Array

🔶 What is an Array in Java?

An **array** is like a container that holds **multiple values of the same type**.

Imagine a row of boxes where each box can store one value. All boxes (elements) are of the **same data type** like int, float, or String.

🔷 Why Use Arrays?

- To store multiple values in one variable instead of creating many individual variables.
- Easy to manage collections of data using loops.

🔷 Types of Arrays

✅ 1. One-Dimensional Array

- Like a list of items (e.g., marks of 5 students).

🔶 How to Declare and Initialize:

java

CopyEdit

```
int[] marks = new int[5];  // creates an array to store 5 integers

marks[0] = 90;

marks[1] = 85;

// and so on...
```

🔶 Shortcut Initialization:

java

CopyEdit

```
int[] marks = {90, 85, 75, 88, 92};

System.out.println(marks[0]); // prints 90
```

🔶 **Looping Through Array:**

java

CopyEdit

```
for (int i = 0; i < marks.length; i++) {
   System.out.println(marks[i]);
}
```

✅ **2. Multi-Dimensional Array**

- Like a table or grid (e.g., a matrix or a chessboard).

🔶 **Example: 2D Array**

java

CopyEdit

```
int[][] matrix = new int[2][3];  // 2 rows, 3 columns
matrix[0][0] = 1;
matrix[0][1] = 2;
matrix[1][0] = 3;
```

🔶 **Initialization with Values:**

java

CopyEdit

```
int[][] matrix = {
   {1, 2, 3},
   {4, 5, 6}
};
System.out.println(matrix[1][2]); // prints 6
```

🔶 **Looping Through 2D Array:**

java

CopyEdit

```
for (int i = 0; i < matrix.length; i++) {
   for (int j = 0; j < matrix[i].length; j++) {
      System.out.print(matrix[i][j] + " ");
   }
   System.out.println();
}
```

🔷 Alternative Declarations

All of the following are valid:

```
java
CopyEdit
int[] a;      // recommended
int a[];      // also valid
String[] names;
```

🔷 Key Points to Remember

- Arrays are **fixed in size** – you can't change the length after creation.
- Indexing starts from **0**.
- If you try to access an index beyond the limit, Java will throw **ArrayIndexOutOfBoundsException**.

✅ Example Program: Sum of Array Elements

```
java
CopyEdit
public class ArraySum {
   public static void main(String[] args) {
      int[] numbers = {10, 20, 30, 40};
      int sum = 0;
```

```
        for (int i = 0; i < numbers.length; i++) {
            sum += numbers[i];
        }

        System.out.println("Sum = " + sum);
    }
}
```

String

🔶 What is a String in Java?

In Java, a **String** is a **sequence of characters** (like letters, digits, or symbols). For example: "Hello", "Java123" are strings.

- Strings are **objects**, not just text.
- Java provides the **String class** to work with strings.
- Strings are **immutable** — once created, their values can't be changed.

🔷 How to Create Strings

✅ 1. Using Double Quotes (Most common way):

java

CopyEdit

String name = "Java";

✅ 2. Using the new Keyword:

java

CopyEdit

String name = new String("Java");

🔷 Common String Methods

Method	Description	Example
length()	Returns number of characters	"Java".length() → 4
charAt(index)	Returns character at given index	"Java".charAt(1) → 'a'
toUpperCase()	Converts to uppercase	"java".toUpperCase() → "JAVA"

Method	Description	Example
toLowerCase()	Converts to lowercase	"JAVA".toLowerCase() → "java"
equals(str)	Checks if two strings are equal	"Java".equals(\"Java\") → true
equalsIgnoreCase()	Compares ignoring case	"java".equalsIgnoreCase(\"JAVA\") → true
substring(start, end)	Extracts part of string	"Hello".substring(1, 4) → "ell"
indexOf(char)	Returns first index of a character	"hello".indexOf('l') → 2
trim()	Removes leading and trailing spaces	" Java ".trim() → "Java"
replace(a, b)	Replaces characters	"Java".replace('a', 'o') → "Jovo"

🔷 Example Program Using Strings

java

CopyEdit

```
public class StringExample {
    public static void main(String[] args) {
        String str = " Java Programming ";

        System.out.println("Original: " + str);
        System.out.println("Length: " + str.length());
        System.out.println("Trimmed: " + str.trim());
        System.out.println("Uppercase: " + str.toUpperCase());
        System.out.println("Substring: " + str.substring(1, 5));
        System.out.println("Character at index 2: " + str.charAt(2));
    }
```

