GLOBAL
INSTITUTE OF TECHNOLOGY AND MANAGEMENT
FARRUKHNAGAR, GURUGRAM
Approved by AICTE, Govt. of India & Affiliated to Gurugram University (Haryana)

JAVA NOTES

GLOBAL INSTITUTE OF TECHNOLOGY AND MANAGEMENT

UNIT 2

java CSE 4th Sem

Introduction to Java

DESIGNED BY

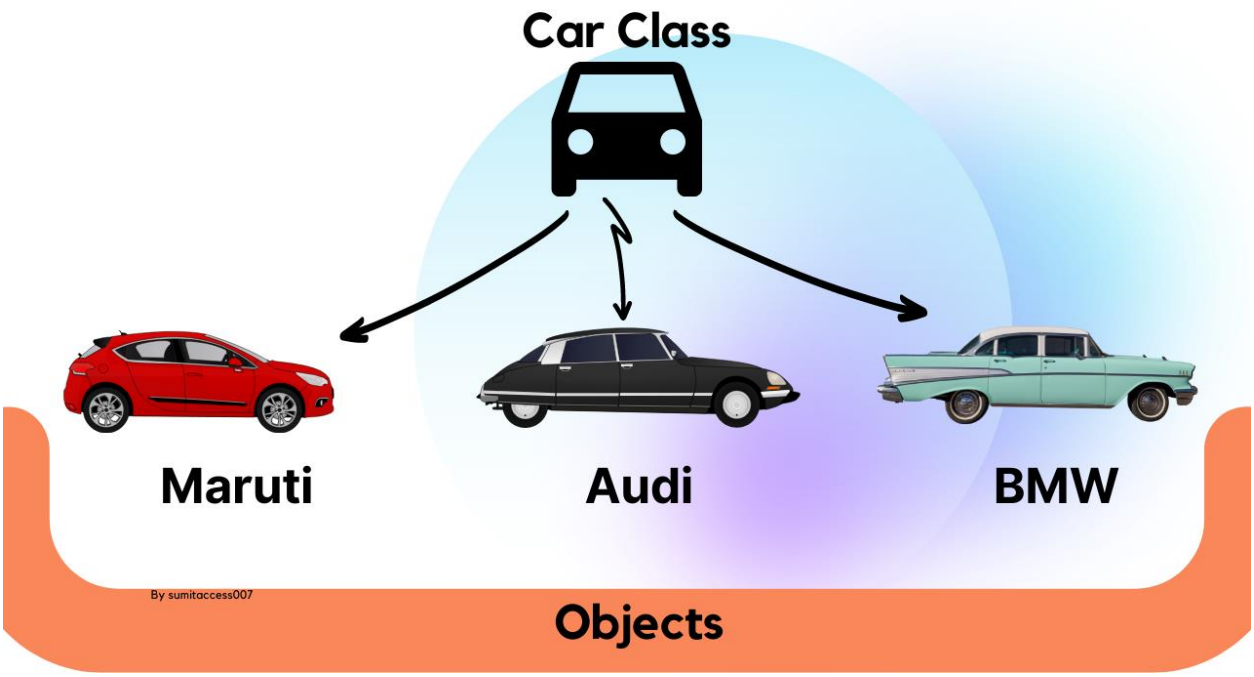
Assistant Professor

Muzamil Aslam

◆ 1. What is a Class in Object-Oriented Programming?

A **class** is a user-defined data type that acts as a **blueprint** or **template** for creating objects. It groups **data members (attributes)** and **member functions (methods)** under a single unit.

Classes help achieve modularity and code reusability in programs. All real-world entities (like cars, students, or accounts) can be modeled as classes in programming.



Example:

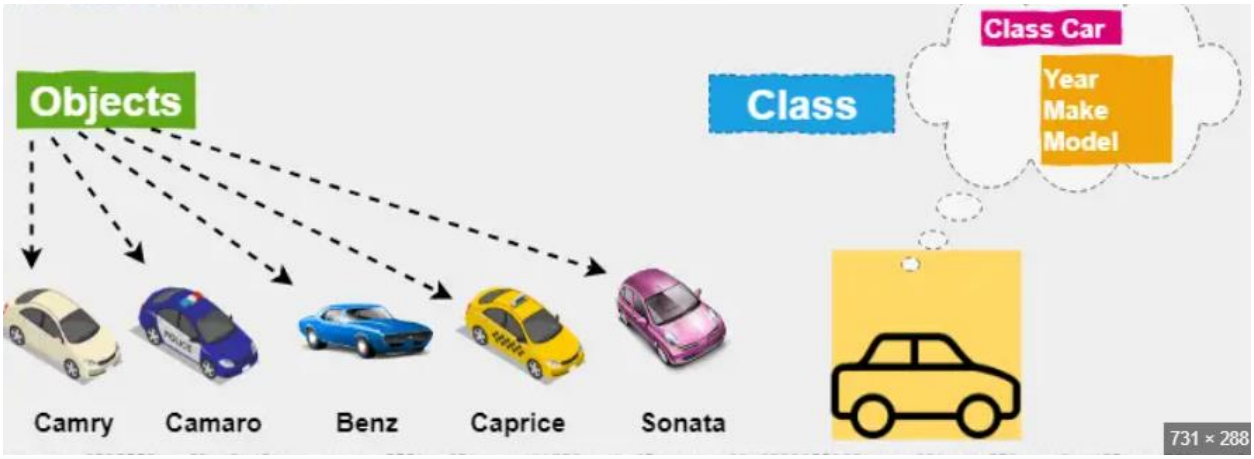
```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
```

Here, `Car` is a class with two attributes (`brand` and `model`) and one constructor method.

◆ 2. What is an Object in OOP?

An **object** is an **instance** of a class. When a class is defined, no memory is allocated. Only when an object is created, memory is allocated for its attributes.

Objects have **state** (data stored in attributes) and **behavior** (functions or methods).



