2024

CORE_JAVA

TISON C SUNNY

JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in **which Java bytecode can be executed**. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. **JVM, JRE, and JDK are platform dependent** because the configuration of each OS is different from each other. However, **Java is platform independent.** There are three notions of the JVM: *specification*, *implementation*, and *instance*.

The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. **The Java Runtime Environment is a set of software tools which are used for developing Java applications**. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. **It contains a set of libraries + other files that JVM uses at runtime.**

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.

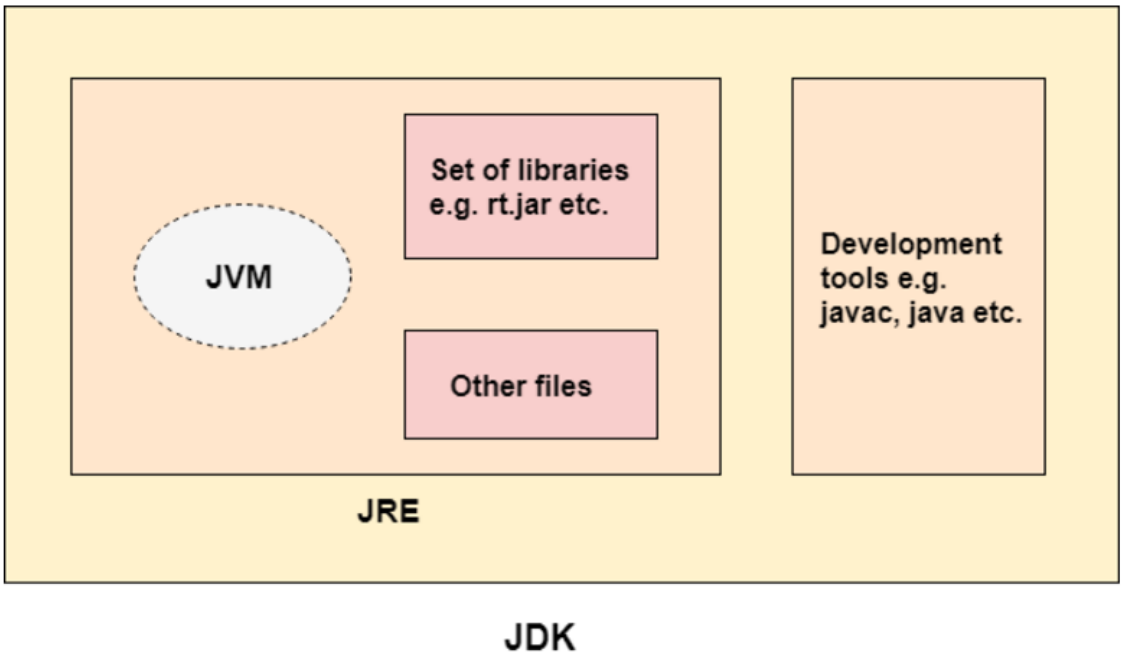
JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. **It contains JRE + development tools.**

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



Why Java platform independent

Java is platform-independent because it uses **bytecode**, an intermediate format generated by the Java compiler. This bytecode can run on any platform with a **Java Virtual Machine (JVM)**, which translates the bytecode into platform-specific machine code. This ensures the principle of "Write Once, Run Anywhere" (WORA).

Other language which is platform dependent , the compiler convert to Machin code that makes platform dependent. **In java compiler only create byte code and byte code can run on JVM which create Machin code**.

Write Once, Run Anywhere (WORA)

- Developers write Java code once, compile it into bytecode, and this bytecode can be run on any system without modification as long as it has a JVM.

Java Variables

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: **local, instance and static.**

There are two types of data types in Java: primitive and non-primitive.

1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

Let's understand in detail about the two major data types of Java in the following paragraphs.

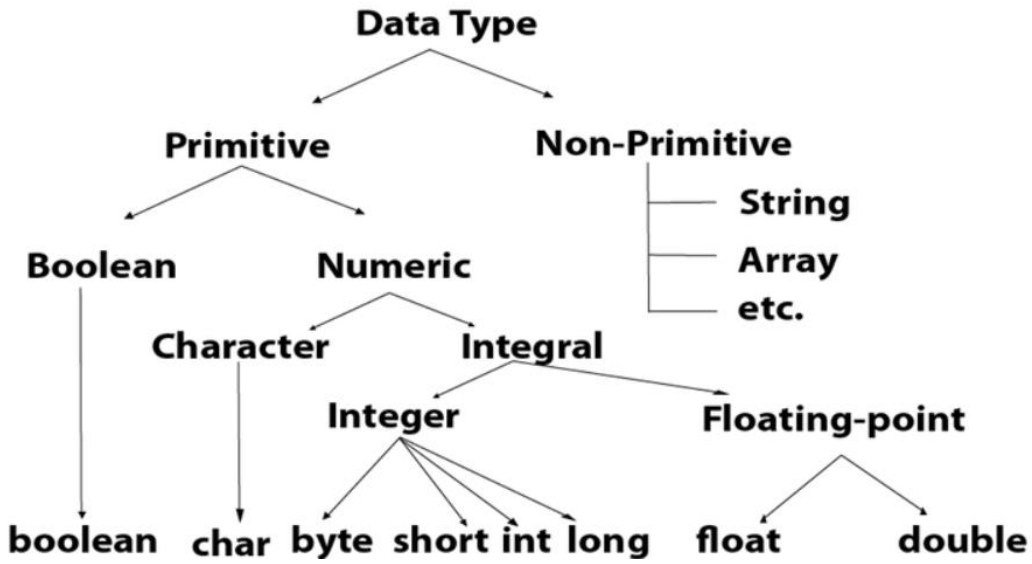
Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

Java Primitive data types:

1. boolean data type
2. byte data type
3. char data type
4. short data type
5. int data type
6. long data type

7. float data type
8. double data type



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Non-Primitive Data Types in Java

In Java, non-primitive data types, also known as reference data types, are used to store complex objects rather than simple values. Unlike **primitive data types that store the actual values, reference data types store references or memory addresses that point to the location of the object in memory.** This distinction is important because it affects how these data types are stored, passed, and manipulated in Java programs.

Class

One common non-primitive data type in Java is the class. Classes are used to create objects, which are instances of the class. A class defines the properties and behaviors of objects, including variables (fields) and methods. For example, you might create

a **Person** class to represent a person, with variables for the person's name, age, and address, and methods to set and get these values.

