

- 1. Java Garbage Collection**
- 2. Object-Oriented Concepts in Java**
- 3. Constructor in Java**
- 4. Inheritance**
- 5. Relationship Between Classes in Java**
- 6. Polymorphism**
- 7. Abstraction**
- 8. Interfaces**
- 9. Encapsulation**
- 10.This Keyword**
- 11.Super Keyword**
- 12.Final Keyword**
- 13.Static Keyword**

# 1. Java Garbage Collection

- In java, garbage means unreferenced objects.
- Garbage Collection is process of reclaiming the runtime unused memory automatically.
- In other words, it is a way to destroy the unused objects.
- To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically.
- So, java provides better memory management.

## 1) Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

## 2) How can an object be unreferenced?

There are many ways:

I. Nulling a reference:

```
java
Employee e = new Employee();
e = null;
```

Copy code

II. Assigning a reference to another:

```
java
Employee e1 = new Employee();
Employee e2 = new Employee();
e1 = e2; // Now the first object referred to by e1 is available for garbage
```

Copy code

III. Using an anonymous object:

```
java
new Employee();
```

Copy code

## 3) WAY TO SUGGEST JVM FOR GC

### 1) finalize() method

- The finalize() method is invoked each time before the object is garbage collected.
- This method can be used to perform cleanup processing. This method is defined in Object class as:

```
java
protected void finalize() {
    // Cleanup processing before garbage collection (Not guaranteed to run)
}
```

Copy code

- Note: The Garbage collector of JVM collects only those objects that are created by new keyword.
- So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

### 2) gc() method

- The gc() method is used to invoke the garbage collector to perform cleanup processing. 'The gc()' is found in System and Runtime classes

```
java
public static void gc() {
    // Suggest garbage collection to run (Not guaranteed to execute immediat
}
```

Copy code

- Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

#### 4) Simple Example of garbage collection in java

```
1
2 public class TestGarbage1
3 {
4     public void finalize()
5     {
6         System.out.println("object is garbage collected");
7     }
8
9     public static void main(String args[])
10    {
11        TestGarbage1 s1=new TestGarbage1();
12        TestGarbage1 s2=new TestGarbage1();
13
14        s1=null;
15        s2=null;
16
17        System.gc();
18    }
19 }
```

## 2. Object-Oriented Concepts in Java

### 5) Introduction to Object-Oriented Programming (OOP):

Object-Oriented Programming (OOP) is a programming paradigm that provides a structured approach to solving problems using objects and their interactions. It focuses on modeling real-world entities and their behaviors, making it easier to understand and manage complex systems. OOP encourages the organization of code into reusable and modular components, promoting code reusability and maintainability.

### 6) Main Concepts (Pillars) of OOP:


- 1) **Class:** A class is a blueprint or template that defines the structure and behavior of objects. It serves as a blueprint for creating objects of a specific type. In OOP, a class is a user-defined data type that encapsulates data and behavior.
- 2) **Objects and methods:** Objects are instances of a class representing real-world entities with states and behaviors. Each object has its own unique identity and can interact with other objects through methods and messages. Objects represent the runtime entities in a Java program.
- 3) **Inheritance:** Inheritance is a mechanism that allows a class (subclass or derived class) to inherit properties and behaviors from another class (superclass or base class). It promotes code reuse and hierarchical organization of classes, creating a parent-child relationship.
- 4) **Polymorphism:** Polymorphism is the ability of objects to take on multiple forms or have multiple behaviors based on their context. It enables flexibility and extensibility in the code. Polymorphism can be achieved through method overriding and method overloading.
- 5) **Abstraction:** Abstraction is the process of hiding the implementation details and showing only the relevant features of an object. It simplifies complex systems by focusing on essential aspects. Abstract classes and interfaces are used to achieve abstraction in Java.
- 6) **Encapsulation:** Encapsulation is the bundling of data (attributes or fields) and methods (functions) that operate on the data, restricting direct access from outside the class. It enhances data security and maintains data integrity by controlling access to class members.

### 3) Class:

- A class is a category or template of objects representing real-world entities or concepts.
- It acts as a blueprint for creating objects with specific properties and behaviors defined by class members (variables and methods).
- A class is a user-defined data type, and its objects are instances of that data type.
- A class does not occupy memory itself, but its objects do.

#### Syntax:

java

 Copy code


```
class ClassName {  
    // Class members (variables and methods)  
}
```

### 4) Methods:

- Methods are sets of code or instructions that perform specific tasks or operations.
- They encapsulate behavior and allow objects to perform actions and manipulate data.
- Methods define the behavior of objects and are essential for the interaction between objects.
- They promote code reusability and code optimization.

#### Syntax:

java

 Copy code


```
access-modifier return-type methodName(parameter-list) {  
    // Method body  
}
```

## 5) Objects:

- Objects are instances of a class representing real-world entities with their own identity, states (attributes), and behaviors (methods).
- They are created using the **new** keyword, which allocates memory for the object and returns its reference.
- Each object is independent and can interact with other objects to perform tasks.

### Syntax:

java

 Copy code

```
ClassName objectName = new ClassName();
```

- **Initializing Objects** : Objects can be initialized using reference variables or methods.

