

Class (Blueprint):

A class is like a blueprint or template.

It defines how the object should look and behave — it contains variables (properties) and methods (functions).

But it doesn't occupy memory until you create an object.

👉 Think of a class like a house plan — it shows how the house will look but is not the house itself.

```
class Car {
    String color;
    void drive() {
        System.out.println("Car is driving");
    }
}
```

Object (Instance):

An object is a real-world instance created from the class.

It occupies memory and can use the methods/variables of the class.

👉 Think of an object like an actual house built from the blueprint.

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(); // Object
        myCar.color = "Red";
        myCar.drive();          // Calls method
    }
}
```

Output:

Car is driving

Difference between Class and Object

Feature	Class	Object
Meaning	Blueprint or template	Real instance created from the class
Definition	Defined using <code>class</code> keyword	Created using <code>new</code> keyword
Memory	Does not occupy memory by itself	Occupies memory when created
Usage	Defines properties and behavior	Accesses properties and behavior of the class
Example	<code>class Car {}</code>	<code>Car myCar = new Car();</code>
Type	Logical entity	Physical (real) entity
Lifespan	Exists throughout program execution	Exists as long as reference is alive

Creating Objects

Objects are created using the new keyword:

```
ClassName obj = new ClassName();
```

Why this is enough for now?

99% of beginner/intermediate Java programs use only this.

Other ways like `clone()`, reflection, deserialization — you can learn later.

Constructor

A constructor is a special method used to **initialize an object**.

Rules:

1. Constructor name must be same as class name.
2. Constructor should not have a return type.
3. Constructors **cannot be abstract, static, final, or synchronized**.

Example:

```
class Person {  
    public Person() {  
        System.out.println("Constructor called");  
    }  
}
```

Not a constructor:

```
public class Person {  
    public void Person() {  
        System.out.println("This is not a constructor");  
    }  
}
```

```
    }  
}
```

Why certain keywords are invalid with constructors?

- **final:** Constructors can't be inherited or overridden.
 - **static:** Constructors are tied to objects, not classes.
 - **abstract:** Constructors must be complete to run.
 - **synchronized:** Object isn't available until constructor completes.
-

Types of Constructors

Java supports 2 main types:

1. Default Constructor (No-Arg Constructor)

Constructor without parameters.

```
class Bike1 {  
    public Bike1() {  
        System.out.println("Bike is created");  
    }  
  
    public static void main(String[] args) {  
        Bike1 b = new Bike1();  
    }  
}
```

If no constructor is written, Java compiler adds a default one automatically.

```
class Bike2 {} // Compiler turns it into: public Bike2() {}
```

Default constructor assigns default values: 0, null, false, etc.

```
class Student3 {
    int id;
    String name;

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

    public static void main(String[] args) {
        Student3 s1 = new Student3();
        Student3 s2 = new Student3();
        s1.display();
        s2.display();
    }
}
// Output: 0 null
//          0 null
```

2. Parameterized Constructor

Constructor with parameters to assign specific values.

```
class Student4 {
    int id;
    String name;

    Student4(int i, String n) {
        id = i;
        name = n;
    }

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

    public static void main(String[] args) {
        Student4 s1 = new Student4(111, "Karan");
        Student4 s2 = new Student4(222, "Aryan");
    }
}
```

```
        s1.display();
        s2.display();
    }
}
```

Constructor Overloading

Multiple constructors in one class with different parameters.

```
class Student5 {
    int id;
    String name;
    int age;

    Student5(int i, String n) {
        id = i;
        name = n;
    }

    Student5(int i, String n, int a) {
        id = i;
        name = n;
        age = a;
    }

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

    public static void main(String[] args) {
        Student5 s1 = new Student5(111, "Karan");
        Student5 s2 = new Student5(222, "Aryan", 25);
        s1.display();
        s2.display();
    }
}

// Output:
```

```
// 111 Karan 0
// 222 Aryan 25
```

Constructor Chaining

Calling one constructor from another.

1. Using this() (same class):

```
class ConstructorChaining {
    ConstructorChaining() {
        this(5);
        System.out.println("Default constructor");
    }

    ConstructorChaining(int i) {
        this("constructor");
        System.out.println("This is int type constructor");
    }