Interface

Interfaces are another important non-primitive data type in Java. An interface defines a contract for what a class implementing the interface must provide, without specifying how it should be implemented. Interfaces are used to achieve abstraction and multiple inheritance in Java, allowing classes to be more flexible and reusable.

Arrays

Arrays are a fundamental non-primitive data type in Java that allow you to store multiple values of the same type in a single variable. Arrays have a fixed size, which is specified when the array is created, and can be accessed using an index. Arrays are commonly used to store lists of values or to represent matrices and other multi-dimensional data structures.

Enum

Java also includes other non-primitive data types, such as enums and collections. Enums are used to define a set of named constants, providing a way to represent a fixed set of values. Collections are a framework of classes and interfaces that provide dynamic data structures such as lists, sets, and maps, which can grow or shrink in size as needed.

Overall, non-primitive data types in Java are essential for creating complex and flexible programs. They allow you to create and manipulate objects, define relationships between objects, and represent complex data structures. By understanding how to use non-primitive data types effectively, you can write more efficient and maintainable Java code.

Type Casting

Type casting in Java is the process of explicitly converting a variable from one data type to another. It is usually performed between compatible types (e.g., int to double).

Types of Casting:

1. **Widening Casting (Implicit)**: Conversion
    - Converts a smaller type to a larger type.
    - Safe and done automatically by Java.
    - Example:

      ```
      int num = 10;
      double result = num; // int to double
      ```

2. **Narrowing Casting (Explicit)**:

- Converts a larger type to a smaller type.
- May lead to data loss and requires explicit casting.
- Example:

```
double num = 10.5;
int result = (int) num; // double to int (data loss: fractional part removed)
```

Operators in Java

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a Boolean

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

```
public class OperatorExample{
public static void main(String args[]){
System.out.println(10<<2);//10*2^2=10*4=40
System.out.println(10<<3);//10*2^3=10*8=80
System.out.println(20<<2);//20*2^2=20*4=80
```

```
    System.out.println(15<<4);//15*2^4=15*16=240
}}
```

Java Right Shift Operator

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

```
public OperatorExample{
public static void main(String args[]){
System.out.println(10>>2);//10/2^2=10/4=2
System.out.println(20>>2);//20/2^2=20/4=5
System.out.println(20>>3);//20/2^3=20/8=2
}}
```

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a<c);//false && true = false
System.out.println(a<b&a<c);//false & true = false
}}
```

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a>b||a<c);//true || true = true
System.out.println(a>b|a<c);//true | true = true
//|| vs |
```

```
System.out.println(a>b||a++<c);//true || true = true
System.out.println(a);//10 because second condition is not checked
System.out.println(a>b|a++<c);//true | true = true
System.out.println(a);//11 because second condition is checked
}}
```

Java Ternary Operator

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

```
public class OperatorExample{
public static void main(String args[]){
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Java Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Java Control Statements | Control Flow in Java

Java provides three types of control flow statements.

1. Decision Making statements
    - if statements
    - switch statement
2. Loop statements
    - do while loop
    - while loop
    - for loop
    - for-each loop
3. Jump statements
    - break statement
    - continue statement

If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(condition) {
statement 1; //executes when condition is true
}
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

4)Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
switch (expression){
   case value1:
    statement1;
    break;
   .
   .
   .
   case valueN:
    statementN;
    break;
   default:
    default statement;
}
```

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

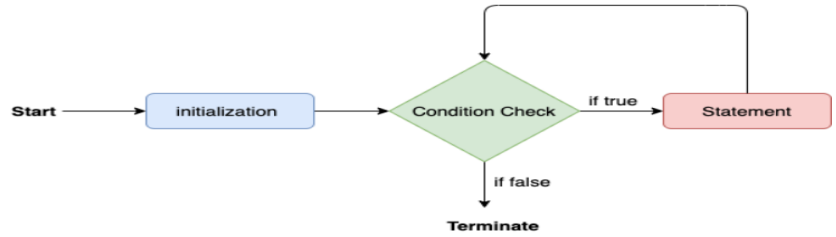
Let's understand the loop statements one by one.

Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```

The flow chart for the for-loop is given below.



we want to execute the block of code.

Java while loop

The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

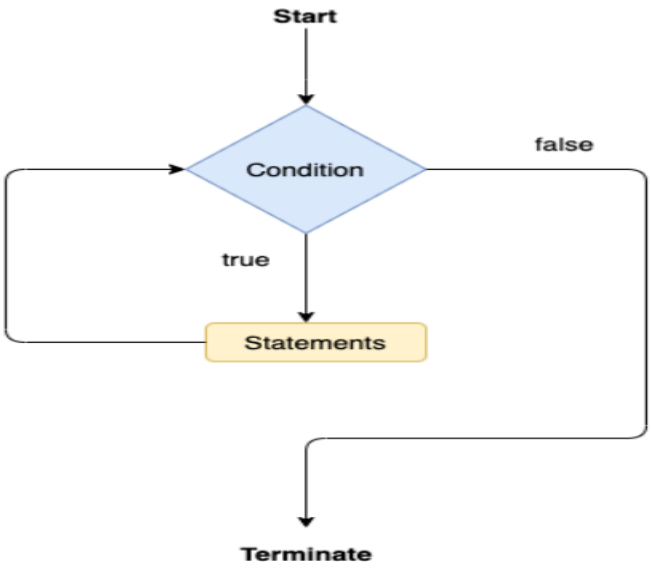
It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

The syntax of the while loop is given below.

```
while(condition){
//looping statements
}
```

The flow chart for the while loop is given in the following image.