}

🔷 String Immutability (Why it matters)

java

CopyEdit

```
String s = "Java";
s.concat(" Language");  // This does not change 's'
System.out.println(s);  // Output: Java
```

To modify a string, you must assign the result to a new variable:

java

CopyEdit

```
s = s.concat(" Language");
System.out.println(s);  // Output: Java Language
```

🔷 Comparing Strings

✅ **Correct Way:**

java

CopyEdit

```
String a = "Java";
String b = "Java";
System.out.println(a.equals(b)); // true
```

⚠️ **Don't use == for string comparison (it checks memory, not content).**

🔷 String Concatenation

✅ **Using + operator:**

java

CopyEdit

String firstName = "John";

String lastName = "Doe";

String fullName = firstName + " " + lastName;

✅ **Using concat() method:**

java

CopyEdit

String fullName = firstName.concat(" ").concat(lastName);

🔷 **StringBuilder & StringBuffer (Mutable Strings)**

If you need a **changeable string**, use these:

java

CopyEdit

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

System.out.println(sb); // Output: Hello World

🧠 **Summary:**

- **String = text in double quotes**
- Strings are **objects** and **immutable**
- Use equals() to compare, not ==
- Many useful methods to manipulate and check strings
- Use StringBuilder for changeable strings

✅ **Does a Method Actually Change the Object or Return Something in Java?**

🔷 **The answer is: It depends on the method and the type of object.**

🔶 **Two Cases:**

✅ **1. If the object is mutable (can be changed),**

Then the method can **change the actual object**.

Example: StringBuilder (mutable)

java

CopyEdit

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World"); // modifies the original object

System.out.println(sb); // Output: Hello World

- append() modifies the **original object** (sb) directly.
- No need to assign it to a new variable.

❌ **2. If the object is immutable (can't be changed),**

Then the method **returns a new object**, and the original stays the same.

Example: String (immutable)

java

CopyEdit

String s = "Hello";

s.toUpperCase(); // returns a new string, but does not change 's'

System.out.println(s); // Output: Hello (not changed!)

✅ To update:

java

CopyEdit

s = s.toUpperCase();

System.out.println(s); // Output: HELLO

🔶 **Important Concept: Java is Pass-by-Value**

Even for objects, Java passes a **copy of the reference**. So:

- If the method **modifies internal data**, the original object **is changed**.

- If the method **reassigns the reference**, the original object **is NOT changed**.

🔷 **Example: Object Modification vs Reassignment**

java

CopyEdit

```
class Car {
   String color;
}

void paint(Car c) {
   c.color = "Red";  // changes original object
}

void replace(Car c) {
   c = new Car();    // changes only the local copy of reference
   c.color = "Blue";
}

Car myCar = new Car();
paint(myCar);      // myCar.color becomes "Red"
replace(myCar);    // myCar is still the same object with color "Red"
```

🧠 **Summary:**

Method Affects Object?	**Depends on...**
✅ Changes object	If object is **mutable**
❌ Returns new object	If object is **immutable**
❌ Changing parameter	If reference is **reassigned**

🔶 What are Mutable and Immutable Objects?

Term	Meaning
Mutable	Can be changed after it's created
Immutable	Cannot be changed after it's created

🔷 Immutable Objects in Java

✅ Examples:

- String
- Wrapper classes (Integer, Double, Boolean, etc.)
- LocalDate, LocalTime, etc.

✅ Key Features of Immutable Objects:

- Their internal state (data) **cannot be changed** after creation.
- Any method that seems to modify the object actually returns a **new object**.

🔶 Example: String is immutable

java

CopyEdit

```
String s = "Java";
s.concat(" Programming"); // Doesn't change 's'
System.out.println(s);    // Output: Java (not changed!)

s = s.concat(" Programming"); // Now the new value is stored
System.out.println(s);       // Output: Java Programming
```

🔶 Why are immutable objects useful?

- They are **safe to use in multi-threading**.
- They help avoid **unexpected changes** in data.
- They make debugging easier.

🔷 Mutable Objects in Java

✅ Examples:

- StringBuilder, StringBuffer
- Arrays
- Custom classes (if fields can be changed)

✅ Key Features of Mutable Objects:

- You **can change the content** without creating a new object.