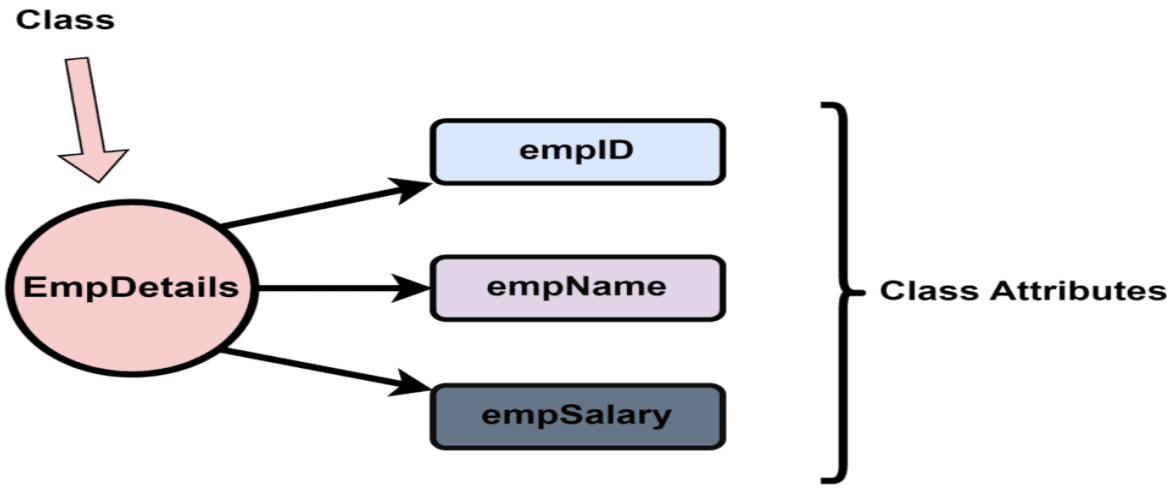
Example:

```
car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")
```

Here, `car1` and `car2` are two different objects of the class `Car`, each with its own state.

◆ 3. What are Attributes in a Class?

Attributes are the **variables** that hold data related to the object. They define the **state** of an object.



There are two types of attributes:

- **Instance Attributes**: Unique to each object and defined using `self` inside methods.
- **Class Attributes**: Shared by all instances; defined outside methods but inside the class.

□ Example:

```
class Student:
    college = "ABC Institute"  # class attribute

    def __init__(self, name, roll_no):
        self.name = name          # instance attribute
        self.roll_no = roll_no
```

Here, `name` and `roll_no` are instance attributes, and `college` is a class attribute.

◆ 4. What are Methods in a Class?

Methods are **functions** defined inside a class that describe the **behavior** of the objects. These are used to perform operations using the object's data.

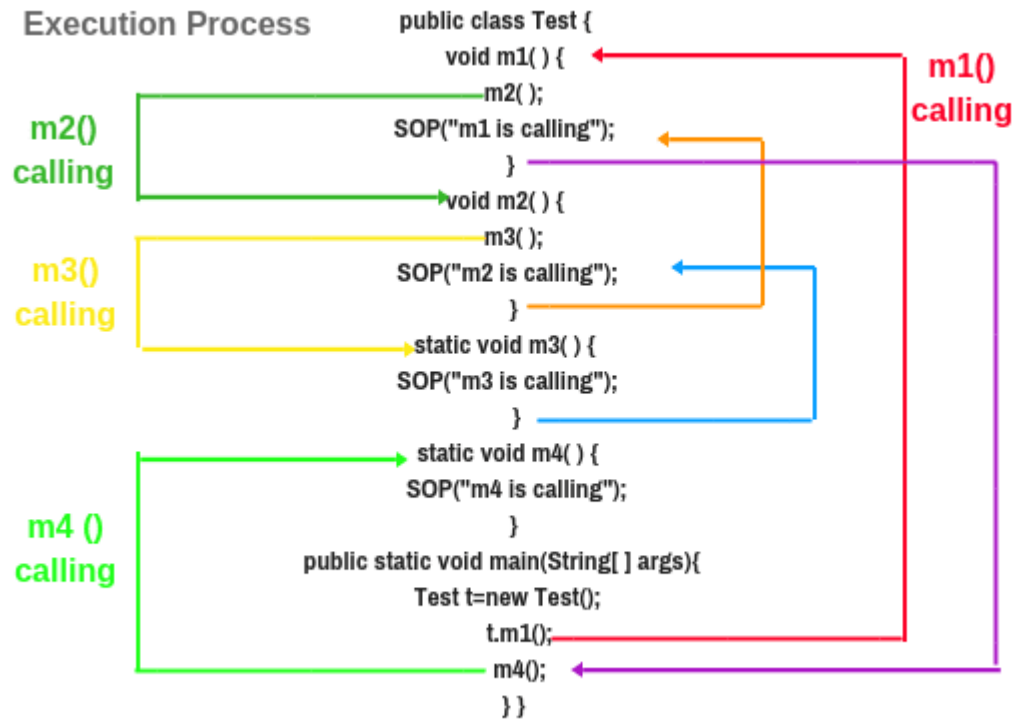


Fig: Calling of Instance and Static method

There are different types of methods:

- **Instance Methods**: Use `self` to access instance attributes.
- **Class Methods**: Use `@classmethod` and access class attributes.
- **Static Methods**: Use `@staticmethod` and don't access any class or instance data.

□ Example:

```
class Calculator:
    def add(self, a, b):
        return a + b

    def multiply(self, a, b):
        return a * b
```

These methods define behavior that the `Calculator` class can perform.

◆ 5. What is Data Encapsulation in OOP?

Encapsulation is one of the four main principles of OOP. It refers to the **bundling of data and methods** within a single unit (class) and **restricting direct access** to the internal details of an object.

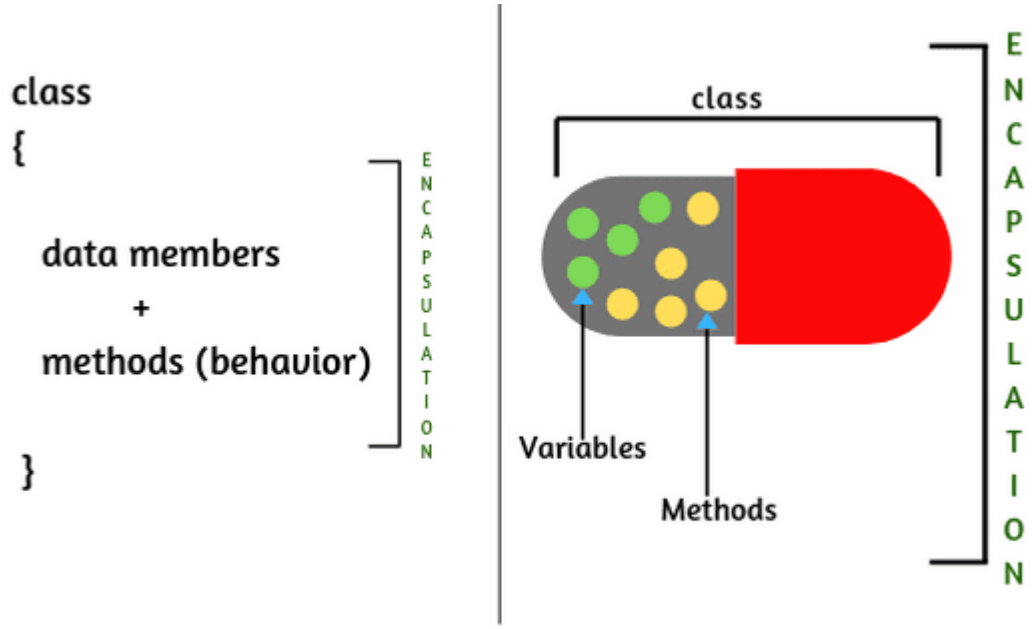


Fig: Encapsulation

This is done by declaring attributes as **private** or **protected**, and accessing them through **getter/setter methods**.