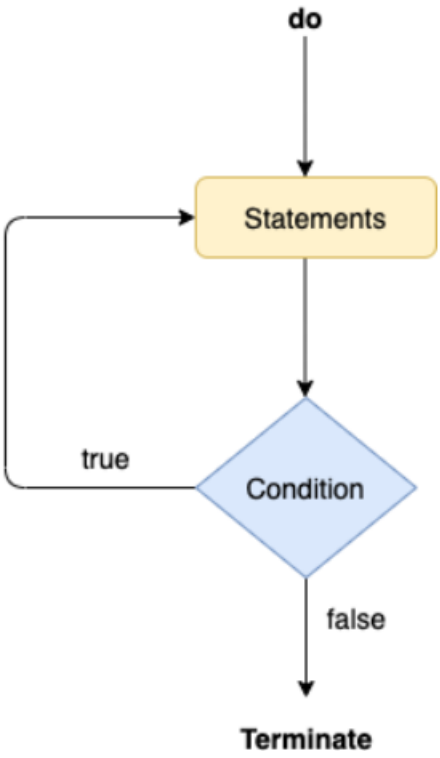


Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

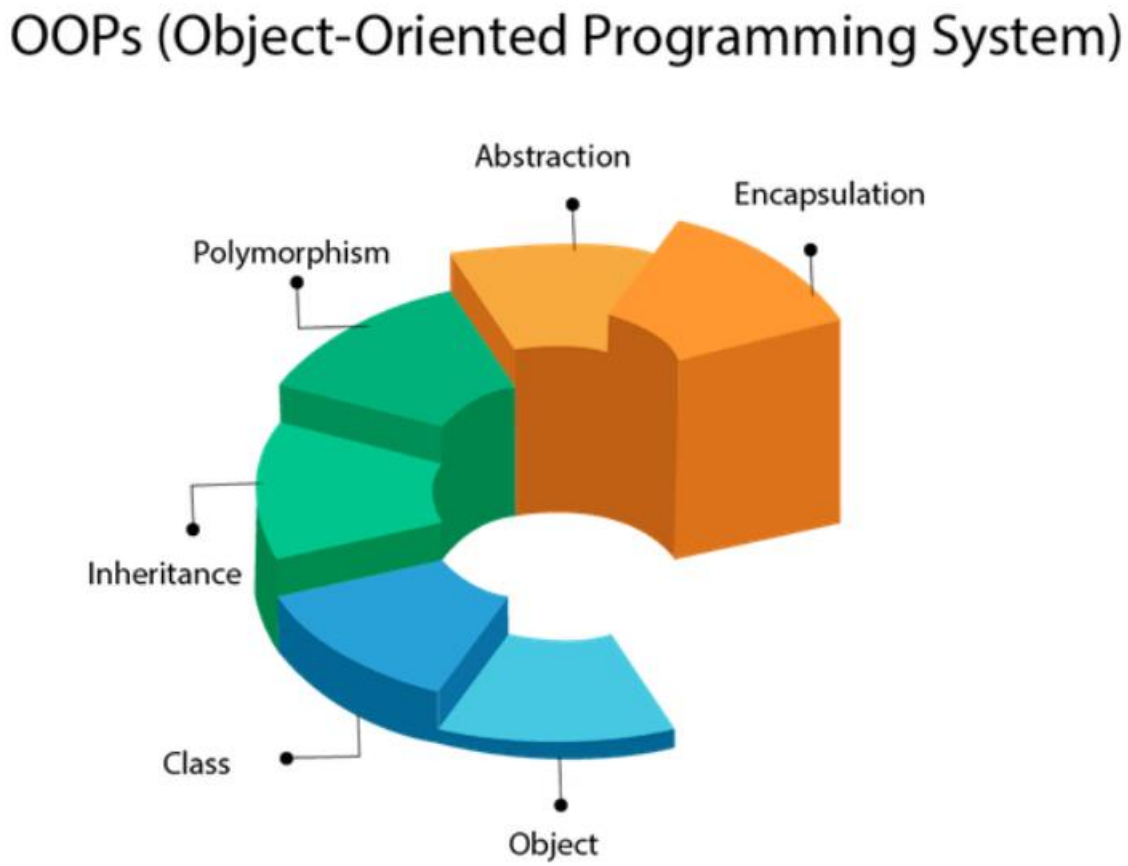
It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

```
do
{
//statements
} while (condition);
```



OOPs (Object-Oriented Programming)

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:



Class

A **class** in Java is a blueprint or template for creating objects. It defines the structure (fields/attributes) and behaviour (methods) that the objects of the class will have. Class does not consume any space.

Example:

```java
class Car {
    String brand; // Attribute
    void drive() { // Method
        System.out.println("Car is driving");
    }
}
```

Object

An **object** in Java is an instance of a class. It is a real-world entity with state (attributes) and behaviour (methods) as defined by the class.

Example:

```java
Car myCar = new Car(); // Creating an object of the Car class
myCar.brand = "Toyota"; // Setting state
myCar.drive(); // Invoking behavior
```

Polymorphism in Java

Polymorphism is the ability of an object to take on many forms. It allows a single action to behave differently based on the object performing it. In Java, polymorphism is primarily achieved through **method overloading** and **method overriding**.

Types of Polymorphism

Compile-Time Polymorphism (Static Polymorphism):

- Achieved through **method overloading**.
- The method to be executed is determined at compile time.

Method Overloading

Definition: Method overloading occurs when two or more methods in the same class have the **same name** but different **parameter lists** (type, number, or order of parameters).

Key Points:

- Happens **within the same class**.
- Determined at **compile time** (Compile-time Polymorphism).
- Does **not depend** on the return type.

- **Example**:

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

Run-Time Polymorphism (Dynamic Polymorphism):

- Achieved through **method overriding**.
- The method to be executed is determined at runtime based on the object type.

Method Overriding

- **Definition**: Method overriding occurs when a subclass provides a **specific implementation** of a method that is already defined in its superclass.
- **Key Points**:
  - Happens **across classes** in an inheritance hierarchy.
  - Determined at **runtime** (Runtime Polymorphism).
  - The method must have the **same name, parameters, and return type**.
  - Requires the use of the `@Override` annotation (optional but recommended).
  - The overridden method in the superclass must be `public` or `protected` (not private).

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

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.sound(); // Output: Dog barks
    }
}
```

Key Differences

Aspect	Method Overloading	Method Overriding
Definition	Same method name, different parameter list.	Same method name and parameters, but in a subclass.
Classes Involved	Same class.	Superclass and subclass.
Polymorphism Type	Compile-time polymorphism.	Runtime polymorphism.
Return Type	Can be different.	Must be the same or covariant.
Access Modifier	No restrictions.	Cannot reduce visibility.

Array in Java

An **array** in Java is a data structure used to store multiple elements of the **same data type** in a single variable. It provides a way to group and manage similar types of data efficiently.

Key Features

1. **Fixed Size**: The size of an array is defined at the time of creation and cannot be changed dynamically.
2. **Indexed Access**: Array elements are accessed using an index, starting from 0.
3. **Homogeneous Data**: Arrays store elements of the same data type.
4. **Continuous Memory**: Array elements are stored in contiguous memory locations.

Combined Declaration and Instantiation:

java Copy code

```java
dataType[] arrayName = new dataType[size];
```

Example

```java
// Declare and initialize an array
int[] numbers = new int[5];

// Assign values
numbers[0] = 10;
numbers[1] = 20;

// Access values
System.out.println(numbers[0]); // Output: 10
System.out.println(numbers[1]); // Output: 20
```

Types of Arrays

1. **Single-Dimensional Array**:

   Example:

   ```java
   int[] arr = {10, 20, 30};
   System.out.println(arr[1]); // Output: 20
   ```

2. **Multi-Dimensional Array**:

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

Jagged Arrays in Java

A **jagged array** (or irregular array) is a **multi-dimensional array** where the rows have **different lengths**. Unlike regular 2D arrays, jagged arrays allow you to have variable column sizes for each row.

Key Characteristics

1. Rows can have **different numbers of columns**.
2. Useful for representing irregular data structures like triangle matrices or sparse data.
3. Declared as an array of arrays.