🔶 Example: StringBuilder is mutable

java

CopyEdit

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");  // Modifies the original object
System.out.println(sb); // Output: Hello World
```

🔷 Custom Mutable and Immutable Classes

✅ Mutable Class Example:

java

CopyEdit

```
class Person {
   String name;
   void setName(String newName) {
      name = newName;
   }
}
```

✅ Immutable Class Example:

java

CopyEdit

```
final class Person {
  private final String name;

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

  public String getName() {
    return name;
  }
}
```

- final class prevents subclassing.
- private final ensures field can't change.
- No setter method to modify the value.

🔶 **How to Make a Class Immutable:**

1. Make the class final.
2. Make all fields private final.
3. Don’t provide any setter methods.
4. Initialize all fields in the constructor.
5. If fields are objects, return copies (not references).

🧠 **Summary Table**

Feature	Mutable Object	Immutable Object
Can change data	✅ Yes	❌ No
Returns same object	✅ Yes	❌ No (returns new)
Thread-safe	❌ Not always	✅ Yes

Feature	Mutable Object	Immutable Object
Examples	StringBuilder, Array	String, Integer

Class and Object

✅ 1. Class Fundamentals

🔷 What is a Class?

- A **class** is like a **blueprint** or **template**.
- It defines what **data (variables)** and **behaviors (methods)** an object can have.

🔶 Syntax:

java

CopyEdit

```
class ClassName {
   // data (fields/variables)
   // behavior (methods)
}
```

🔶 Example:

java

CopyEdit

```
class Car {
   // data (fields)
   String color;
   int speed;

   // behavior (methods)
   void start() {
      System.out.println("Car started");
   }
}
```

✅ 2. Declaring Objects

🔷 What is an Object?

- An **object** is a **real-world entity** created from a class.
- It holds its own **data** and can **use methods** defined in the class.

🔶 Syntax:

java

CopyEdit

```
ClassName objName = new ClassName();
```

🔶 Example:

java

CopyEdit

```
Car myCar = new Car();  // 'myCar' is an object of class Car
```

✅ 3. Assigning Object Reference Variables

🔷 Object references are like pointers to the object in memory.

🔶 Example:

java

CopyEdit

```
Car car1 = new Car();    // car1 points to a Car object
Car car2 = car1;         // car2 now points to the same object as car1


car2.color = "Red";
System.out.println(car1.color);  // Output: Red (because both refer to the same object)
```

If you change the object using one reference, the change is reflected through the other.

✅ 4. Adding Methods to a Class

🔷 Methods are functions inside a class — they define the behavior of an object.

🔶 **Example:**

java

CopyEdit

```
class Calculator {
   int add(int a, int b) {
      return a + b;
   }

   void printMessage() {
      System.out.println("Welcome to Calculator!");
   }
}
```

🔷 **Using Methods:**

java

CopyEdit

```
Calculator calc = new Calculator();
int sum = calc.add(10, 20);
System.out.println("Sum = " + sum);  // Output: Sum = 30
calc.printMessage();  // Output: Welcome to Calculator!
```

🧠 **Summary**

Concept	**Description**
Class	A blueprint for objects
Object	An instance of a class
Declaring an object	ClassName obj = new ClassName();
Object reference variable	Points to the object in memory

Concept	Description
Methods in a class	Define behaviors that objects can perform

✅ 5. Returning a Value from a Method

🔷 What does it mean?

A method can **compute something** and give the result back using the return keyword.

🔶 Syntax:

java

CopyEdit

returnType methodName(parameters) {

  // logic

  return value;

}

🔶 Example:

java

CopyEdit

class Calculator {

  int add(int a, int b) {

    return a + b; // returns the sum

  }

}

🔷 How to use it:

java

CopyEdit

Calculator calc = new Calculator();

```
int result = calc.add(5, 3);
System.out.println("Sum = " + result);  // Output: Sum = 8
```

🔷 **Important Notes:**

- The **return type** must match the **type of value returned**.
- You can return any type: int, double, String, even objects.

✅ **6. Constructors in Java**

🔷 **What is a Constructor?**

- A **constructor** is a special method that runs **when an object is created**.
- It is used to **initialize** object data.
- It has the **same name as the class** and **no return type**.

🔶 **Default Constructor:**

If you don't create any constructor, Java provides a default one.

java

CopyEdit