**Example 1 - Initializing by reference variable:**

java

```
class Animal {
    String color;
    int age;
}

public static void main(String[] args) {
    Animal buzo = new Animal();
    buzo.color = "black";
    buzo.age = 10;
    System.out.print(buzo.color + " " + buzo.age);
}

// Output: black 10
```

**Example 2 - Initializing by using a method:**

java

```
class Animal {
    String color;
    int age;

    void initObject(String C, int a) {
        color = C;
        age = a;
    }

    void display() {
        System.out.print(color + " " + age);
    }
}

public static void main(String[] args) {
    Animal buzo = new Animal();
    buzo.initObject("black", 10);
    buzo.display();
}

// Output: black 10
```

## 7) Advantages of OOP in Java:

- **Modularity**: OOP promotes dividing code into smaller, manageable modules (classes), making it easier to develop, maintain, and debug code. Each module represents a specific entity or functionality.
- **Code Reusability**: Inheritance and polymorphism allow code reuse, reducing redundant code and improving development efficiency. Reusing existing code saves time and effort in coding.
- **Encapsulation and Data Hiding**: Encapsulation helps protect data from unauthorized access, ensuring data integrity and security. Access to class members is controlled, and implementation details are hidden.
- **Flexibility and Extensibility**: Polymorphism enables flexibility by allowing objects to take multiple forms, facilitating code extension without modifying the existing codebase. New functionalities can be added without affecting existing code.
- **Abstraction and Simplification**: Abstraction hides complex implementation details, presenting only relevant features, and simplifies the understanding of the program's functionality. It focuses on what an object does rather than how it does it.

## 8) Conclusion:

Object-oriented programming in Java provides a powerful way to structure and organize code, making it easier to manage and maintain large-scale applications. It facilitates code reuse, abstraction, and encapsulation, leading to efficient and modular development. By utilizing the main concepts of OOP, developers can create robust, scalable, and easily maintainable software systems. Java's OOP features make it one of the most popular and widely used programming languages in the software development industry.

## 3. Constructor in Java

### 1) Definition:

- A constructor initializes an object immediately upon creation. It is a special method that contains a collection of statements (i.e., instructions) executed at the time of object creation. Constructors automatically initialize objects when they are created.
- Constructors are essential for initializing objects and providing initial values to their attributes. They allow us to set the initial state of objects at the time of creation. By using different types of constructors, we can create and initialize objects based on different requirements and scenarios in Java programs

### 2) Rules for Writing Constructors:

- A constructor must have the same name as the class.
- Constructors do not have any return type like **void**, **int**, etc.
- Access modifiers can be used in constructor declarations to control its accessibility in other classes.
- Constructors in Java cannot be abstract, final, static, or synchronized.

### 3) Representation of Constructor:

```
java Copy code

class Test {
    // This is a constructor
    public Test() {
        // code
    }
}
```

### 4) Need of Constructor:

The constructor is the first method that is invoked when you create an object. Suppose we have a class of Employees, and we need to include the name and ID of each employee. If we create objects without a constructor, we have to initialize each variable every time we add a new employee. This approach is not efficient, especially when dealing with a large number of employees. Constructors provide a solution to this problem.

### 5) Program for Constructor in Java:

```
java

class Employee {
    String name;
    int id;

    public Employee(String name, int id) {
        this.name = name;
        this.id = id;
        System.out.println("Name is " + name);
        System.out.println("Id is " + id);
    }

    public static void main(String[] args) {
        Employee e1 = new Employee("Ram", 10);
        Employee e2 = new Employee("Rahul", 20);
    }
}
```

csharp

```
Name is Ram
Id is 10
Name is Rahul
Id is 20
```

## 6) Types of Constructors:

We have three types of constructors in Java:

1. **Default Constructor (No-arguments):** Default constructors are automatically created by the compiler if not provided by the programmer. They provide default values to the object, like 0, null, etc., depending on the type.
2. **User-Defined Constructor (No-argument):** User-defined constructors are created by the programmer with no arguments.
3. **Parameterized Constructor:** A constructor that has parameters is known as a parameterized constructor. It allows us to initialize the class fields with custom values.

### 1) Default Constructor Example:

```
java Copy code

class Employee {
    String name;
    int id;

    public void display() {
        System.out.println(id + " " + name);
    }

    public static void main(String[] args) {
        Employee e1 = new Employee();
        Employee e2 = new Employee();
        e1.display(); // Output: 0 null
        e2.display(); // Output: 0 null
    }
}
```

### 2) User-Defined Constructor Example:

```
java Copy code

class Employee {
    public Employee() {
        System.out.println("I am user-defined constructor");
    }

    public static void main(String[] args) {
        Employee e1 = new Employee();
    }
}

Output: I am user-defined constructor
```

### 3) Parameterized Constructor Example:

```
java Copy code

class Employee {
    String name;

    public Employee(String name) {
        this.name = name;
        System.out.println(name);
    }

    public static void main(String[] args) {
        Employee e1 = new Employee("John");
    }
}

Output: John
```