Encapsulation improves **security**, **data hiding**, and **maintainability**.

Example:

```
class BankAccount:
    def __init__(self, holder, balance):
        self.holder = holder
        self.__balance = balance  # private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
        return self.__balance
```

- `__balance` is a **private** attribute.
- It cannot be accessed directly from outside the class.
- It can only be accessed using the `get_balance()` method.

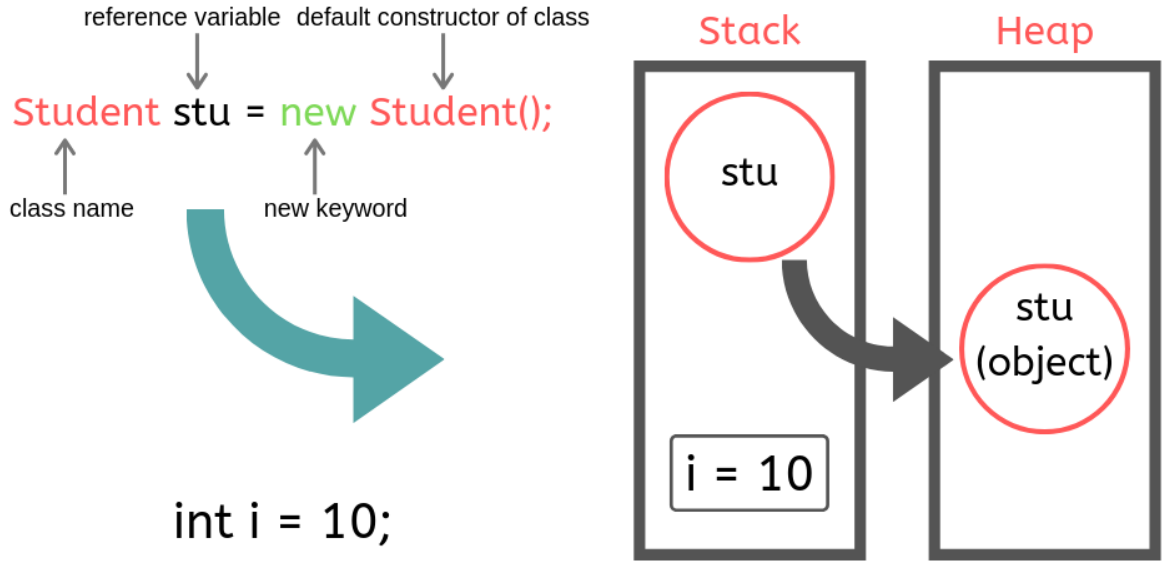
◆ Conclusion:

Concept	Description
Class	Blueprint for creating objects. Groups data and behavior.
Object	Instance of a class. Represents a real-world entity.
Attributes	Variables storing object's state. Can be instance-level or class-level.
Methods	Functions inside a class. Define the behavior of objects.
Encapsulation	Restricts access to internal data for security and abstraction.

1. Reference Variables

📘 Definition:

A **reference variable** is a variable that refers to the memory address of an object. In object-oriented languages like Java, objects are **not stored directly in variables**, instead, **reference variables hold the address of the object in memory**.



In simple terms, it's like a **remote control** that controls an object (TV). You don't hold the TV itself — you hold the remote to interact with it.

✅Example in Java:

```
class Student {
  String name;
  int rollNo;
}

public class Main {
  public static void main(String[] args) {
    Student s1 = new Student(); // s1 is a reference variable
    s1.name = "Amit";
    s1.rollNo = 101;

    Student s2 = s1; // s2 is also referring to the same object
    System.out.println(s2.name); // Output: Amit
  }
}
```

💡 Key Points:

- Multiple reference variables can refer to the same object.
- Changes made using one reference reflect in others.
- Reference variables **do not hold actual object data**, they hold the **memory address**.

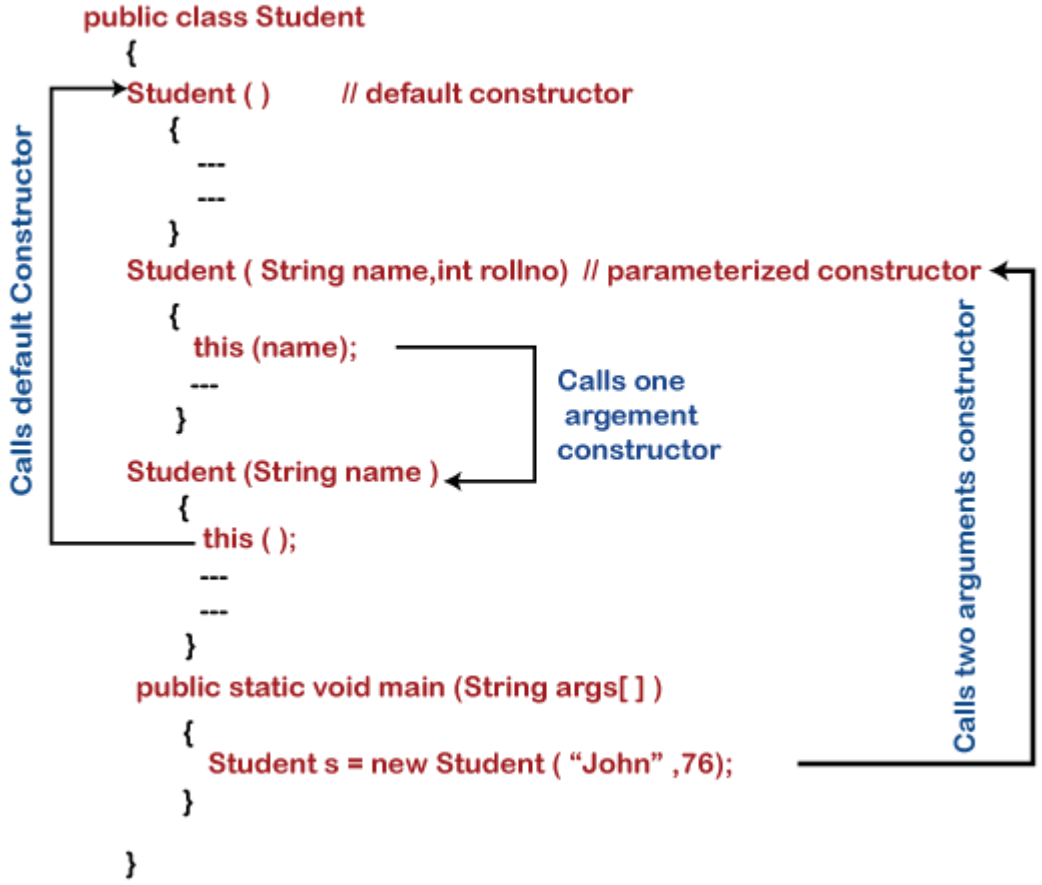
◆ 2. Constructors

📘 Definition:

A **constructor** is a special method in a class that is **automatically called when an object is created**. It has the **same name as the class** and **no return type** (not even void).