```
class Student {
   Student() {
      System.out.println("Student object created");
   }
}
```

Usage:

java

CopyEdit

```
Student s = new Student();  // Output: Student object created
```

🔶 **Parameterized Constructor:**

You can define your own constructor that takes arguments.

java

CopyEdit

```
class Student {
   String name;

   Student(String studentName) {
      name = studentName;  // initializing name
   }

   void display() {
      System.out.println("Name: " + name);
   }
}
```

Usage:

java

CopyEdit

```
Student s = new Student("Alice");
s.display();  // Output: Name: Alice
```

🔶 **Constructor Overloading:**

You can have **multiple constructors** with different parameter lists.

java

CopyEdit

```
class Box {
   int length, width;
```

```
  Box() {
    length = 0;
    width = 0;
  }

  Box(int l, int w) {
    length = l;
    width = w;
  }
}
```

🧠 Summary

Concept	Description
return keyword	Sends a value back from a method
Constructor	Special method to initialize an object
Default constructor	No-argument constructor provided by Java
Parameterized constructor	Allows setting values during object creation
Constructor overloading	Multiple constructors with different parameters

This

✅ What is this in Java?

- this is a **special keyword** in Java.
- It is a **reference to the current object** — the object on which the method or constructor is called.

🔶 Why and Where We Use this?

Use Case	Meaning
To refer to **current object's variables**	When local and instance variables have the same name
To call **current object's methods**	Inside the class
To call **another constructor** in the same class	Constructor chaining
To **pass current object** as argument	Useful in callbacks or chaining
Used **in inheritance** to distinguish superclass and subclass	

🔷 1. this to Refer Current Object's Variable

java

CopyEdit

```
class Student {
  String name;

  Student(String name) {
    this.name = name;  // this.name = instance variable, name = parameter
  }
```

```
    void display() {
        System.out.println("Name: " + this.name);
    }
}
```

Without this, Java would get confused between local and instance variable.

🔷 2. this to Call Current Class Method

java

CopyEdit

```
class Demo {
    void show() {
        System.out.println("Hello");
    }

    void call() {
        this.show(); // same as just calling show()
    }
}
```

🔷 3. this() to Call Another Constructor (Constructor Chaining)

java

CopyEdit

```
class Box {
    int length, width;

    Box() {
        this(10, 5);  // calls the constructor below
```

```
    }

    Box(int l, int w) {
        length = l;
        width = w;
    }

    void display() {
        System.out.println("Length: " + length + ", Width: " + width);
    }
}
```

🔷 4. this as Method Argument

java

CopyEdit

```
class A {
    void display(A obj) {
        System.out.println("Method called with object: " + obj);
    }

    void call() {
        display(this); // passing current object
    }
}
```

✅ this in Inheritance

In inheritance, this always refers to the **current class object**, even if it's a subclass.

java

CopyEdit

```
class Animal {
   void showType() {
      System.out.println("This is an animal");
   }
}

class Dog extends Animal {
   void display() {
      this.showType(); // calls showType() from Animal
   }
}
```

- this.showType() works even if showType() is in the **parent class**.
- It can also call **overridden methods** if they exist in the subclass.

🔶 **Overriding + this**

java

CopyEdit

```
class Animal {
   void sound() {
      System.out.println("Animal sound");
   }
}

class Cat extends Animal {
   void sound() {
      System.out.println("Meow");
   }
```

```
    void callSound() {
        this.sound();  // calls Cat's version
        super.sound(); // calls Animal's version
    }
}
```

Output:

```
nginx
CopyEdit
Meow
Animal sound
```

🧠 Summary

Use of this	Purpose
this.variable	Access current object’s field
this.method()	Call current object’s method
this()	Call another constructor in the same class
this	Pass current object as parameter
With inheritance	Refers to current subclass object

Garbage Collection and Finalize()

✅ Garbage Collection (GC) in Java

What is Garbage Collection?

- In Java, **objects are created in memory (heap)** when you use new.
- When objects are **no longer needed** or **no references point to them**, they become **garbage**.
- **Garbage Collection** is the process where Java **automatically frees memory** by deleting these unused objects.
- You **don't need to manually free memory** (like in C/C++).

How Garbage Collection Works?

- The **Java Garbage Collector** runs in the background.
- It **identifies objects that are unreachable** (no active reference).
- It **removes those objects from memory** to free space.
- The exact time when GC runs is **not predictable** and controlled by the JVM.

Example of Garbage Collection

java

CopyEdit