# 4. Inheritance

## 1) Introduction:

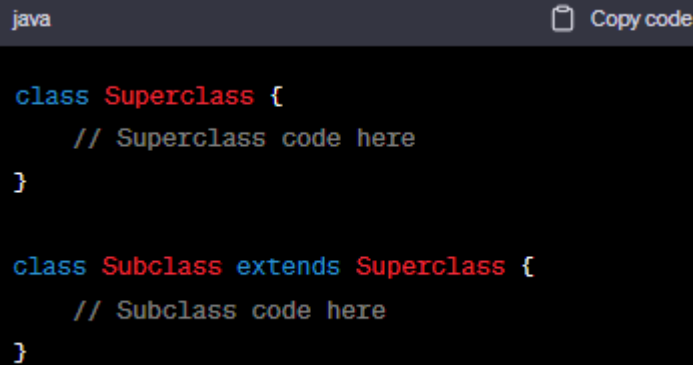
Inheritance is a critical feature of Object-Oriented Programming (OOP) in Java, where one class acquires the properties and behaviors of another class. It allows the creation of a new class (derived class or subclass) from an existing class (base class or superclass), enabling code reuse and promoting the concept of hierarchical organization in Java programs.

## 2) What is Inheritance?

Inheritance is a mechanism by which a class can acquire the properties and methods of another class. The class that is inherited from is called the superclass or base class, while the class that inherits is known as the subclass or derived class. Inheritance allows the subclass to extend or override the functionality of the superclass, facilitating code reuse and promoting a more organized and efficient code structure.

## 3) Syntax of Inheritance in Java:

The basic syntax for creating a subclass that inherits from a superclass in Java is as follows:



```
java Copy code

class Superclass {
    // Superclass code here
}

class Subclass extends Superclass {
    // Subclass code here
}
```

## 4) Importance of Java Inheritance:

- **Code Reusability:** Inheritance allows classes to inherit properties and methods from their parent classes, reducing the need for redundant code and making the codebase more maintainable and efficient.
- **Hierarchical Organization:** Inheritance facilitates the organization of classes in a hierarchical manner, creating a clear and structured class hierarchy. It promotes a better understanding of class relationships.
- **Polymorphism:** Inheritance, combined with method overriding, enables polymorphism, where a method can take different forms depending on the class it is called upon. This allows for dynamic binding at runtime.
- **Abstraction:** Inheritance promotes the abstraction of common attributes and behaviors into superclass, leading to more modular and scalable code. It allows developers to focus on the essential aspects of a class while inheriting the common functionality.



## 5) Types of Inheritance in Java:

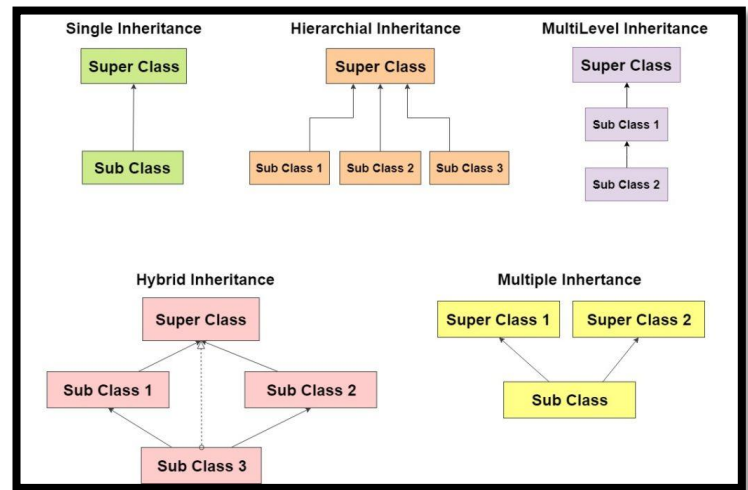
### 1) Single-level Inheritance:

1. In single-level inheritance, one subclass is derived from a single superclass.
2. It allows the subclass to inherit properties and methods from only one parent class.

```
java Copy code

class A {
    // Superclass code
}

class B extends A {
    // Subclass code
}
```



### 2) Multi-level Inheritance:

3. In multi-level inheritance, a chain of inheritance is formed with multiple classes.
4. One subclass inherits from a superclass, and then another subclass is derived from the previous subclass.

```
java Copy code

class A {
    // Superclass code
}

class B extends A {
    // Subclass code
}

class C extends B {
    // Subclass code
}
```

### 3) Hierarchical Inheritance:

5. Hierarchical inheritance involves multiple subclasses inheriting from a single superclass.
6. Multiple subclasses share the properties and methods of a common parent class.

```
java Copy code

class A {
    // Superclass code
}

class B extends A {
    // Subclass code
}

class C extends A {
    // Subclass code
}
```

### 4) Multiple Inheritance (Through Interfaces):

7. Java does not support multiple inheritance with classes, but it can be achieved through interfaces.
8. Multiple inheritance occurs when a subclass inherits from more than one parent class (interface)

```
java Copy code

interface A {
    // Interface code
}

interface B {
    // Interface code
}

class C implements A, B {
    // Subclass code
}
```

### 5) Hybrid Inheritance (Combination of Different Inheritance Types):

9. Hybrid inheritance is a combination of multiple inheritance and hierarchical inheritance.
10. It involves using interfaces and classes to achieve a complex inheritance structure.

```
java Copy code

interface A {
    // Interface code
}

class B {
    // Superclass code
}

class C extends B implements A {
    // Subclass code
}
```

# 5. Relationship Between Classes in Java

## 1) Association in Java:

Association in Java defines the connection between two classes that are set up through their objects. It manages one-to-one, one-to-many, and many-to-many relationships. In Java, the multiplicity between objects is defined by the Association, showing how objects communicate with each other and use each other's functionality and services.