    ConstructorChaining(String s) {
        System.out.println("This is string constructor");
    }

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

Output:

```
This is string constructor
This is int type constructor
Default constructor
```

2. Using super() (parent class):

```
class Demo {
```

```

    Demo() {
        System.out.println("This is a default constructor of
Demo class");
    }
}

public class ConstructorChaining extends Demo {
    ConstructorChaining() {
        this(5);
        System.out.println("This is default constructor");
    }

    ConstructorChaining(int i) {
        this("Keerthi");
        System.out.println("This is int type constructor");
    }

    ConstructorChaining(String s) {
        super();
        System.out.println("String constructor");
    }

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

```

Output:

```

This is a default constructor of Demo class
String constructor
This is int type constructor
This is default constructor

```


Copying Values Without Constructor

```
class Student7 {
    int id;
    String name;

    Student7(int i, String n) {
        id = i;
        name = n;
    }

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

    public static void main(String[] args) {
        Student7 s1 = new Student7(111, "Karan");
        Student7 s2 = new Student7(222, "Aryan");

        s2.id = s1.id;
        s2.name = s1.name;

        s1.display();
        s2.display();
    }
}
```

Copy Constructor (Manually Defined)

```
class Student6 {
    int id;
    String name;

    Student6(int i, String n) {
        id = i;
        name = n;
    }
}
```

```

Student6(Student6 s) {
    id = s.id;
    name = s.name;
}



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

public static void main(String[] args) {
    Student6 s1 = new Student6(111, "Karan");
    Student6 s2 = new Student6(s1);
    s1.display();
    s2.display();
}
}

```

Constructor vs Method in Java

Feature	Constructor	Method
Purpose	Initializes an object	Defines the behavior of an object
Name	Same as class name	Any valid identifier
Return Type	No return type (not even void)	Must have a return type (void, int, String, etc.)
Called When	Automatically when an object is created	Manually by calling through object
Default Provided by Compiler	Yes, if no constructor is written	No
Overloading	Yes (Constructor Overloading)	Yes (Method Overloading)

Inheritance	Not inherited by subclass	Can be inherited and overridden
Can be abstract/static/final?	 Not allowed	 Allowed
Example	Student s = new Student();	s.display();

What is static in Java?

- The static keyword is used for **class-level members**.
- Static members **belong to the class**, not to individual objects.
- They are **shared among all objects** of the class.

Static Variables

- Declared using the static keyword.
- **Only one copy** exists for all objects.
- Useful for values common to all objects (like college name, company name).

Example:

```
class Student {
    int id;
    String name;
    static String college = "ABC College"; // static variable
```

```
Student(int i, String n) {  
    id = i;  
    name = n;  
}  
  
void display() {  
    System.out.println(id + " " + name + " " + college);  
}  
  
public static void main(String[] args) {  
    Student s1 = new Student(101, "Keerthi");  
    Student s2 = new Student(102, "gopi");  
  
    s1.display();  
    s2.display();  
}  
}
```

Output:

```
101 Keerthi ABC College  
102 gopi ABC College
```

Static Method – (Class method)

Definition:

A method with static keyword. It **belongs to the class**, not to any object.

Key Points:

- You can call it **without creating object**.
- It can access **only static variables** directly.

Example:

```
class Hello {  
    static int x = 10;  
  
    static void show() {  
        System.out.println("Static method called");  
        System.out.println("x = " + x); // OK: x is static  
    }  
  
    public static void main(String[] args) {  
        Hello.show(); // no object required  
    }  
}
```

Output:

```
Static method called  
x = 10
```

You don't need to write `Hello h = new Hello();`
You can call static method using class name.

Static Block – (Auto runs before main)**Definition:**

Static block is a set of statements in `{}` with `static` keyword.

It runs **only once** when class is loaded (before `main()` starts).

Use:

- To **initialize static variables**.

- To print or run setup code **once** at the start.

Example:

```
class Demo {  
    static int a;  
  
    static {  
        a = 100;  
        System.out.println("Static block executed");  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Main method");  
        System.out.println("a = " + a);  
    }  
}
```

Output:

Static block executed

Main method

a = 100

Static block runs **before main()**

Used for **initial setup** (like DB connections, constants etc.)