```
class Demo {
  public static void main(String[] args) {
    Demo obj1 = new Demo();  // obj1 points to a new object
    obj1 = null;          // object is now eligible for GC
    System.gc();          // Request JVM to run garbage collector (not guaranteed)
    System.out.println("End of main");
  }
```

}

- When obj1 is set to null, the object it was pointing to has **no references**.
- JVM can garbage collect that object to free memory.

✅ **finalize() Method in Java**

What is finalize()?

- finalize() is a method called **by the Garbage Collector** before an object is removed from memory.
- It gives the object a chance to **clean up resources** like closing files or releasing connections.

How to use finalize()?

- You **override** the finalize() method in your class.
- JVM calls it **once before the object is destroyed** by GC.

Example:

java

CopyEdit

```
class Demo {
  protected void finalize() {
    System.out.println("finalize method called");
  }

  public static void main(String[] args) {
    Demo obj = new Demo();
    obj = null;
    System.gc();  // Request GC to run
    System.out.println("End of main");
```

```
    }
}
```

Important Notes on finalize():

- It is **not guaranteed** when or even if finalize() will be called.
- It is **deprecated** since Java 9 because it is unreliable and can cause performance issues.
- Instead of finalize(), use **try-with-resources** or explicit resource management.

🧠 Summary

Concept	Description
Garbage Collection	Automatically deletes objects that are no longer referenced to free memory
finalize() method	A method called before GC deletes an object; used for cleanup but **not reliable or recommended**

Method Overloading

✅ 1. Method Overloading

What is Method Overloading?

- **Method Overloading** means having **multiple methods in the same class** with the **same name** but **different parameters** (different type, number, or order).
- It lets you perform similar actions in different ways using the same method name.

Why use method overloading?

- Makes code easier to read.
- Allows the same method to handle different types of inputs.

Rules for Method Overloading:

Parameter Difference	Allowed?
Different number of parameters	Yes
Different types of parameters	Yes
Different order of parameters	Yes
Different return types only	No (not enough to overload)

Example of Method Overloading:

java

CopyEdit

```
class Calculator {
   // add two integers
   int add(int a, int b) {
      return a + b;
```

```
    }

    // add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // add two double values
    double add(double a, double b) {
        return a + b;
    }
}

public class Test {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(10, 20));      // Output: 30
        System.out.println(calc.add(10, 20, 30));  // Output: 60
        System.out.println(calc.add(5.5, 3.3));    // Output: 8.8
    }
}
```

✅ 2. Argument Passing in Java

How are arguments passed to methods?

- Java uses **pass-by-value** for all method arguments.
- **What does this mean?**

Explanation:

- When you pass a **primitive type** (like int, float), a **copy of the value** is passed.
- When you pass an **object**, a **copy of the reference** to the object is passed.

Pass-by-value for Primitives

java

CopyEdit

```
void change(int x) {
   x = 100;   // changes local copy only
}

int a = 10;
change(a);
System.out.println(a);  // Output: 10 (original value unchanged)
```

Pass-by-value for Objects

java

CopyEdit

```
class Box {
   int size;
}

void changeSize(Box b) {
   b.size = 50;  // changes the object's size because reference points to the same object
}

Box box = new Box();
box.size = 10;
changeSize(box);
```

System.out.println(box.size); // Output: 50

- The reference to box is copied, but **both references point to the same object**.
- Changing the object via the reference affects the original object.

What about reassigning the reference?

java

CopyEdit

void changeBox(Box b) {

  b = new Box(); // b now points to a new object, but this doesn't affect the original reference

  b.size = 100;

}

Box box = new Box();

box.size = 10;

changeBox(box);

System.out.println(box.size); // Output: 10 (unchanged)

🧠 **Summary**

Concept	**Explanation**
Method Overloading	Multiple methods with same name but different parameters
Argument Passing	Java always passes **copy of value**
Primitives	Copy of actual value is passed
Objects	Copy of reference is passed, so object can be changed inside method

Object as Parameter and returning a Object

✅ 1. Object as Parameter in Java

What does it mean?

- You can **pass an object** of a class **to a method** as a parameter.
- The method can then **access or modify the object's fields** using that reference.

How does it work?

- When you pass an object to a method, a **copy of the reference** (memory address) to the object is passed.
- So both the caller and the method refer to the **same object**.
- Changes made to the object inside the method will affect the original object.

Example:

java

CopyEdit