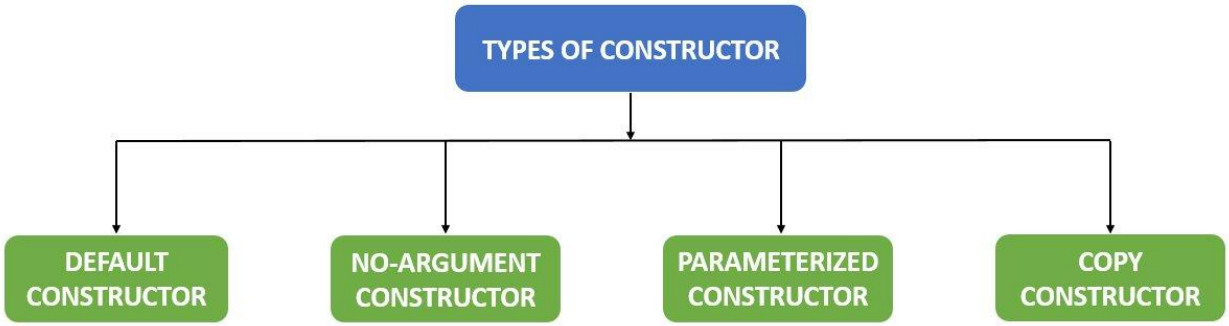
Constructors are used to **initialize objects** — that is, to set their initial values.

Constructor Chaining in Java



✓Types of Constructors in Java:

1. **Default Constructor** – No parameters
2. **Parameterized Constructor** – Takes arguments
3. **Copy Constructor** – Not built-in in Java, but can be manually created



✅Example:

```
class Car {
   String brand;
   int price;

   // Constructor
   Car(String b, int p) {
      brand = b;
      price = p;
   }

   void show() {
      System.out.println("Brand: " + brand + ", Price: " + price);
   }
}

public class Main {
   public static void main(String[] args) {
      Car c1 = new Car("Toyota", 1000000);  // Constructor called
      c1.show();
   }
}
```

💡 Key Points:

- Constructor has the **same name** as the class.
- Used to **initialize** object attributes at the time of object creation.
- If no constructor is defined, **Java provides a default constructor** automatically.
- Constructors can be **overloaded**.

◆ 3. Anonymous Blocks (Instance Initializer Blocks)

📘 Definition:

An **anonymous block**, also called an **instance initializer block**, is a block of code enclosed in `{}` that is **executed every time an object is created**, **before the constructor**.

It is used when you want to write **common initialization code** for all constructors.

✅Example:

```
class Test {
  {
    System.out.println("This is an instance initializer block.");
  }

  Test() {
    System.out.println("Constructor called.");
  }
}

public class Main {
  public static void main(String[] args) {
    Test t1 = new Test();
    Test t2 = new Test();
  }
}
```

🔍 Output:

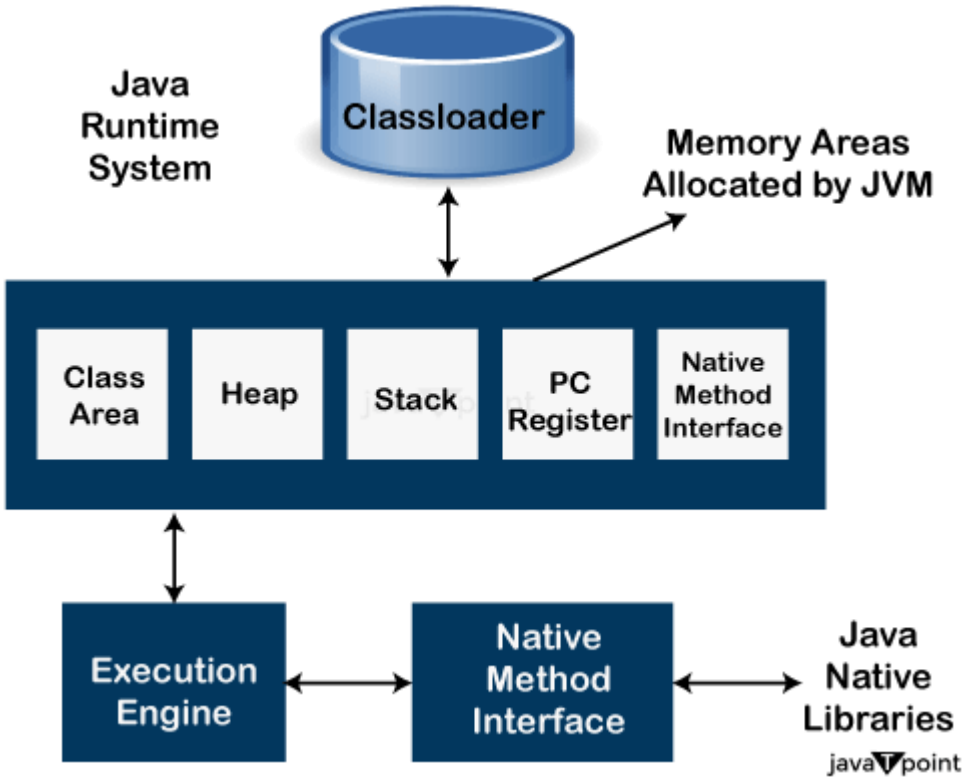
```
This is an instance initializer block.
Constructor called.
This is an instance initializer block.
Constructor called.
```

💡 Key Points:

- Executed **before the constructor**.
- Used for **common initialization tasks**.
- Useful when multiple constructors exist and you don't want to duplicate code.
- Not frequently used in everyday programming, but important for understanding Java object lifecycle.

◆ Java Memory Structure

Java uses a well-defined memory model to **manage the lifecycle of variables, objects, methods**, and **class definitions**. It is divided into several parts, each serving a specific purpose in memory management.



The primary areas are:

- **Stack Area**
- **Heap Area**
- **Method Area (also known as Class Area or Metaspace)**
- **Program Counter Register (optional for exam focus)**

Let's explore each in detail:

◈ 1. Stack Area

📓 Definition:

The **stack area** stores:

- **Local variables**

- **Method call information**
- **References to objects in the heap**

Each thread in Java has its own **stack**, which holds **frames**. Each frame corresponds to a method call.

