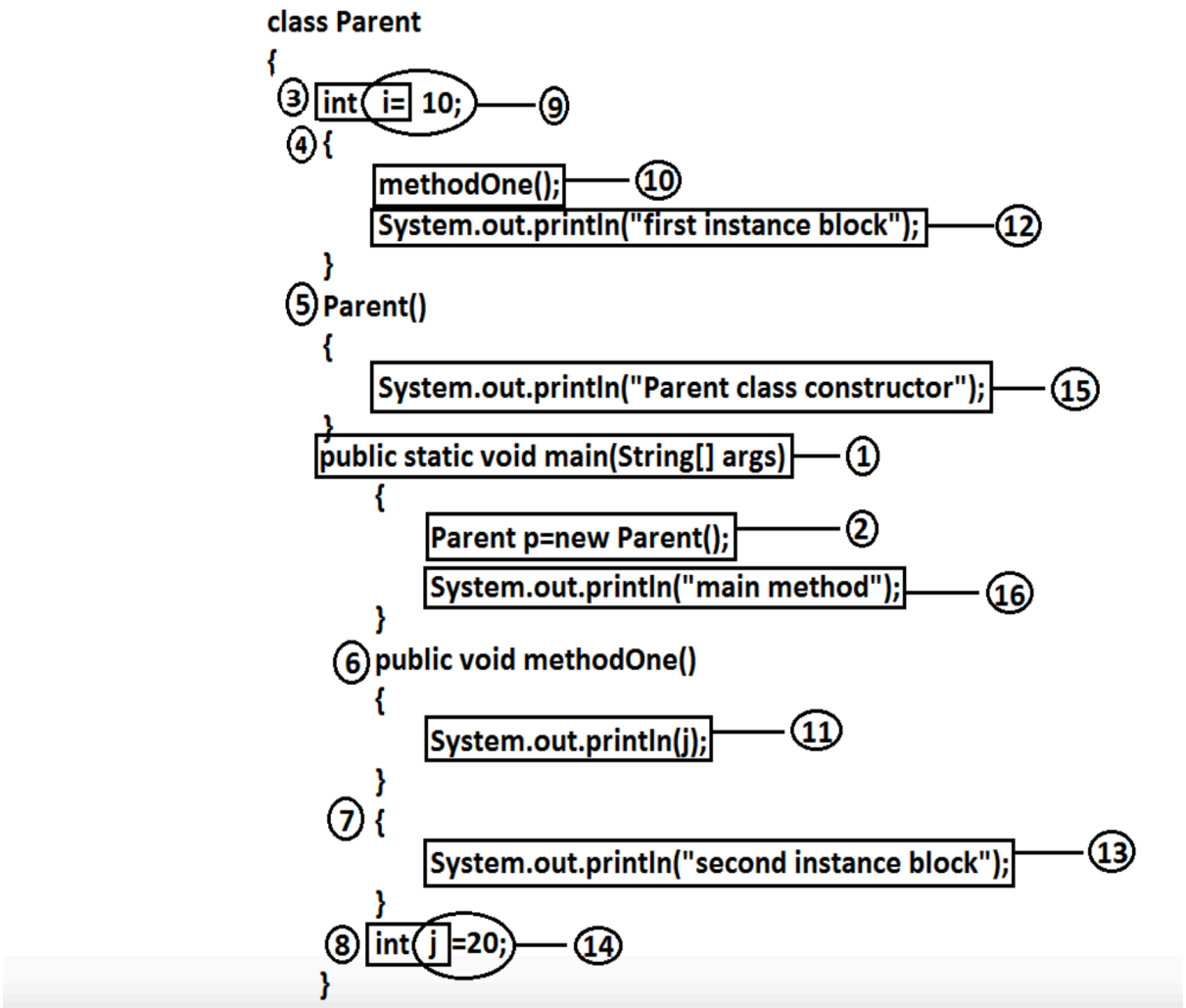
```
class Base
{
  ①static int i=10;  ——⑦
  ②static
    {
        methodOne();  ——⑧
        System.out.println("first static block");  ——⑩
    }
  ③public static void main(String[] args)
    {
        methodOne();  ——⑬
        System.out.println("main method");  ——⑮
    }
  ④public static void methodOne()
        {
            System.out.println(j);  ——⑨,⑭
        }
    ⑤static
        {
            System.out.println("second static block");  ——⑪
        }
    ⑥static int j=20;  ——⑫
  }
```

**Instance Control flow:**

Whenever we are executing a java class static control flow will be executed. In the Static control flow Whenever we are creating an object the following sequence of events will be performed automatically.

- Identification of instance members from top to bottom(3 to 8).
- Execution of instance variable assignments and instance blocks from top to bottom(9 to 14).
- Execution of constructor

## Instance control flow:



**Type casting:**
Parent class reference can be used to hold Child class objects but by using that reference we can't call Child specific methods.

**Example:**

```
Object o=new String("Test");//valid
System.out.println(o.hashCode());//valid
System.out.println(o.length());// C.E:cannot find symbol,
symbol : method length(), location: class java.lang.Object
```