```
public class Main {
    public static void main(String[] args) {
        // Declare and initialize a jagged array
        int[][] jaggedArray = new int[3][]; // 3 rows

        // Define column sizes for each row
        jaggedArray[0] = new int[2]; // Row 0 has 2 columns
        jaggedArray[1] = new int[4]; // Row 1 has 4 columns
        jaggedArray[2] = new int[3]; // Row 2 has 3 columns

        // Assign values
        jaggedArray[0][0] = 1;
        jaggedArray[0][1] = 2;
        jaggedArray[1][0] = 3;
        jaggedArray[1][1] = 4;
```

```
        // Print jagged array
        for (int i = 0; i < jaggedArray.length; i++) {
            for (int j = 0; j < jaggedArray[i].length; j++) {
                System.out.print(jaggedArray[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Disadvantages of Arrays in Java

1. **Fixed Size**
   - The size of an array is static and must be defined at the time of creation.
   - It cannot grow or shrink dynamically, leading to potential memory wastage or insufficient storage.
2. **Homogeneous Data**
   - Arrays can only store elements of the same data type.
   - For storing heterogeneous data, you need to use **collections like ArrayList or HashMap.**
3. **No Built-in Methods**

- Arrays lack built-in methods for operations like adding, removing, or searching elements efficiently.
- Collections like **ArrayList provide these features.**

4. **Memory Usage**
   - Arrays use contiguous memory, which can cause issues when a large block of memory is unavailable, even if there is enough fragmented memory.
5. **Performance for Insertion/Deletion**
   - Inserting or deleting an element in an array requires shifting elements, which is time-consuming, especially for large arrays.
6. **Type-Specific**
   - Java arrays are type-specific, and you cannot store objects of different types in a single array.
7. **Lack of Flexibility**
   - Arrays are not resizable, unlike data structures like ArrayList, LinkedList, or Vector.
8. **No Direct Sorting/Searching Support**
   - Arrays require manual coding or external libraries to perform tasks like sorting or searching efficiently, whereas collections have built-in support.

Array of Objects in Java

An **array of objects** is a collection where each element is an object. This is useful for storing and managing multiple objects of the same class.

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

class ArrayOfObject {

    public static void main(String args[]) {

        Student s1 = new Student();
        s1.rollno = 10;
        s1.name = "Tison";

        Student s2 = new Student();
        s2.rollno = 20;
        s2.name = "Giya";

        Student s3 = new Student();
        s3.rollno = 30;
        s3.name = "Don";

        Student students[] = new Student[3];
        students[0] = s1;
        students[1] = s2;
        students[2] = s3;
        for (Student s : students)
            System.out.println(s.rollno + " " + s.name);
    }
}
```

Mutable vs Immutable Strings in Java

In Java, strings can be **mutable** or **immutable**, depending on the type of class used to handle the string.

Immutable Strings

1. **Definition**: Immutable strings cannot be modified after they are created. Any operation that alters a string creates a new object rather than modifying the original string.
2. **Class Used**: String class.
3. **Characteristics**:
    - String objects are **stored in the String Pool** for memory efficiency.
    - Modifying a string (e.g., concatenation) creates a new string object.

4. **Example**:

```java
String str = "Hello";
str = str + " World"; // A new String object is created
System.out.println(str); // Output: Hello World
```

4. **Advantages**:

- **Thread-safe**: No risk of data corruption when accessed by multiple threads.
- Efficient for scenarios like keys in a `HashMap` where immutability is desirable.

6. **Disadvantages**:

- Frequent modifications result in **many intermediate objects** and may cause performance overhead.

Mutable Strings

1. **Definition**: Mutable strings can be modified directly without creating a new object.
2. **Classes Used**: StringBuilder and StringBuffer.
3. **Characteristics**:
    - Modifications happen within the same object, reducing overhead.
    - StringBuffer is **synchronized** (thread-safe), while StringBuilder is **not synchronized** (faster in single-threaded operations).

4. **Example**:

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

5.**Advantages**:

- Efficient for scenarios requiring frequent modifications (e.g., loops or concatenations).
- Reduces memory usage since no new objects are created.

6. **Disadvantages**:

- **StringBuilder** is not thread-safe, so it requires external synchronization in multi-threaded environments.

Key Differences

Feature	Immutable String (`String`)	Mutable String (`StringBuilder` / `StringBuffer`)
Modification	Creates a new object.	Modifies the same object.
Thread Safety	Thread-safe inherently.	`StringBuffer` is thread-safe, `StringBuilder` is not.
Performance	Slower for frequent changes.	Faster for frequent changes.
Memory Efficiency	Uses more memory with frequent changes.	Efficient as no intermediate objects are created.
Use Case	Ideal for fixed or read-only strings.	Ideal for strings with frequent modifications.

When to Use Which?

- Use **String** for immutable operations, like string keys in maps or constant strings.
- Use **StringBuilder** for single-threaded programs requiring frequent modifications.
- Use **StringBuffer** in multi-threaded environments needing synchronized operations.

Static Variable

1. **Definition**: A static variable is shared across all objects of the class. It belongs to the class, not the instances.
2. **Key Points**:
    - **One copy per class**: All objects share the same value of the static variable.
    - **Memory allocation**: Static variables are allocated memory once, in the **method area**, at the time of class loading.
    - **Access**: Can be accessed directly using the class name or through an object reference (though not recommended).

Static Method

1. **Definition**: A static method belongs to the class rather than any specific object and can be called without creating an instance of the class.
2. **Key Points**:
    - **No need for an object**: Static methods can be called directly using the class name.
    - **Access restrictions**:
        - Static methods **cannot access non-static variables or methods** directly (since they belong to the object).
        - They can only access other static variables and methods.
    - **Use case**: Utility or helper methods (e.g., Math.max() or Math.sqrt()).