## 2) Types of Relationships Managed by Association:

1. **One-to-One Relationship:** A person can have only one passport. It defines a one-to-one relationship.
2. **One-to-Many Relationship:** A college can have many students. It defines a one-to-many relationship.
3. **Many-to-One Relationship:** A state can have several cities, and those cities are related to that single state. It defines a many-to-one relationship.
4. **Many-to-Many Relationship:** A single student can associate with multiple teachers, and multiple students can also be associated with a single teacher. Both are created or deleted independently, so it defines a many-to-many relationship.

## 3) Types of Association in Java:

### 1) IS-A Association (Inheritance):

It is referred to as inheritance. It represents the relationship between classes where one class (subclass) inherits the properties and behaviors of another class (superclass).

### 2) HAS-A Association:

It represents a relationship where one class contains an object of another class. It is further classified into two parts:

- **Aggregation:** Represents a one-to-one or one-way relationship, where the aggregated object can exist independently of the main object.
- **Composition:** Represents a part-of relationship, where the composed object cannot exist independently and its lifecycle depends on the main object.

## 1) IS-A Association (Inheritance):

Inheritance represents the relationship between classes where one class (subclass or derived class) inherits the properties and behaviors of another class (superclass or base class). It allows the subclass to reuse the code of the superclass and extend its functionality.

```
java Copy code

class Animal {
    void eat() {
        System.out.println("Animal is eating.");
    }
}

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

public class InheritanceExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // This method is inherited from the Animal class
        dog.bark();
    }
}
```

In the above example, the **Dog** class is a subclass of the **Animal** class. The **Dog** class inherits the **eat()** method from the **Animal** class and extends its functionality by adding the **bark()** method.

## II. Composition:

Composition represents a part-of relationship, where the composed object cannot exist independently, and its lifecycle depends on the main object. If the main object is destroyed, the composed object will also be destroyed.

Example:

```
java Copy code

class Engine {
    void start() {
        System.out.println("Engine started.");
    }
}

class Car {
    private Engine engine;

    public Car() {
        engine = new Engine();
    }

    void startCar() {
        engine.start();
        System.out.println("Car started.");
    }
}

public class CompositionExample {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.startCar();
    }
}
```

In the above example, the **Car** class has a composition relationship with the **Engine** class. The **Car** class contains an object of the **Engine** class, and the existence of the **Engine** object is dependent on the **Car** object. When the **Car** object is created, the **Engine** object is also created, and when the **Car** object is destroyed, the **Engine** object will also be destroyed.

## 2) HAS-A Association:

HAS-A Association represents a relationship where one class contains an object of another class. It allows one class to use the functionalities and services provided by the other class by creating an object of it.

### I. Aggregation:

Aggregation represents a one-to-one or one-way relationship, where the aggregated object can exist independently of the main object. It means the lifetime of the aggregated object is not dependent on the main object.

In the above example, the **Employee** class has an aggregation relationship with the **Department** class. The **Employee** class contains an object of the **Department** class, but the existence of the **Department** object is independent of the **Employee** object.

```
class Department {
    private String deptName;

    public Department(String deptName) {
        this.deptName = deptName;
    }

    public String getDeptName() {
        return deptName;
    }
}

class Employee {
    private String empName;
    private Department department;

    public Employee(String empName, Department department) {
        this.empName = empName;
        this.department = department;
    }

    public String getEmpName() {
        return empName;
    }

    public Department getDepartment() {
        return department;
    }
}

public class AggregationExample {
    public static void main(String[] args) {
        Department hrDept = new Department("HR");
        Employee emp1 = new Employee("John", hrDept);

        System.out.println(emp1.getEmpName() + " works in "
+emp1.getDepartment().getDeptName() + " department.");
    }
}
```

# 6. Polymorphism:

Polymorphism is the process of performing the same action in different ways.  
It allows a single instance to have more than one form.

## 1) Real-life Example of Polymorphism:

Water can exhibit different forms at different places: solid, liquid, and gas. This is an example of polymorphism, where water possesses different behaviors in different situations.

## 2) Types of Polymorphism:

In Java, we have two types of polymorphism:

1. Compile-time Polymorphism
2. Run-time Polymorphism

### I. Compile-time Polymorphism:

Compile-time polymorphism is also known as static polymorphism, and it is handled by the compiler. It is achieved through method overloading in Java.

#### 1) Method Overloading:

Method overloading is when a class has multiple functions with the same name but different arguments. To overload a method in Java, it must follow one of the following conditions:

- i. Number of arguments should be different.
- ii. Sequence of arguments should be different.
- iii. Types of arguments should be different.