```
class Person {
   String name;

   Person(String name) {
      this.name = name;
   }
}

class Test {
   // method takes a Person object as parameter
```

```
    void changeName(Person p) {
        p.name = "John";  // modifies the original object's field
    }

    public static void main(String[] args) {
        Person person = new Person("Alice");
        System.out.println("Before: " + person.name);  // Output: Alice

        Test test = new Test();
        test.changeName(person);

        System.out.println("After: " + person.name);   // Output: John
    }
}
```

- The method changeName changed the original person object's name field because both refer to the same object.

✅ 2. Returning an Object from a Method in Java

What does it mean?

- A method can **create and return an object** to the caller.
- This allows the caller to get a new object or any object created inside the method.

Example:

java

CopyEdit

```
class Person {
    String name;
```

```
    Person(String name) {
        this.name = name;
    }
}

class Test {
    // method returns a Person object
    Person createPerson(String name) {
        Person p = new Person(name);  // create new object
        return p;                     // return the object reference
    }

    public static void main(String[] args) {
        Test test = new Test();

        Person person = test.createPerson("Bob");
        System.out.println(person.name);  // Output: Bob
    }
}
```

- The method createPerson creates a new Person object and returns it.
- The caller stores the returned reference and uses the object.

🧠 **Summary**

Concept	Explanation
Object as parameter	Passing an object reference to a method, allowing method to modify the object

Concept	Explanation
Returning an object	Method creates or obtains an object and returns its reference to the caller

Static

✅ What is static in Java?

- The keyword **static** means **“belongs to the class, not to instances (objects)”**.
- When a member (variable or method) is declared static, it means there is **only one copy shared by all objects** of that class.
- You **don't need to create an object** to access static members; you can use the class name directly.

🔷 Where can you use static?

- **Static variables (also called class variables)**
- **Static methods**
- **Static blocks**
- **Static nested classes**

✅ 1. Static Variables

- A **static variable** is shared among all instances (objects).
- If one object changes the value, it affects all other objects.

Example:

java

CopyEdit

```
class Counter {
   static int count = 0;  // static variable

   Counter() {
      count++;         // increments when new object is created
   }
}
```

```
public class Test {
   public static void main(String[] args) {
      Counter c1 = new Counter();
      Counter c2 = new Counter();
      Counter c3 = new Counter();

      System.out.println(Counter.count);  // Output: 3
   }
}
```

- Even though we created 3 objects, there is **only one count variable shared** among them.
- Each constructor call increments the same static count.

✅ **2. Static Methods**

- Static methods belong to the **class**, not objects.
- You can **call static methods without creating an object**.
- Static methods **cannot access non-static (instance) variables or methods directly** because those belong to objects.

Example:

java

CopyEdit

```
class MathUtil {
   static int square(int n) {
      return n * n;
   }
}
```

```
public class Test {
   public static void main(String[] args) {
      int result = MathUtil.square(5);  // calling static method without object
      System.out.println(result);       // Output: 25
   }
}
```

✅ **3. Static Block**

- A **static block** runs **once when the class is loaded**.
- Useful to initialize static variables or perform one-time setup.

Example:

java

CopyEdit

```
class Example {
   static int data;

   static {
      data = 100;          // initialize static variable
      System.out.println("Static block executed");
   }
}

public class Test {
   public static void main(String[] args) {
      System.out.println("Data = " + Example.data);
   }
}
```

Output:

java

CopyEdit

Static block executed

Data = 100

✅ 4. Static Nested Classes

- A **static nested class** is a class defined inside another class with static keyword.
- It **can be accessed without an instance** of the outer class.

Example:

java

CopyEdit