```java
class Mobile {

    String brand;
    int price;
    static String name;

    public void show() {

        System.out.println("Brand is: " + brand + " price is: " + price + " name is:" + name);
    }

    public static void show1(String brand,int price ) {

        System.out.println("Brand is: " + brand + " price is: " + price + " name is:" + name);
    }

}

class StaticVariable {

    public static void main(String args[]) {
        Mobile s1 = new Mobile();

        s1.brand = "Apple";
        s1.price = 1500;

        Mobile.name = "Mobile phone";
        s1.show();

        Mobile s2 = new Mobile();
        s2.brand = "Samsaung";
        s2.price = 1000;
        s2.show();

        Mobile.show1(s1.brand, s1.price);
        Mobile.show1(s2.brand, s2.price);
    }
}
```

Key Differences

Feature	Static Variable	Static Method
Belongs To	Class (shared by all objects).	Class (can be called without objects).
Access	Accessed using `ClassName.variable`.	Accessed using `ClassName.method()`.
Interaction	Can be accessed by both static and non-static methods.	Can only access static variables/methods.
Memory	Allocated once during class loading.	Exists in memory only when invoked.

When to Use

- Use a **static variable** for class-level properties that should be shared among all instances (e.g., a counter, configuration settings).
- Use a **static method** for utility-like functionality that doesn't depend on instance variables (e.g., mathematical calculations, helper methods).

Static Block in Java

A **static block** in Java is a block of code that is executed **once** when the class is loaded into memory. It is used to initialize **static variables** or perform any startup tasks for the class.

Key Points

1. **Execution Time**:
    - Executes when the class is loaded into the JVM.
    - Executes **before the main method** or any instance creation.
2. **One-Time Execution**: Static blocks run only once, regardless of how many objects of the class are created.
3. **Purpose**:
    - Initialize static variables.
    - Perform tasks that need to be done only once for the class.

When to Use Static Blocks

1. To initialize **static variables** or **constants**.
2. To load **configuration settings** or **files**.
3. To execute **class-level startup logic** (e.g., loading native libraries).

Static Block vs Static Method

Feature	Static Block	Static Method
Execution	Automatically executed when the class is loaded.	Requires an explicit call.
Purpose	Used for class-level initialization tasks.	Used for reusable utility logic.
Number of Executions	Runs only once per class load.	Can be called multiple times.

Limitations

1. Cannot use **non-static variables or methods** directly.
2. Overuse of static blocks may reduce code readability.

How to Load a Class in Java

In Java, a class is loaded into memory by the **ClassLoader** when it is required for execution. There are multiple ways to load a class programmatically or automatically

1. Automatically Loaded Classes

Java automatically loads classes when:

- You create an object of a class using `new`.
- You call a static method or access a static variable.
- The JVM encounters the `main()` method in a class.

2. Using `Class.forName()`

The `Class.forName()` method dynamically loads the class at runtime.

Using `ClassLoader.loadClass()`

The `ClassLoader` class can be used to load a class programmatically.

3. Using `ClassLoader.loadClass()`

The `ClassLoader` class can be used to load a class programmatically.

4. Using `Object.getClass()`

If you have an instance of an object, you can get its class reference using the `getClass()` method.

Encapsulation in Java

Encapsulation is a fundamental principle of object-oriented programming (OOP) that involves bundling data (fields) and methods (functions) that operate on the data into a single unit (a class). It restricts direct access to certain components of an object and allows controlled access through defined methods.

Key Features of Encapsulation

1. **Data Hiding**:
    - Internal representation (fields) of an object is hidden from the outside world.
    - Only selected information is accessible via public methods.
2. **Controlled Access**:
    - Provides getter and setter methods to access or update private fields.
    - Ensures data integrity and validation.
3. **Improved Code Maintainability**:
    - Changes to the internal implementation do not affect external code interacting with the class.
4. **Enhanced Security**:
    - Protects data from unintended modifications.

Implementation of Encapsulation

1. **Declare Fields as `private`**:
    - This restricts direct access to the fields from outside the class.
2. **Provide `public` Getter and Setter Methods**:
    - These methods control and validate access to the private fields.

Advantages of Encapsulation

1. **Improves Security**: Protects sensitive data from unintended access or modification.
2. **Data Validation**: Ensures that only valid data is stored in the fields (e.g., through validation in setters).
3. **Ease of Maintenance**: Changes to internal implementation don't affect external code.
4. **Reusability**: Encapsulated classes are modular and easier to reuse.

```java
class Human {

    private int age;
    private String name;

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

    public int getAge() {
        return age;
    }

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

    public String getName() {
        return name;
    }

}

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

        Human obj = new Human();
        obj.setAge(32);
        obj.setname("Don");
        System.out.println("Name is: " + obj.getName() + " Age is: " + obj.getAge());

    }
}
```

Getters and Setters in Java

Getters and **setters** are methods used to access and update the private fields of a class. They provide controlled access to the class's data by following the principles of **encapsulation**.

Why Use Getters and Setters?

1. **Encapsulation**:
    - They ensure that the internal representation of an object is hidden from the outside world.
2. **Data Validation**:
    - Setters allow you to add validation logic before changing a field's value.
3. **Flexibility**:
    - You can modify the implementation of a getter or setter without changing the external interface.
4. **Consistency**:
    - Promotes a standard way to access and modify data across your code.

this Keyword in Java

The this keyword in Java is a reference variable that refers to the **current object** of the class. It is commonly used to resolve naming conflicts and to access the instance variables, methods, or constructors of the current object

Key Uses of this

1. **Refer to Instance Variables**:
    - When a method or constructor has a local variable with the same name as an instance variable, this is used to distinguish between them.
2. **Call Another Constructor (Constructor Chaining)**:
    - Used to call one constructor from another in the same class.
3. **Pass the Current Object**:
    - Pass the current object as an argument to another method or constructor.
4. **Return the Current Object**:
    - Return the object itself from a method.
5. **Call an Instance Method**:
    - Call a method of the current class explicitly.

Key Points About this

- this cannot be used in static methods because static methods belong to the class, not any specific object.
- It is automatically passed as a reference to instance methods of a class.
- Improves code readability and resolves ambiguity when variable names conflict.

What is a Constructor in Java?

A **constructor** in Java is a special method used to initialize objects. It is called automatically when an object of a class is created. The constructor typically initializes the instance variables of the class and allocates memory for the object.

Key Features of Constructors

1. **Same Name as the Class**: The constructor's name must exactly match the class name.
2. **No Return Type**: Constructors do not have a return type, not even void.
3. **Automatically Invoked**: The constructor is called automatically when an object is instantiated.
4. **Types of Constructors**:
    - **Default Constructor**: A no-argument constructor automatically provided by Java if no constructor is defined in the class.
    - **Parameterized Constructor**: A constructor with arguments to initialize fields with specific values.

Types of Constructors

1. **Default Constructor**:
    - A no-argument constructor provided by the compiler if no constructor is explicitly defined.
    - Used to initialize fields to default values
2. **No-Argument Constructor**:

    - Explicitly defined by the programmer.
    - Similar to the default constructor but can include logic.