#### Examples of Compile-time Polymorphism:

##### I. With different numbers of arguments:

```
java
class Test {
    void show(int a) {
        System.out.println("1");
    }
    void show() {
        System.out.println("2");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.show(5); // It will call show(int a)
    }
}
```

##### II. With different sequences of arguments:

```
java
class Test {
    void show(int a, String b) {
        System.out.println("1");
    }
    void show(String a, int b) {
        System.out.println("2");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.show(1, "Ram"); // It will call show(int a, String b)
        t.show("Rahul", 2); // It will call show(String a, int b)
    }
}
```

##### III. With different types of arguments:

```
java
class Test {
    void show(int a) {
        System.out.println("1");
    }
    void show(String b) {
        System.out.println("2");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.show("Ram"); // It will call show(String b)
    }
}
```

## II. Run-time Polymorphism:

Run-time polymorphism is also known as dynamic polymorphism, and it is achieved through method overriding in Java.

### 2) Method Overriding:

Method overriding allows providing a specific implementation of the same method in the child class that is already provided in the superclass. For a method to be overridden in Java, it must have the following conditions:

1. Number of arguments should be the same.
2. Sequence of arguments should be the same.
3. Types of arguments should be the same.
4. There must be an IS-A relationship (Inheritance) between the classes.

#### Examples of Run-time Polymorphism:

<div><h4>I. With no arguments:</h4><pre>java  class Test {     void show() {         System.out.println("Parent class function called");     } }  class Paper extends Test {     void show() {         System.out.println("Child class function called");     }      public static void main(String[] args) {         Test t = new Test();         t.show(); // It will call the parent class's show()         Paper p = new Paper();         p.show(); // It will call the child class's show()     } }</pre></div>	<div><h4>I. With the same type of arguments:</h4><pre>java <span>Copy code</span>  class Test {     void show(int a) {         System.out.println("Parent class function called");     } }  class Paper extends Test {     void show(int a) {         System.out.println("Child class function called");     }      public static void main(String[] args) {         Test t = new Test();         t.show(1); // It will call the parent class's show(int a)         Paper p = new Paper();         p.show(2); // It will call the child class's show(int a)     } }</pre></div>
<div><h4>I. With the same sequence of arguments:</h4><pre>1- class Test { 2-     void show(String a, int b) { 3-         System.out.println("Parent class function called"); 4-     } 5- } 6- 7- class Paper extends Test { 8-     void show(String a, int b) { 9-         System.out.println("Child class function called"); 10-    } 11- 12-    public static void main(String[] args) { 13-        Test t = new Test(); 14-        t.show("Hello", 1); // It will call the parent class's show(String a, int b) 15-        Paper p = new Paper(); 16-        p.show("Hi", 2); // It will call the child class's show(String a, int b) 17-    } 18- }</pre></div>	

# 7. Abstraction:

- Abstraction is the process of hiding the implementation or details and showing the main services or interfaces to the user. It deals with exposing the interfaces to the user and hiding their implementations.
- Abstraction helps in designing more modular and flexible systems by hiding unnecessary details and exposing only relevant functionalities. It enables programmers to focus on essential aspects of a program's behavior and helps in achieving code reusability and maintainability.
- **Real-life Example of Abstraction:** In the case of a car, the driver interacts with relevant parts like the steering wheel, gear, horn, accelerator, and brakes, as they are necessary for driving. However, the driver need not understand the internal functioning of the engine, gear system, brakes, etc. This concept of showing the interface to the user (driver) and hiding the implementation is called Abstraction.

## 3) In Java, Abstraction is achieved by two ways:

1. Abstract class (and abstract method)
2. Interface

### a. Abstract class (and abstract method)

- **Abstract class:** An abstract class is a class that is declared with the **abstract** keyword.
  1. An abstract class can have both abstract methods and concrete methods.
  2. Any class that has abstract methods must be declared with the **abstract** keyword.
  3. An abstract class cannot be directly instantiated using the **new** keyword, meaning we cannot create objects of an abstract class.
  4. If any regular class extends an abstract class, then the class must implement all the abstract methods of the abstract parent class.
- **Abstract method:** An abstract method is a method that is declared without an implementation.
  1. A method defined as abstract needs to be redefined (overridden) in the subclass, making overriding compulsory for concrete subclasses.
  2. An abstract method must be in an abstract class, but the inverse is not true.

Program to illustrate Abstraction in Java:

```
java Copy code

abstract class Vehicle {
    abstract void start();
}

class Car extends Vehicle {
    void start() {
        System.out.println("car starts with key");
    }
}

class Scooter extends Vehicle {
    void start() {
        System.out.println("scooter starts with kick");
    }
}

public static void main(String[] args) {
    Car c = new Car();
    c.start();

    Scooter s = new Scooter();
    s.start();
}
```

Output:

```
vbnet Copy code

car starts with key
scooter starts with kick
```

## 8. Interfaces:

- Interfaces in Java are similar to classes.
- They also contain methods and fields, but all methods in an interface are by default abstract.
- Interfaces act as blueprints for classes, specifying what a class must do without explaining how to do it.
- Interfaces in Java are powerful tools for achieving abstraction, multiple inheritance, and designing loosely coupled systems.
- They allow classes to provide specific implementations for the methods defined in an interface, promoting code reusability and flexibility.