✅Key Characteristics:

- Memory is **allocated and deallocated automatically**.
- **Faster** than heap memory.
- Data is stored in **LIFO** (Last In, First Out) order.
- Only **primitive values** and **object references** are stored (not the actual objects).

Example:

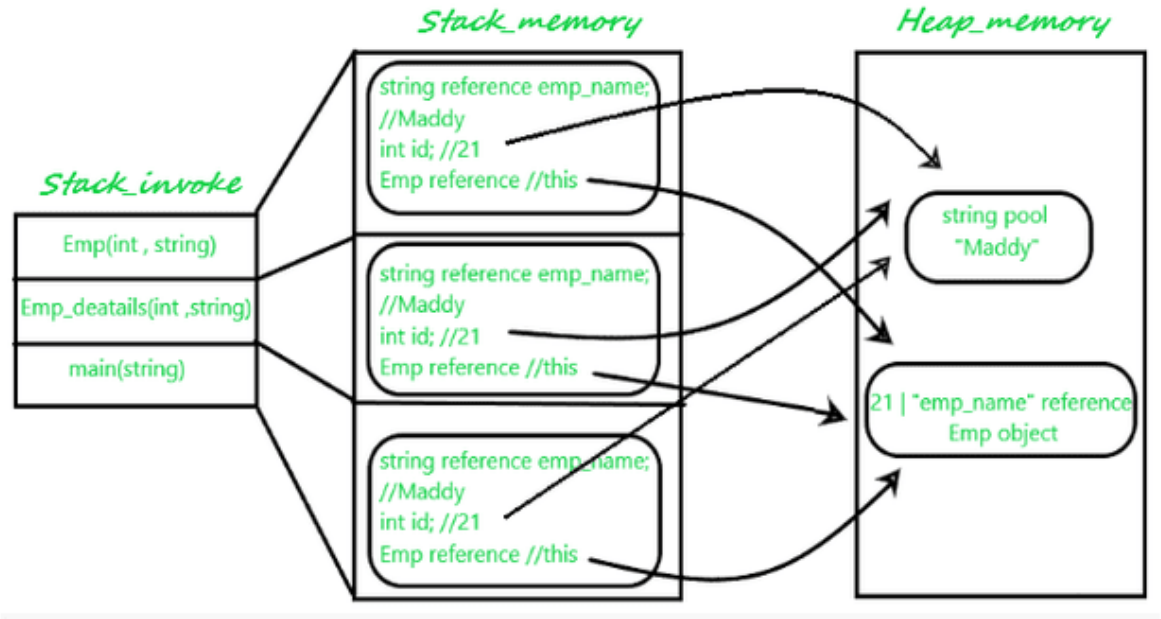
```
public class Example {
    public static void main(String[] args) {
        int a = 10;           // Stored in stack
        String name = "Ali";  // Reference stored in stack, actual object in
heap
    }
}
```

---

2. Heap Area

Definition:

The **heap** is the runtime data area where **objects are allocated**. When you use the `new` keyword, memory is allocated in the heap.



✅Key Characteristics:

- Shared among all threads.
- Stores **objects** and **class instances**.
- **Garbage collected** (unused objects are cleaned automatically).
- Slower access compared to stack.

□ Example:

```
Student s1 = new Student();  // 's1' is in stack, object is in heap
```

◆ 3. Method Area (Class Area or Metaspace in Java 8+)

📚 Definition:

The **method area** (also known as **class area** or **Metaspace**) stores:

- Class bytecode
- Static variables
- Method metadata
- Constant pool (symbolic references)

This area is used to store **information about the classes and methods** used in the application.

✅Key Characteristics:

- Shared among all threads.
- Stores **class-level data** (e.g., static members).
- Also contains **constructors**, **method data**, and **type information**.

🧪 Example:

```
class Car {
    static String brand = "BMW";  // Stored in method/class area
}
```

🔷 4. Program Counter Register (Optional – For Advanced Students)

📘 Definition:

Each thread has its own **Program Counter (PC) Register**, which stores the **address of the currently executing JVM instruction**.

It's mostly used internally by the JVM and isn't something a programmer typically interacts with.

🔷 Summary Table

Memory Area	Stores What?	Access	Thread Scope
Stack	Local variables, method calls, object references	Fast	Thread-local
Heap	All Java objects (allocated via `new`)	Slower	Shared by all
Method Area	Class structure, static variables, constant pool	Medium	Shared by all
PC Register	Address of current JVM instruction (internal)	N/A	Thread-local

---

◆ Real-World Analogy

Imagine a **kitchen** as a program:

- **Stack** = Kitchen counter (small space, fast access, temporary work)
- **Heap** = Fridge or cupboard (stores ingredients — objects)
- **Method Area** = Recipe book (contains instructions for cooking — class info)
- **PC Register** = Bookmark in the recipe book (shows current step)

◆ Diagram: Java Memory Structure (Text Format)

```
+--------------------------+
|      Method Area         |
| - Class info             |
| - Static variables       |
| - Method definitions     |
+--------------------------+
|          Heap            |
| - Objects                |
| - Instance variables     |
+--------------------------+
|          Stack           |
| - Local variables        |
| - Method call frames     |
| - Object references      |
+--------------------------+
|    Program Counter (PC)  |
| - Current instruction    |
+--------------------------+
```

Class Loading & Execution Flow: Static vs Dynamic Class Loading

Introduction to Class Loading

Class loading in Java is the process of loading Java classes into the Java Virtual Machine (JVM) at runtime. Java provides a mechanism to load classes either at compile-time (static loading) or at runtime (dynamic loading).

Class Loading Process in Java

The Java ClassLoader follows three main steps to load a class into memory:

1. **Loading**: The JVM loads the class file (.class) into memory.
2. **Linking**:
   - **Verification**: Ensures the class is correctly formatted and follows Java specifications.
   - **Preparation**: Allocates memory for static variables and assigns default values.
   - **Resolution**: Converts symbolic references into direct references.
3. **Initialization**: Executes static blocks and initializes static variables.

Static Class Loading

What is Static Class Loading?

In static class loading, the class is loaded at compile-time when the program starts executing. It is done using the new keyword or direct class reference.

Example of Static Class Loading:

```
class Student {

    void showDetails() {

        System.out.println("This is an example of Static Class Loading.");

    }


    public static void main(String[] args) {
```

```
        // Static Class Loading: Object is created at compile-time
        Student obj = new Student();
        obj.showDetails();
    }
}
```

Explanation:

1. **Class Student**: This is a simple class with a method showDetails().
2. **Method showDetails()**: Prints a message when called.
3. **Static Class Loading**:
    - new Student(); creates an object during program execution.
    - The method showDetails() is called using this object.

Step-by-Step Execution:

1 **Program Execution Starts:**

- The JVM searches for the `main` method inside the `Student` class.