```
class Outer {
   static class Inner {
      void display() {
         System.out.println("Inside static nested class");
      }
   }
}

public class Test {
   public static void main(String[] args) {
      Outer.Inner obj = new Outer.Inner();
      obj.display();
   }
}
```

🧠 Summary

Feature	Description	Example usage
Static Variable	Shared by all objects of the class	static int count
Static Method	Called using class name, no object needed	MathUtil.square(5)
Static Block	Runs once when class loads	Initialize static variables
Static Nested Class	Nested class accessed without outer class instance	Outer.Inner obj = new Outer.Inner()

Access modofire

✅ **What is Access Control in Java?**

- Access control decides **who can access members (variables, methods) of a class**.
- It is controlled using **access modifiers**.
- Helps **protect data and implementation details** from unwanted access.

🔷 **Four Access Modifiers in Java**

Modifier	Where Accessible?	Meaning
public	Anywhere (any class, package, subclass, everywhere)	Fully accessible
private	Only inside the **same class**	Not accessible outside the class
protected	Same package + subclasses (even in different packages)	Accessible to subclasses and package classes
(default)	Same package only (when no modifier is used)	Package-private (default access)

---

1. public

- Members declared public are accessible **from anywhere**.
- Usually used for methods and variables you want to expose.

Example:

java

CopyEdit

```
public class Person {
   public String name;   // accessible everywhere


   public void display() {
```

```
    System.out.println("Name: " + name);
  }
}
```

---

2. private

- Members declared private are accessible **only inside the same class**.
- Used to **hide internal details** (data hiding).
- Cannot be accessed directly from outside.

Example:

java

CopyEdit

```
class Person {
  private int age;   // only accessible inside Person class

  void setAge(int a) {
    if (a > 0) age = a;  // setter method to control access
  }

  int getAge() {
    return age;        // getter method to access private variable
  }
}
```

---

3. protected

- Accessible **within the same package** and **in subclasses** even if subclasses are in different packages.
- Useful for inheritance.

Example:

java

CopyEdit

```
package animals;


public class Animal {
   protected void sound() {
      System.out.println("Animal makes sound");
   }
}
```

java

CopyEdit

```
package animals;


public class Dog extends Animal {
   void bark() {
      sound();  // can access protected method from parent class
   }
}
```

---

4. Default Access (Package-Private)

- When **no modifier** is specified, access is limited to **same package only**.
- Not accessible outside the package.

Example:

java

CopyEdit

```
class Car {
   void start() {
      System.out.println("Car started");
```

```
  }
}
```

- The start method can only be called by other classes in the same package.

🧠 **Summary Table**

Modifier	Inside Class	Same Package	Subclass (diff package)	Anywhere (world)
public	Yes	Yes	Yes	Yes
private	Yes	No	No	No
protected	Yes	Yes	Yes	No
default	Yes	Yes	No	No

Why is Access Control Important?

- **Protects data** (encapsulation).
- Helps **maintain code integrity**.
- Controls **who can modify or use** your class members.
- Makes your program more secure and modular

Final

✅ **What is final in Java?**

- The keyword **final** is used to **declare constants or restrict something** so it cannot be changed or overridden.
- It can be applied to **variables**, **methods**, and **classes**.

How does final work in different contexts?

1. Final Variables

- When you declare a variable as final, **its value cannot be changed once assigned**.
- It means the variable becomes a **constant**.

Example:

java

CopyEdit

```
final int MAX_VALUE = 100;

MAX_VALUE = 200;  // Error! Can't change a final variable.
```

- You **must assign** a final variable a value either at the time of declaration or in the constructor (if instance variable).
- After that, any attempt to change it will cause a **compile-time error**.

---

2. Final Methods

- When a method is declared final, it **cannot be overridden** by subclasses.
- Useful when you want to **prevent changes to method behavior** in child classes.

Example:

java

CopyEdit

```
class Parent {
   final void show() {
      System.out.println("Final method");
   }
}

class Child extends Parent {
   void show() {  // Error! Can't override final method
      System.out.println("Override attempt");
   }
}
```

---

3. Final Classes

- When a class is declared final, it **cannot be subclassed**.
- Useful when you want to **prevent inheritance**.

Example:

java

CopyEdit

```
final class Animal {
   void sound() {
      System.out.println("Animal sound");
   }
}

class Dog extends Animal {  // Error! Can't inherit from final class
}
```

🧠 **Summary**

Usage	Effect
final variable	Value cannot be changed (constant)
final method	Method cannot be overridden
final class	Class cannot be extended (no inheritance)

Why use final?

- **Immutability:** For constants and values that must not change.
- **Security:** Prevent accidental modification of methods or classes.
- **Performance:** JVM can optimize final variables and methods.

Nested class and inner class

✅ **Nested Classes in Java**

- A **nested class** is a class defined **inside another class**.
- It helps logically group classes that are only used in one place.
- There are two main types of nested classes:
    1. **Static Nested Classes**
    2. **Inner Classes (non-static nested classes)**

---

1. Static Nested Class

- A **static nested class** is declared with the static keyword inside another class.
- It behaves like a **static member** of the outer class.
- You **do not need an instance** of the outer class to create an object of the static nested class.
- It **cannot access instance variables or methods** of the outer class directly (because it's static).

Example:

java

CopyEdit