### 4) Here are some important points about interfaces:

1. Interface is the blueprint of a class.
2. By default, all the methods in an interface are **public** and **abstract**.
3. By default, the compiler sets **public static final** before fields in an interface, making them constants.
4. The **interface** keyword is used to declare an interface.
5. Interfaces provide total abstraction, as all methods are abstract by default.
6. Any class that implements an interface must implement all the abstract methods declared in the interface.
7. The **implements** keyword is used to implement an interface in a class.
8. We cannot create objects of an interface, similar to abstract classes.
9. Multiple Inheritance can be achieved through interfaces in Java.
10. Interfaces can be used to achieve loose coupling between classes.

### 5) Syntax for Interface:

```
java Copy code

interface Interface_name {
    // Declare constant fields
    // Declare methods that are abstract by default
}
```

### 6) Program to demonstrate the working of an interface:

```
java Copy code

interface I1 {
    void show();
}

class Test implements I1 {
    public void show() {
        System.out.println("Interface function is implemented");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.show();
    }
}
```

Output:

```
vbnet Copy code

Interface function is implemented
```

### 7) Multiple Inheritance through Interface:



```
java

interface I1 {
    void show();
}

interface I2 {
    void display();
}

class Test implements I1, I2 {
    public void show() {
        System.out.println("I1 is implemented");
    }

    public void display() {
        System.out.println("I2 is implemented");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.show();
        t.display();
    }
}
```

Output:

```
csharp Copy code

I1 is implemented
I2 is implemented
```

## 9. Encapsulation:

- Encapsulation is the process of wrapping or binding data (variables) and code (methods) acting on the data in a single unit.
- It is also known as data-hiding because data is hidden from other classes and can only be accessed through member functions of the class itself.
- A common analogy for encapsulation is a medicine capsule that contains several medicines hidden inside.
- Encapsulation allows for better control over data access and manipulation, as it provides a level of abstraction and hides the internal implementation details of a class.
- By using getter and setter methods, you can enforce data validation rules, ensuring that data is accessed and modified correctly.

### 8) Encapsulation in Java:

- Variables
- Methods
- Class

### 9) Important points about Encapsulation:

- Encapsulation is also known as data-hiding because data is hidden from other classes.
- Hidden data can only be accessed by member functions of its own class and not by other classes.
- Java Bean class is an example of a fully encapsulated class.
- Encapsulation is used to achieve proper security for data, preventing it from being modified directly.
- Encapsulation in Java can be achieved by:
  1. Making all the members of the class private.
  2. Using setter and getter methods to set and get the data.

### 10) Illustration of Encapsulation:

```
java Copy code

class Employee {
    private String name; // Data-hiding

    public void setName(String name) // for setting the data
    {
        this.name = name;
    }

    public String getName() // for getting the data
    {
        return name;
    }
}
```

### 11) Program for Encapsulation:

```
java Copy code

class Employee {
    private int empId;

    public void setEmpId(int eid) {
        empId = eid;
    }

    public int getEmpId() {
        return empId;
    }
}

class Company {
    public static void main(String[] args) {
        Employee e = new Employee();
        e.setEmpId(101);
        System.out.println(e.getEmpId());
    }
}
```

```
Output:

Copy code

101
```



## 10. This Keyword:

- The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity
- The "this" keyword is a valuable tool in Java that helps in differentiating between local variables and instance variables, as well as in simplifying method and constructor calls within the same class.

### Usage of This keyword:

#### 1. Refer to the current class instance variable:

- When a local variable in a method has the same name as an instance variable, the "this" keyword can be used to distinguish between the two.

#### 2. Invoke the current class method (implicitly):

- The "this" keyword can be used to invoke other methods within the same class, even if they are not directly called from the main method.

#### 3. Invoke the current class constructor:

- The "this" keyword can be used to call another constructor of the same class from within a constructor.

#### 4. Pass as an argument in method calls:

- The "this" keyword can be passed as an argument to other methods.

#### 5. Pass as an argument in constructor calls:

- The "this" keyword can be passed as an argument to other constructors.

#### 6. Return the current class instance from a method:

- A method can return the current instance of the class using the "this" keyword.