2 **Class is Loaded:**

- The `Student` class is loaded into memory by the ClassLoader.

3 **Object Creation:**

- The statement `Student obj = new Student();` creates an object of the `Student` class.
- The constructor (if any) gets executed (default constructor is provided by Java if not defined).

4 **Method Call:**

- `obj.showDetails();` is executed.
- The `showDetails()` method prints:

5 **Program Ends:**

- Once all statements inside `main()` are executed, the program terminates.

Advantages of Static Class Loading:

- Faster execution since classes are preloaded.
- Simple and easy to implement.

Disadvantages of Static Class Loading:

- Less flexible; all class dependencies must be known at compile-time.
- Can increase memory usage if unnecessary classes are loaded.

Dynamic Class Loading

What is Dynamic Class Loading?

Dynamic class loading allows the **JVM to load classes at runtime**. This is useful when the class name is **not known at compile-time** and needs to be loaded dynamically.

How to Perform Dynamic Class Loading?

There are **three ways** to load a class dynamically in Java:
1**Using** `Class.forName("ClassName")` – Loads and initializes the class.
2 **Using** `ClassLoader.loadClass("ClassName")` – Loads the class without initialization.
3 **Using Reflection (**`newInstance()`**)** – Creates an object dynamically.

Example of Dynamic Class Loading (Easy Explanation)

```
class Student {
void showDetails() {
System.out.println("This is an example of Dynamic Class Loading.");
```

```
}

public static void main(String[] args) {
try {
// Load class dynamically
Class<?> cls = Class.forName("Student");
```

Breaking It Down:

`Class<?>` → Generic Class Reference

- `Class<?>` is a **generic type reference** to represent any Java class.
- The `<?>` wildcard means it can hold a reference to **any class type**.
- Here, it will store a reference to the `Student` class.

```
// Create instance dynamically
Student obj = (Student) cls.getDeclaredConstructor().newInstance();

// Call method
obj.showDetails();
} catch (Exception e) {
e.printStackTrace();
}
}
}
```

Step-by-Step Execution of Dynamic Class Loading Program

1 The `main` method starts execution.

- JVM starts running the program.

2 Class Loading at Runtime:

- `Class.forName("Student")` loads the class dynamically.
- Unlike static loading, the class is **not loaded at compile-time**.

3 Object Creation at Runtime:

- `newInstance()` creates an object of `Student` class **without using `new` keyword**.

4 Method Execution:

- The `showDetails()` method is called.

5 **Program Ends:**

- Once all statements in `main()` are executed, the program terminates.

Advantages of Dynamic Class Loading:

- More flexible; useful for frameworks and plugin-based applications.
- Reduces memory usage by loading classes only when needed.

Disadvantages of Dynamic Class Loading:

- Slightly slower due to runtime processing.
- May throw ClassNotFoundException if the class is not found.

Comparison Table: Static vs Dynamic Class Loading

Feature	Static Class Loading	Dynamic Class Loading
When Loaded?	At compile-time.	At runtime.
Object Creation	Uses new keyword.	Uses reflection (Class.forName()).
Performance	Faster.	Slightly slower.
Flexibility	Less flexible.	More flexible, allows runtime decision-making.
Use Cases	Standard Java applications.	JDBC, plugins, frameworks, reflection.

Use Cases in Real-World Applications

Static Class Loading:

- General Java applications where class dependencies are known.

- Example: Instantiating an object using new ClassName().

Dynamic Class Loading:

- JDBC drivers (Class.forName("com.mysql.jdbc.Driver")).
- Plugin-based systems (Eclipse plugins, game modding).
- Web frameworks (Spring, Hibernate).

Introduction to Argument Passing in Java

What is Argument Passing?

- When a method is called, values are passed as arguments.
- Java **does not support call by reference**, only **call by value**.
- However, Java can **simulate call by reference** using objects.

Two Types of Arguments in Java

1. **Primitive Data Type Arguments (int, float, char, boolean, etc.)**
   - Passed **by value** (copy of the value is sent).
   - Changes inside the method do not affect the original variable.
2. **Reference Data Type Arguments (Objects, Arrays, etc.)**
   - The reference (memory address) is passed.
   - Changes to the object's properties affect the original object.

```
class Demo {

    public static void main(String[] args) {

        int a = 1000;       // Step 1: Initialize a with 1000

        System.out.println(a);  // Output: 1000
```

```
        int b;
        b = a;            // Step 2: Copy the value of a into b
        System.out.println(b);  // Output: 1000

        b = 2000;         // Step 3: Modify b, but a remains unchanged
        System.out.println(b);  // Output: 2000
        System.out.println(a);  // Output: 1000
    }
}
```

Call by Reference (Object Passing)

Definition

- The reference (address) of an object is passed to the method.
- Changes to the object inside the method affect the original object.

Example: Call by Reference with Arrays

```
class Example {
    void modify(int arr[]) {
        arr[0] = arr[0] + 10;
        System.out.println("Inside method: " + arr[0]);
    }
```

```
    public static void main(String args[]) {
        Example obj = new Example();
        int arr[] = {5};
        System.out.println("Before method call: " + arr[0]);
        obj.modify(arr);
        System.out.println("After method call: " + arr[0]);
    }
}
```

Output

```
Before method call: 5
Inside method: 15
After method call: 15
```

Explanation

- The **reference** of `arr[]` is passed.
- Modifications inside `modify()` affect the original array.

Example: Call by Reference with Objects

```
class Example {
    int value;
    Example(int value) {
        this.value = value;
    }
    void modify(Example obj) {
```

```
        obj.value += 10;
        System.out.println("Inside method: " + obj.value);
    }
    public static void main(String args[]) {
        Example obj = new Example(5);
        System.out.println("Before method call: " + obj.value);
        obj.modify(obj);
        System.out.println("After method call: " + obj.value);
    }
}
```

Output

```
Before method call: 5
Inside method: 15
After method call: 15
```

Explanation

- The method receives a reference to `obj`.
- Modifications inside `modify()` persist outside.

Call by Value vs Call by Reference

Feature	Call by Value (Primitives)	Call by Reference (Objects/Arrays)
What is passed?	Copy of the value	Reference to the original object
Effect on original variable?	No change	Changes persist
Used for	int, float, char, boolean	Objects, Arrays, Custom Classes
Example	`modify(int x)`	`modify(Example obj)`

Feature	Call by Value (Primitives)	Call by Reference (Objects/Arrays)
Security	More secure	Risk of unintentional modification

Real-Life Analogy

- **Call by Value**: Giving someone a photocopy of a document (original remains unchanged).
- **Call by Reference**: Giving someone the original document (they can modify it).

Wrapper Class in Java

In Java, **wrapper classes** are used to convert primitive data types into objects. Java provides wrapper classes for each of its eight primitive data types. These classes are part of the **java.lang** package.