```
class Outer {
   static int data = 30;


   static class StaticNested {
      void display() {
         System.out.println("Data is " + data);  // Can access static members of outer class
      }
   }
}
```

```
public class Test {
    public static void main(String[] args) {
        Outer.StaticNested nested = new Outer.StaticNested(); // No outer object needed
        nested.display();  // Output: Data is 30
    }
}
```

---

2. Inner Class (Non-static Nested Class)

- An **inner class** is a **non-static nested class**.
- It **belongs to an instance of the outer class**.
- You **need an instance** of the outer class to create an object of the inner class.
- It **can access all members** (including private) of the outer class directly.

Example:

java

CopyEdit

```
class Outer {
    int data = 10;

    class Inner {
        void display() {
            System.out.println("Data is " + data);  // Can access outer class instance variable
        }
    }
}

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

```
        Outer outer = new Outer();        // Create outer class object
        Outer.Inner inner = outer.new Inner(); // Create inner class object using outer instance
        inner.display();                  // Output: Data is 10
    }
}
```

Why use Nested and Inner Classes?

- Helps **group classes logically** and improve code organization.
- Inner classes can **access private members** of the outer class, useful in event handling or callbacks.
- Static nested classes are useful when the nested class is **independent of the outer class instance**.

🧠 **Summary Table**

Type	Declared With	Needs Outer Class Instance?	Can Access Outer Instance Members?	Example Usage
Static Nested Class	static	No	No (only static members)	Utility/helper classes
Inner Class	No static	Yes	Yes	Accessing outer class members

Command line arguments, variable - length arguments

✅ 1. Command Line Arguments in Java

What are Command Line Arguments?

- When you run a Java program, you can **pass extra information (arguments)** to the program via the command line.
- These arguments are received by the main method as an **array of Strings**.

How do they work?

- The main method is defined as:

java

CopyEdit

public static void main(String[] args)

- args is an array of Strings that stores command line arguments.
- You can access each argument using args[0], args[1], etc.

Example:

java

CopyEdit

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Number of arguments: " + args.length);
    for (int i = 0; i < args.length; i++) {
      System.out.println("Argument " + i + ": " + args[i]);
    }
  }
```

}

How to run with arguments?

If you compile Test.java and run:

bash

CopyEdit

java Test Hello World 123

Output:

yaml

CopyEdit

Number of arguments: 3

Argument 0: Hello

Argument 1: World

Argument 2: 123

Why use command line arguments?

- To give input to the program without changing the code.
- Useful for configuration, filenames, or user inputs when running the program.

✅ 2. Variable-Length Arguments (Varargs)

What are Varargs?

- Varargs let you pass **a variable number of arguments** of the **same type** to a method.
- Instead of defining multiple methods with different numbers of parameters, you can use varargs.

Syntax:

java

CopyEdit

void methodName(type... varName)

- The ... means **"zero or more arguments"**.
- Inside the method, varargs behave like an **array**.

Example:

java

CopyEdit

```
class Test {
   // Method with varargs
   static void printNumbers(int... numbers) {
      System.out.println("Number of arguments: " + numbers.length);
      for (int num : numbers) {
         System.out.print(num + " ");
      }
      System.out.println();
   }

   public static void main(String[] args) {
      printNumbers(1, 2, 3);
      printNumbers(10);
      printNumbers();  // no arguments
   }
}
```

Output:

javascript

CopyEdit

```
Number of arguments: 3
```

1 2 3

Number of arguments: 1

10

Number of arguments: 0

Rules for varargs:

- Varargs must be the **last parameter** in the method.
- You can have **only one varargs parameter** in a method.

🧠 **Summary**

Concept	Explanation	Example
Command Line Args	Input passed to program via command line	public static void main(String[] args)
Varargs	Method parameter that accepts variable number of arguments	void print(int... nums)