## Examples of Usage of This keyword:

### 1. Refer to the current class instance variable:

```
java Copy code

class Test1 {
    int i;

    void setValues(int i) {
        this.i = i;
    }

    void show() {
        System.out.print(i);
    }
}
```

### 2. Invoke the current class method (implicitly):

```
java Copy code

class ThisDemo {
    void display() {
        System.out.print("hello");
    }

    void show() {
        this.display();
    }
}
```

### 3. Invoke the current class constructor:

```
java

class ThisDemo {
    ThisDemo() {
        System.out.print("no argument constructor");
    }

    ThisDemo(int a) {
        this();
        System.out.print("parameterized constructor")
    }
}
```

### 4. Pass as an argument in method calls:

```
java

class ThisDemo {
    void m1(ThisDemo td) {
        System.out.print("I'm in m1 method");
    }

    void m2() {
        m1(this);
    }
}
```

### 5. Pass as an argument in constructor calls:

```
java

class Test {
    Test(ThisDemo td) {
        System.out.print("test class constructor");
    }

    public static void main(String[] args) {
        ThisDemo t = new ThisDemo();
        Test tst = new Test(t);
    }
}
```

### 6. Return the current class instance from a method:

```
java

class ThisDemo {
    ThisDemo m1() {
        return this;
    }

    public static void main(String[] args) {
        ThisDemo t = new ThisDemo();
        t.m1();
    }
}
```

# 11. Super Keyword:

- The "super" keyword is a reference variable that is used to refer to the immediate parent class object in Java.
- It is primarily used with the concept of inheritance to access parent class objects and members from the child class.
- The "super" keyword is a powerful tool in Java when dealing with inheritance. It allows you to access and use the features of the parent class within the child class, promoting code reusability and maintaining the relationship between classes.

## 12) Uses of 'Super' keyword:

1. "Super" keyword can be used to refer to the immediate parent class instance variables.
2. "Super" keyword can be used to invoke immediate parent class methods.
3. "Super()" can be used to invoke the immediate parent class constructor.

## 13) Examples:

1. Super keyword for parent class instance variables:

```
java

class A {
    int i = 10;
}

class B extends A {
    void show(int i) {
        System.out.println(i);
        System.out.println(super.i);
    }

    public static void main(String[] args) {
        B obj = new B();
        obj.show(20);
    }
}
```

Output:

```
20
10
```

2. Super keyword to invoke parent class methods:

```
java

class A {
    void m1() {
        System.out.println("I am in class A");
    }
}

class B extends A {
    void show() {
        System.out.println("I am in class B");
        super.m1();
    }

    public static void main(String[] args) {
        B obj = new B();
        obj.show();
    }
}
```

Output:

```
I am in class B
I am in class A
```

3. Super() to invoke parent class constructor:

```
java Copy code

class A {
    A() {
        System.out.println("I am A class constructor");
    }
}

class B extends A {
    B() {
        super();
        System.out.println("I am B class constructor");
    }

    public static void main(String[] args) {
        B obj = new B();
    }
}
```

Output:

```
css Copy code

I am A class constructor
I am B class constructor
```

## 12. Final Keyword:

- The "final" keyword in Java is used to put restrictions on entities like variables, methods, or classes.
- The "final" keyword is useful when we want to ensure that certain entities, like variables, methods, or classes, should not be modified or overridden, providing better control and security in Java code.

### ***1) It can be used in different contexts such as:***

1. Final Variable
2. Final Method
3. Final Class

## III. Final Variable:

When we declare a variable in Java with the "final" keyword, we can't modify the value of that variable in further coding. The value in the variable becomes a constant, and such variables are called constant variables. Final variables with no value assigned are called blank final variables.

```
java Copy code

class Test {
    public static void main(String[] args) {
        final int i = 10;
        i = i + 20; // Compile-time error
        System.out.println(i);
    }
}
```

## IV. Final Method:

When we make any method final, we can't override it in the child class. If we want to prevent any method from being overridden, we can make it final.

### **Program for the Final Method:**

```
java Copy code

class Demo {
    final void m1() {
        System.out.println("I am in m1 method");
    }
}

class Test extends Demo {
    void m1() {
        System.out.println("I am in overridden m1 method");
    }

    public static void main(String[] args) {
        Demo d = new Demo();
        d.m1(); // Output: I am in m1 method
    }
}
```

## V. Final Class:

When we don't want to allow our class to be inherited by other classes, we can make the class final.

### **Program for the Final Class:**

```
java

final class Demo {
    final void m1() {
        System.out.println("I am in m1 method");
    }
}

class Test extends Demo {
    void m1() {
        System.out.println("I am in overridden m1 method");
    }

    public static void main(String[] args) {
        Demo d = new Demo();
        d.m1(); // Compile-time error
    }
}
```

# 13. Static Keyword:

- The "static" keyword in Java is a non-access modifier that can be used with variables, methods, blocks, and inner classes. It is associated with the class rather than with the instances (objects) of the class. Variables or methods declared as static are shared among all instances of the class.
- The "static" keyword enables the creation of shared resources, provides memory management benefits, and allows direct access to class-level elements without creating instances. However, it should be used carefully, as excessive use of static elements can lead to code complexity and reduced flexibility in some cases.

## 2) Usage of the static keyword:

### 1. Static keyword can be used with class-level variables:

Static variables are associated with the class and not with individual objects. They are shared among all instances (objects) of the class.

### 2. Static keyword can be used with methods:

Static methods belong to the class rather than the object. They can be called directly using the class name without creating an instance of the class.

### 3. Static keyword can be used with blocks:

Static blocks are used for static initialization of variables. They are executed only once when the class is loaded into memory.

### 4. Static keyword can be used in inner classes:

Inner static classes are static nested classes that do not require an instance of the outer class to be instantiated.