List of Wrapper Classes in Java

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Why Use Wrapper Classes?

1. **Collections Framework Compatibility** – Java collections like `ArrayList` work with objects, not primitives.
2. **Autoboxing and Unboxing** – Automatic conversion between primitives and objects.
3. **Utility Methods** – Wrapper classes provide useful methods, e.g., `Integer.parseInt()`, `Double.toString()`.
4. **Handling Null Values** – Primitive types cannot store `null`, but wrapper classes can.
5. **Type Safety in Generics** – Java Generics work with objects, not primitive data types.

Autoboxing and Unboxing

Autoboxing (Primitive → Wrapper Object)

Java automatically converts a primitive type into its corresponding wrapper class.

```
public class AutoboxingExample {
    public static void main(String[] args) {
        int num = 10;  // Primitive type
        Integer obj = num;  // Autoboxing (int → Integer)
        System.out.println(obj);
    }
}
```

Unboxing (Wrapper Object → Primitive)

Java automatically converts an object of a wrapper class to its corresponding primitive type.

```
public class UnboxingExample {
    public static void main(String[] args) {
        Integer obj = 20;  // Wrapper object
        int num = obj;  // Unboxing (Integer → int)
        System.out.println(num);
    }
}
```

Wrapper Class Methods

Wrapper classes provide useful methods:

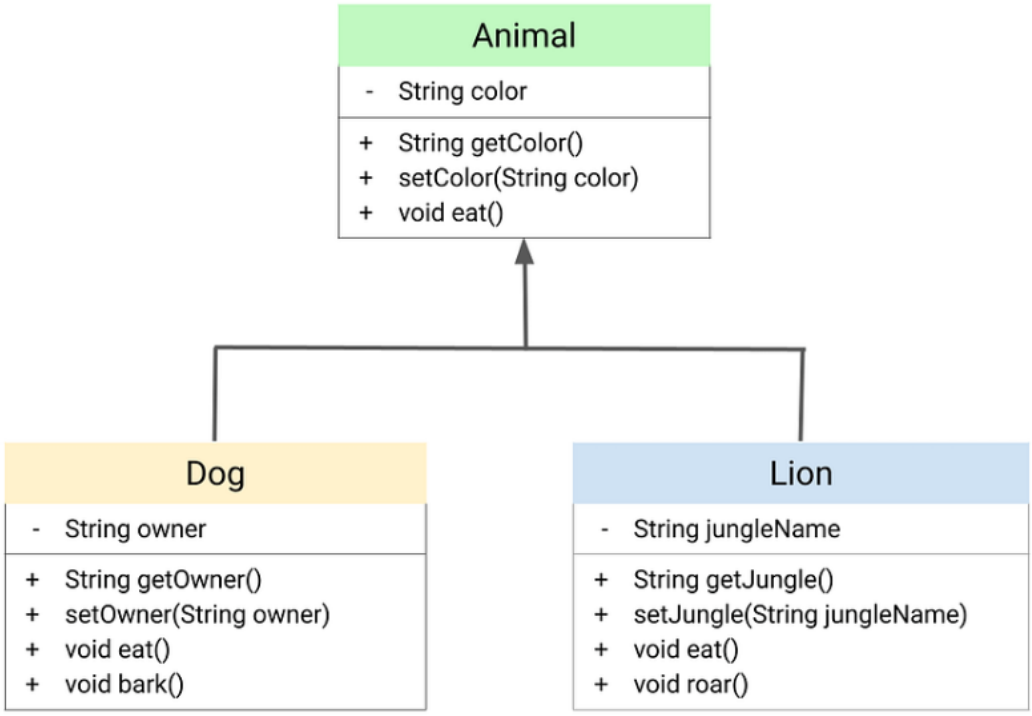
```
public class WrapperMethods {
    public static void main(String[] args) {
        // Converting String to Integer
        int num = Integer.parseInt("100");
        System.out.println(num); // Output: 100
```

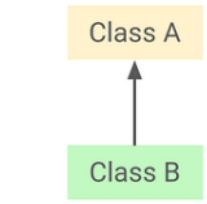
```
        // Getting String representation of a number
        String str = Integer.toString(50);
        System.out.println(str); // Output: "50"

        // Checking minimum and maximum values
        System.out.println("Min Int: " + Integer.MIN_VALUE);
        System.out.println("Max Int: " + Integer.MAX_VALUE);
    }
}
```

Date: 24/02/2025

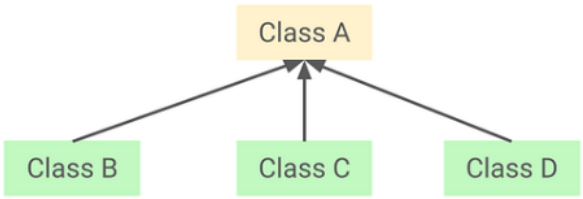
Subject JAVA

Topic: Inheritance and Code Reusability in Java and Super Keyword

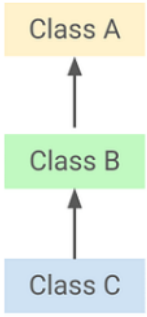

Animal
- String color
+ String getColor()
+ setColor(String color)
+ void eat()
Dog
- String owner
+ String getOwner()
+ setOwner(String owner)
+ void eat()
+ void bark()
Lion
- String jungleName
+ String getJungle()
+ setJungle(String jungleName)
+ void eat()
+ void roar()



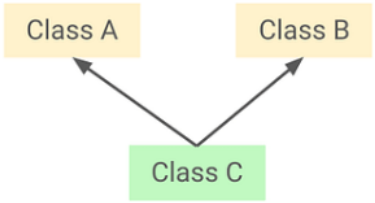
Single Inheritance



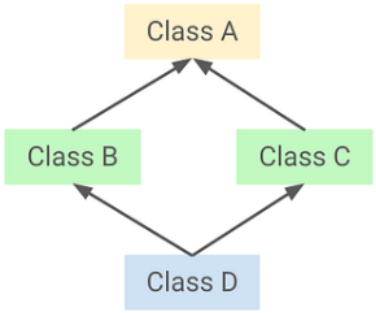
Hierarchical inheritance



Multilevel Inheritance



Multiple Inheritance



Hybrid Inheritance

```
public class Vehicle {
    ..............
    ..............
 }
                IS-A Relationship

public class Car extends Vehicle {
    ..............
    ..............
}               IS-A Relationship

public class Alto extends Car {
    ..............
    ..............
 }
```

Fig a: Is-A relationship between classes

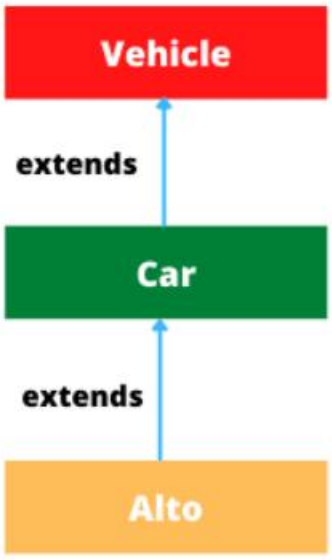


Fig b: Inheritance hierarchy for Vehicle, Car, and Alto

Inheritance and Code Reusability in Java

Inheritance is a fundamental concept in **Object-Oriented Programming (OOP)** that allows one class (**child/subclass**) to inherit the properties and behavior (fields and methods) of another class (**parent/superclass**). This promotes **code reusability** and improves maintainability.