```java
class Human2{

    private int age;
    private String name;
    public Human2() {    //default constructor
        this.age=32;
        this.name="Tison";
    }

    public Human2(int age, String name) {  //parameterized constructor
        this.age=age;
        this.name=name;
    }

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

    public int getAge() {
        return age;
    }

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

    public String getName() {
        return name;
    }

}
class Constructor{

    public static void main(String agrgs[]) {

        Human2 obj=new Human2();
        System.out.println(obj.getAge()+" "+obj.getName());
        Human2 obj2=new Human2(55,"Sunny");
        System.out.println(obj2.getAge()+" "+obj2.getName());
    }

}
```

Naming Conventions in Java

Naming conventions in Java are guidelines to improve the readability and maintainability of code. These conventions make it easier for developers to understand the code and collaborate on projects.

General Guidelines

1. **Names Should Be Meaningful**:
    - Use names that clearly describe the purpose of variables, methods, classes, etc.
    - Avoid single-character names except for loop variables or temporary use (e.g., `i`, `j`).
2. **Avoid Abbreviations**:
    - Use full, descriptive names (e.g., `firstName` instead of `fName`).
3. **Follow Case Sensitivity**:
    - Java is case-sensitive, so follow specific conventions for different elements.

Conventions by Type

1. Classes and Interfaces

- **Convention**: Use **PascalCase** (also known as UpperCamelCase).
- **Rules**:
    - First letter of each word should be capitalized.
    - Should be nouns for classes (e.g., `Student`, `Employee`) and adjectives for interfaces (e.g., `Runnable`, `Serializable`).

Examples:

java Copy code

```java
class StudentDetails {}
interface Runnable {}
```

2. Methods

- **Convention**: Use **camelCase**.
- **Rules**:
    - First letter is lowercase, and subsequent words are capitalized.
    - Use verbs to describe the action the method performs.

Examples:

```java
public void calculateSalary() {}
public int getAge() {}
```

3. Variables

- **Convention**: Use **camelCase**.
- **Rules**:
  - First letter is lowercase, and subsequent words are capitalized.
  - Use meaningful names representing the purpose of the variable.

Examples:

```java
int employeeAge;
String firstName;
```

4. Constants

- **Convention**: Use **UPPERCASE_WITH_UNDERSCORES**.
- **Rules**:
  - All letters should be uppercase.
  - Words are separated by underscores.

Examples:

```java
public static final int MAX_VALUE = 100;
public static final String DEFAULT_NAME = "Guest";
```

5. Packages

- **Convention**: Use **lowercase** and hierarchical structure.
- **Rules**:
  - Typically follows the reversed domain name of the organization (e.g., com.companyname.projectname).
  - Avoid using underscores or special characters.

Examples:

```java
package com.example.myapp;
package org.company.product;
```

6. Variables in Loops

- **Convention**: Use single-letter variables for loop counters.
- **Rules**:
  - Use `i`, `j`, `k` for nested loops.

Example:

```java
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

7. Enum Constants

- **Convention**: Use **UPPERCASE_WITH_UNDERSCORES**.
- **Rules**:
  - Enum constants should be written in uppercase.

Example:

```java
enum Color {
    RED, GREEN, BLUE;
}
```

8. Type Parameters

- **Convention**: Use a single uppercase letter.
- **Rules**:
  - Commonly used letters:
    - `T` for Type
    - `E` for Element
    - `K` for Key
    - `V` for Value

Example:

```java
class GenericClass<T> {}
Map<K, V> map = new HashMap<>();
```

Anonymous Objects in Java

An **anonymous object** in Java is an object that is created without assigning it to a reference variable. It is often used when the object is required only once, making the code concise and efficient.

Characteristics of Anonymous Objects

1. **No Reference**: The object is not stored in a reference variable.
2. **Short-Lived**: The object exists for a short duration and is used immediately after creation.
3. **Syntax**: Created using the `new` keyword followed by the class constructor.

When to Use Anonymous Objects

- **Single-Use Objects**: When the object is needed only once and does not need to be reused later.
- **Short Methods**: When the object's purpose is fulfilled in a single method call.

Examples

1. Calling a Method

You can create and use an anonymous object directly to call a method.

```java
class Calculator {
    public void add(int a, int b) {
        System.out.println("Sum: " + (a + b));
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an anonymous object and call the method
        new Calculator().add(5, 10); // Output: Sum: 15
    }
}
```

In this example, the `Calculator` object is created without a reference variable, used immediately, and then discarded.

Inheritance in a JAVA

Inheritance in Java is a core concept of Object-Oriented Programming (OOP) that allows a class (called a **subclass** or **child class**) to inherit properties and behaviors (fields and methods) from another class (called a **superclass** or **parent class**).

🔹 Basic Syntax

java

Copy Edit

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

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

public class Main {
    public static void main(String[] args) {
        Child obj = new Child();
        obj.display();  // Inherited method
        obj.show();     // Child class method
    }
}
```

🔹 Types of Inheritance in Java

Type	Supported in Java?	Description
Single Inheritance	✅ Yes	One child class inherits from one parent class.
Multilevel Inheritance	✅ Yes	A class inherits from a class which in turn inherits from another class.
Hierarchical Inheritance	✅ Yes	Multiple child classes inherit from a single parent class.
Multiple Inheritance (Classes)	❌ No	Not supported with classes due to ambiguity, but supported using interfaces.

🔹 Example: Multilevel Inheritance

java Copy Edit

```java
class Animal {
    void eat() {
        System.out.println("This animal eats food");
    }
}

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

class Puppy extends Dog {
    void weep() {
        System.out.println("Puppy weeps");
    }
}

public class TestInheritance {
    public static void main(String[] args) {
        Puppy p = new Puppy();
        p.eat();   // from Animal
        p.bark();  // from Dog
        p.weep();  // from Puppy
    }
}
```

🔹 `super` Keyword

Used to:

1. Call the parent class constructor.
2. Access parent class methods/fields.

java Copy Edit

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

class Child extends Parent {
    Child() {
        super();  // calls Parent constructor
        System.out.println("Child Constructor");
    }
}
```

🔹 `this()` Constructor Call

- It **must be the first statement** in the constructor.
- Used for **constructor chaining** within the same class.
- Helps reduce code duplication by reusing constructor logic.