### 5. Memory Management Benefits:

Since static variables are shared among all instances of the class, this allocation occurs only once, reducing memory consumption

## 3) Examples:

### 1. Static variables and methods:

```
java Copy code

class Employee {
    int empid;
    String name;
    static String company = "Smart Programming";

    Employee(int empid, String name) {
        this.empid = empid;
        this.name = name;
    }

    void display() {
        System.out.println(empid + " " + name + " " + company);
    }

    public static void main(String[] args) {
        Employee e1 = new Employee(101, "Amit");
        e1.display();
        Employee e2 = new Employee(102, "Deepak");
        e2.display();
    }
}
```

Output:

Output:

```
Copy code

101 Amit Smart Programming
102 Deepak Smart Programming
```

In this example, **company** is a static variable shared among all instances of the **Employee** class. Both **e1** and **e2** access the same **company** value "Smart Programming."

# 14. Static Method in Java:

- A static method in Java is a method that is associated with the class rather than with instances (objects) of the class.
- It can be called directly using the class name without creating an instance of the class.
- Static methods are useful for utility functions or operations that do not depend on the state of any object.
- Static methods are particularly useful for utility functions, mathematical calculations, or other operations that do not depend on object-specific data.
- They allow the programmer to organize methods that are not tied to any specific instance and can be accessed directly from the class level.

## 4) Rules for Static Methods:

1. **Static Method Accessibility:** A static method can be called without using the class name, but only within the class's scope. It can be accessed directly using the class name within the same class.
2. **Access to Data:** A static method can only access static data (class-level data) and cannot access instance data (non-static data). This is because static methods are not associated with any particular object instance and only operate on class-level data.
3. **Call to Other Methods:** A static method can call only other static methods and cannot call a non-static method. This is because non-static methods depend on the state of the object and cannot be called without an instance.
4. **No Access to this and super:** A static method cannot use the "this" or "super" keywords since they refer to specific instances of objects, and static methods are not tied to any instance.
5. **Memory Management:** Static methods are also used for memory management, as they are associated with the class and not with individual objects. They do not require an instance to be created and can be called directly using the class name.

### Example of Static Method:

```
java Copy code

class Test {
    static void display() {
        System.out.println("1");
    }

    public static void main(String[] args) {
        Test.display();
    }
}
```

Output:

```
Copy code

1
```

In this example, the **display()** method is a static method that can be called directly using the class name **Test.display()**, without creating an instance of the **Test** class.

### 2. A static method can call only other static methods and cannot call a non-static method:

```
java Copy code

class StaticDemo {
    static int i = 10;

    static void display() {
        show();
        System.out.println("1");
    }

    static void show() {
        System.out.println("2");
    }
}
```

In this example, the static method **display()** can call the static method **show()** successfully, as both methods are static. However, it cannot directly call a non-static method, as non-static methods depend on the state of the object, and static methods are not associated with any instance. The method **display()** can print "2" by calling the **show()** method and print "1" separately.

### 1. A static method can access only static data. It cannot access non-static data (Instance data):

```
java Copy code

class StaticDemo {
    static int i = 10;

    static void display() {
        System.out.println(i);
    }
}
```

In this example, the static method **display()** can access the static variable **i**, but it cannot access any non-static (instance) data. The method can only access and display the value of the static variable **i**.

# 15. Static Block:

- A static block in Java is a block of code that is marked with the "static" keyword.
- It is a special type of block that is executed when the class is loaded into memory.
- Static blocks are used to initialize static variables or perform any static initialization tasks.

## 1) Important points about static block:

1. **Execution of Static Blocks:** Static blocks are executed automatically when the class is loaded into memory. They are executed only once, during the class loading process, before any other methods or constructors are called.
2. **Multiple Static Blocks:** A class can have multiple static blocks. They are executed in the same sequence in which they appear in the code.
3. **No Need for Main Method:** Static blocks can be executed without the main method, but this is possible only in JDK 1.6 and earlier versions. Starting from JDK 1.7, a main method is required to execute a Java program.
4. **Initialization and One-time Execution:** The code inside static blocks is used for static initialization tasks, like initializing static variables, loading resources, or setting up the environment. The code is executed only once when the class is loaded.
5. **Static Blocks vs. Constructors:** Static blocks are executed before constructors. They are used for tasks that need to be performed before any instance of the class is created.

## 2) Declaration of static block:

```
java Copy code  
  
class StaticBlockDemo {  
    static {  
        System.out.println("Hello, I am a static block!");  
    }  
  
    public static void main(String[] args) {  
        System.out.println("I am the main method.");  
    }  
}
```

## 3) Program for the Static Block:

```
java Copy code  
  
class StaticBlockDemo {  
    static {  
        System.out.println("I am static block 1");  
    }  
  
    static {  
        System.out.println("I am static block 2");  
    }  
  
    public static void main(String[] args) {  
        System.out.println("I am the main method.");  
    }  
}
```

Output:

```
css Copy code  
  
I am static block 1  
I am static block 2  
I am the main method.
```

In this example, the two static blocks are executed automatically when the class StaticBlockDemo is loaded into memory, and their code is executed only once.