Key Features of Inheritance:

1. **Code Reusability:** Common functionality is defined in the parent class, reducing code duplication.
2. **Extensibility:** New functionalities can be added to a subclass without modifying the existing parent class.
3. **Polymorphism:** Allows method overriding, enabling dynamic method dispatch.

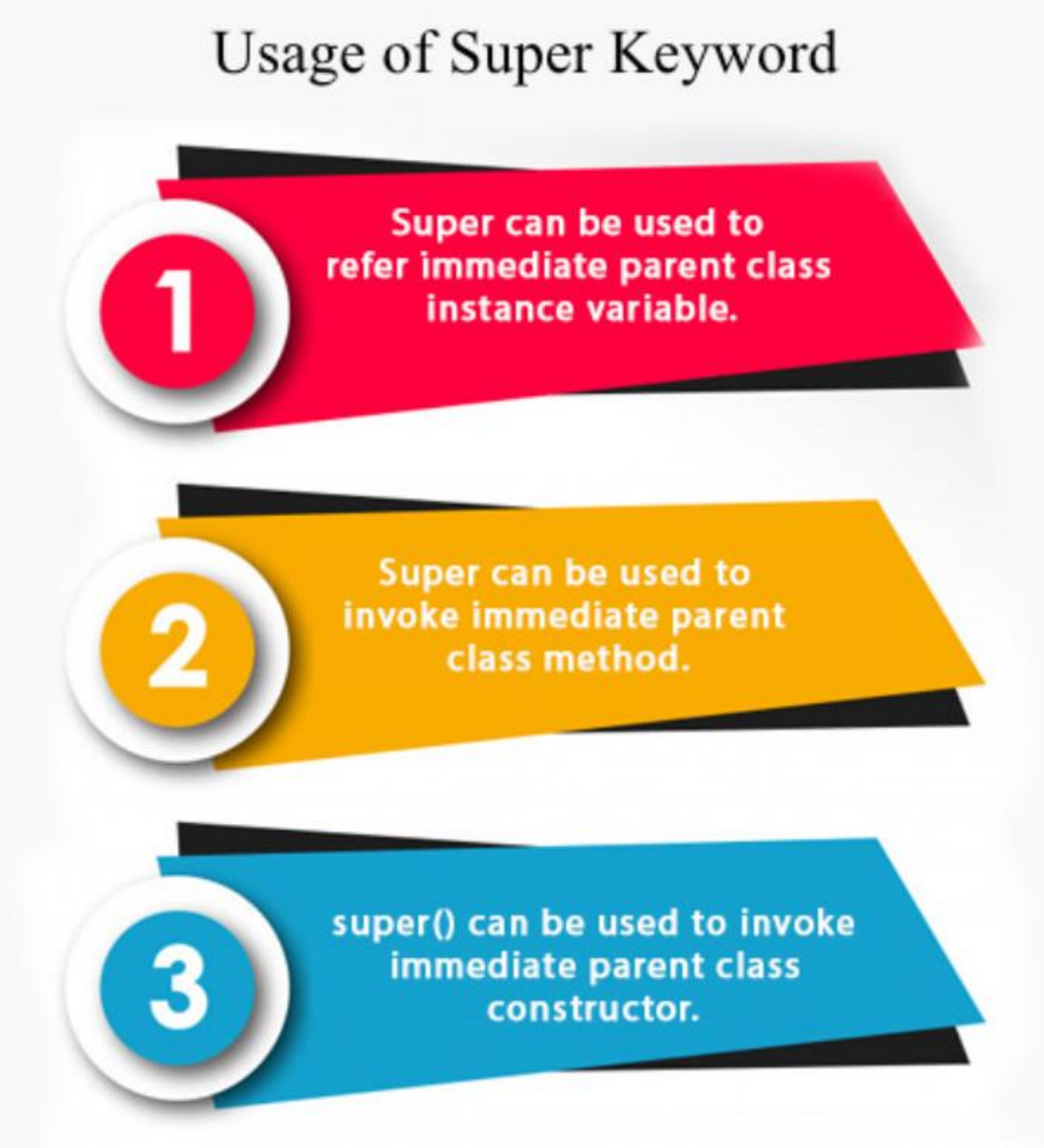
Syntax of Inheritance:

```
class Parent {
    void display() {
        System.out.println("This is the parent class method.");
    }
}

class Child extends Parent {
    void show() {
        System.out.println("This is the child class method.");
    }
}

public class InheritanceExample {
    public static void main(String args[]) {
        Child obj = new Child();
        obj.display();  // Calling the parent class method
        obj.show();     // Calling the child class method
    }
}
```

Output:

```
This is the parent class method.
This is the child class method.
```

Usage of `super` Keyword



The `super` keyword in Java is used to refer to the **parent class**. It has three primary uses:

1. **Accessing Parent Class Methods**
   The `super` keyword can be used to call the **superclass's method** when the subclass has overridden it.

```
class Parent {
    void show() {
        System.out.println("This is the parent class method.");
    }
}

class Child extends Parent {
    void show() {
        super.show();  // Calls the parent class method
        System.out.println("This is the child class method.");
    }
}

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

Output:

```
This is the parent class method.
This is the child class method.
```

2. **Accessing Parent Class Variables**
   If the child class has a variable with the same name as the parent class, `super` helps access the parent's variable.

```
class Parent {
    int num = 100;
}

class Child extends Parent {
    int num = 200;

    void show() {
        System.out.println("Child class variable: " + num);
        System.out.println("Parent class variable: " + super.num);
    }
}

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

```
}
```

Output:

```
Child class variable: 200
Parent class variable: 100
```

3. **Calling Parent Class Constructor**
   `super()` is used to call the **parent class constructor** from the child class constructor.

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

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

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

Output:

```
Parent class constructor
Child class constructor
```

Conclusion

- **Inheritance** helps in reducing redundancy and promotes **code reusability**.
- The `super` keyword is useful for accessing **parent class methods, variables, and constructors** in a subclass.
- **Method overriding** in combination with `super` ensures the correct method execution from both parent and child classes.