⚠️ Rules

- Both `this()` and `super()` **must be the first statement** in a constructor.
- You **cannot use both** `super()` **and** `this()` **in the same constructor**.

Method Overriding in Java

Method Overriding in Java occurs when a **subclass** provides a specific implementation of a **method** that is already defined in its **superclass**. The method in the child class **must have the same name, return type, and parameters** as the one in the parent class.

Purpose of Method Overriding

- To **achieve runtime polymorphism** (dynamic method dispatch).
- To **customize behavior** inherited from a parent class.
- To allow **subclasses to provide specific functionality**.

🔧 Rules for Method Overriding

Rule	Description
Method name	Must be the **same** as in the parent class
Parameters	Must be the **same** (i.e., same signature)
Return type	Must be the **same or subtype** (covariant return type)
Access modifier	Must be **same or more accessible**
Cannot override	`final`, `static`, or `private` methods

💉 Example of Method Overriding

java

Copy Edit

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

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

public class TestOverride {
    public static void main(String[] args) {
        Animal a = new Dog();  // Polymorphism
        a.sound();             // Calls Dog's sound()
    }
}
```

Access Modifiers in Java

🔐 Access Modifiers in Java

Access modifiers in Java control the **visibility** or **accessibility** of classes, methods, and variables to other classes.

There are **four types** of access modifiers:

Modifier	Class	Package	Subclass	World (Other Packages)
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
no modifier (default)	✅	✅	❌	❌
`private`	✅	❌	❌	❌

Final Keyword in java

The final keyword in Java is used to restrict modification**. It can be applied to** variables**,** methods**, and** classes**, each serving a specific purpose.**

✅ 1. Final Variable

- A **final variable** cannot be changed once assigned.
- It becomes a **constant** (usually written in uppercase by convention).

```java
final int MAX_AGE = 100;
// MAX_AGE = 120; ❌ This will cause a compilation error
```

📌 **Note**: If a final variable is not initialized at declaration, it **must be initialized in the constructor** (especially in the case of instance variables).

✅ 2. Final Method

- A **final method cannot be overridden** by subclasses.
- Used to **prevent altering behavior** of inherited methods.

```java
class Parent {
    final void show() {
        System.out.println("Final method - cannot be overridden");
    }
}

class Child extends Parent {
    // void show() ❌ Not allowed — compile-time error
}
```

✅ 3. Final Class

- A **final class cannot be extended**.
- Used to **prevent inheritance**.

```java
final class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}

// class Car extends Vehicle ❌ Not allowed
```

✅ 4. Final Reference Variable

- A **final object reference** cannot point to a different object, but the object's internal state **can still change**.

```java
final StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");      // ✅ allowed
// sb = new StringBuilder(); ❌ Not allowed
```

Dynamic Method Dispatch in java

Dynamic Method Dispatch is a mechanism in Java that resolves method calls at runtime instead of compile-time. It is a key feature of runtime polymorphism.

✅ What is it?

- It allows a **superclass reference** to refer to a **subclass object**.
- At runtime, Java determines **which method implementation to call** based on the object type (not reference type).

🔧 How it Works

When a subclass overrides a method from its superclass, and you use a **superclass reference** to call the method, Java uses the **actual object's type** to decide which version of the method to execute.

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

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}

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

public class Demo {
    public static void main(String[] args) {
        Animal a;  // superclass reference

        a = new Dog();  // refers to Dog object
        a.sound();      // Dog's sound() is called

        a = new Cat();  // refers to Cat object
        a.sound();      // Cat's sound() is called
    }
}
```

Copy Edit

Upcasting and Downcasting in Java

In Java, casting refers to converting one type to another. When working with inheritance, we often deal with object type casting between a superclass and its subclass.

🔼 Upcasting (Widening)

Upcasting is the process of converting a **subclass type** to a **superclass type**.

✅ Key Points:

- **Always safe**
- Happens **implicitly**
- Used for **runtime polymorphism**

java

Copy Edit

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

class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks");
    }

    void wagTail() {
        System.out.println("Dog wags tail");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();  // 🔼 Upcasting (implicit)
        a.makeSound();         // Output: Dog barks
        // a.wagTail(); ❌ Not accessible
    }
}
```

🔽 Downcasting (Narrowing)

Downcasting is the process of converting a **superclass type** to a **subclass type**.

⚠️ Key Points:

- Must be done **explicitly** using casting
- Can cause `ClassCastException` if the object is not actually an instance of the subclass
- Should be done **only when you're sure** of the object's actual type

💉 Example:

java

Copy Edit

```java
public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();  // Upcasting
        Dog d = (Dog) a;       // 🔽 Downcasting (explicit)
        d.wagTail();           // Output: Dog wags tail
    }
}
```

Wrapper Classes in Java

Wrapper classes in Java are used to wrap primitive data types into objects. This is especially useful when working with collections, generics, or when an object reference is required.

🔹 Why Use Wrapper Classes?

- Primitive types like `int`, `char`, `boolean` can't be used directly in collections (like `ArrayList`, `HashMap`).
- Wrapper classes allow primitives to be treated as objects.
- Useful for features like **autoboxing**, **unboxing**, and working with **null values**.

🔧 List of Wrapper Classes

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

🔄 Autoboxing and Unboxing

✅ **Autoboxing:**

Automatic conversion of **primitive → wrapper**.

java Copy Edit

```java
int x = 10;
Integer obj = x;  // autoboxing
```

✅ **Unboxing:**

Automatic conversion of **wrapper → primitive**.

java Copy Edit

```java
Integer obj = 20;
int y = obj;  // unboxing
```

🚫 Differences from Primitives

Feature	Primitive	Wrapper Class
Can be `null`	❌	✅
Used in collections	❌	✅
Object methods (e.g., `equals()`, `compareTo()`)	❌	✅

Abstract Keyword in Java

The abstract keyword in Java is used to declare **a class or a method** that is incomplete and must be implemented by a subclass.

🔹 1. Abstract Class

An **abstract class** is a class that **cannot be instantiated** and may contain **abstract methods** (methods without a body) as well as **concrete methods** (methods with a body).

✅ Syntax:

java Copy Edit

```java
abstract class Animal {
    abstract void makeSound();  // abstract method

    void sleep() {              // concrete method
        System.out.println("Sleeping...");
    }
}
```

🔹 2. Abstract Method

An **abstract method** is a method that **has no implementation** (no body) in the abstract class. Subclasses must **override and implement it**.

✅ Rules:

- Must be in an abstract class.
- Cannot be `private`, `static`, or `final`.

✅ Example:

java Copy Edit

```java
abstract class Animal {
    abstract void makeSound();  // abstract method
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}
```

❌ You Cannot:

- Create an object of an abstract class directly:

java Copy Edit

```java
Animal a = new Animal(); // ❌ Compile-time error
```

🧪 Full Example:

java Copy Edit

```java
abstract class Shape {
    abstract void draw();

    void display() {
        System.out.println("This is a shape");
    }
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape s = new Circle();  // Upcasting
        s.draw();                // Calls Circle's draw
        s.display();             // Calls concrete method from Shape
    }
}
```

Inner Classes in Java

An inner class in Java is a class defined inside another class. Java supports several types of inner classes, which are useful for grouping logic, encapsulating behavior, and improving readability and organization.

🔹 Types of Inner Classes

1. **Member Inner Class** (non-static)
2. **Static Nested Class**
3. **Local Inner Class** (inside a method)
4. **Anonymous Inner Class**

✅ 1. Member Inner Class

- Defined **within a class**, **outside** any method.
- Can access all **members (even private)** of the outer class.

java Copy Edit

```java
class Outer {
    int outerVal = 10;

    class Inner {
        void display() {
            System.out.println("Outer value: " + outerVal);
        }
    }
}

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

✅ 2. Static Nested Class

- Declared using the `static` keyword.
- Cannot access **non-static** members of the outer class directly.

java Copy Edit

```java
class Outer {
    static int x = 30;

    static class Inner {
        void show() {
            System.out.println("Static outer value: " + x);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Outer.Inner i = new Outer.Inner();  // no need for Outer instance
        i.show();
    }
}
```

✅ 3. Local Inner Class

- Defined **inside a method**.
- Can access final or effectively final local variables.

java Copy Edit

```java
class Outer {
    void outerMethod() {
        int num = 100;

        class LocalInner {
            void print() {
                System.out.println("Local var: " + num);
            }
        }

        LocalInner li = new LocalInner();
        li.print();
    }
}
```

✅ 4. Anonymous Inner Class

- A class without a name.
- Used to **override methods or implement interfaces** in one-shot.

```java
abstract class Animal {
    abstract void sound();
}

public class Test {
    public static void main(String[] args) {
        Animal a = new Animal() {
            void sound() {
                System.out.println("Anonymous Animal sound");
            }
        };

        a.sound();
    }
}
```

Interface in Java

🔌 What is an Interface in Java?

An **interface** in Java is a **reference type**, similar to a class, but it can only contain:

- **abstract methods** (until Java 7),
- **default** and **static methods** (from Java 8),
- **private methods** (from Java 9),
- and **constants** (implicitly `public static final`).

✅ Syntax Example:

```java
interface Animal {
    void makeSound();  // implicitly public and abstract
}

class Dog implements Animal {
    public void makeSound() {
        System.out.println("Dog barks");
    }
}
```

❓ Why Use Interfaces in Java? (Need for Interface)

Reason	Explanation
✅ **Achieve 100% abstraction**	No method implementation in traditional interfaces (before Java 8).
✅ **Multiple inheritance**	Java **does not support multiple class inheritance**, but you can implement **multiple interfaces**.
✅ **Loose coupling**	Interfaces let you code to a **contract**, not an implementation.
✅ **Standardization**	They define a **common behavior** that different classes can implement (e.g., `Comparable`, `Runnable`).
✅ **Plug-and-play**	Helpful in **dependency injection**, **frameworks**, and **API design**.

🧠 Key Rules of Interfaces

Feature	Interface
Can contain variables	✅ But only `public static final` (constants)
Can contain method implementations	✅ From Java 8 via `default` and `static` methods
Can be implemented by a class	✅ Using `implements`
Can extend another interface	✅ Multiple inheritance of interfaces
Can have constructors	❌ No constructors allowed

🧪 Example with Multiple Interfaces

java Copy Edit

```java
interface Flyable {
    void fly();
}

interface Swimmable {
    void swim();
}

class Duck implements Flyable, Swimmable {
    public void fly() {
        System.out.println("Duck flies");
    }

    public void swim() {
        System.out.println("Duck swims");
    }
}
```

🆚 Abstract Class vs Interface

Feature	Abstract Class	Interface
Inheritance type	Single inheritance	Multiple inheritance
Method implementation	Can have both abstract and concrete methods	Only abstract (until Java 7), default/static from Java 8
Constructor	Yes	No
Field types	Any type	`public static final` only
Usage	"Is-a" relationship	"Can-do" capability

Enum in Java

In Java, an enum (short for enumeration) is a special data type that enables you to define a fixed set of constants — like days of the week, directions, states, etc.

✅ Why Use `enum`?

- To represent a group of **named constants**.
- Type-safe: An enum variable can only take one of the predefined constants.
- Makes code **more readable**, **maintainable**, and **less error-prone** than using raw strings or integers.

🔧 Basic Syntax

java

Copy Edit

```java
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

🧪 Usage Example

java

Copy Edit

```java
public class TestEnum {
    enum Day {
        MONDAY, TUESDAY, WEDNESDAY
    }

    public static void main(String[] args) {
        Day today = Day.MONDAY;

        switch (today) {
            case MONDAY:
                System.out.println("Start of the week");
                break;
            case TUESDAY:
                System.out.println("Second day");
                break;
            default:
                System.out.println("Another day");
        }
    }
}
```

🔍 Features of Java Enum

Feature	Description
Implicitly extends `java.lang.Enum`	So enums can't extend other classes
Can have **fields**, **constructors**, **methods**	Yes, like classes
Can be used in `switch` **statements**	Yes
Can implement **interfaces**	Yes

Annotation in Java

Annotations in Java are metadata **(data about data) that provide** information to the compiler, tools, or frameworks**. They do** not directly affect program logic**, but can influence** how the code is compiled or executed**.**

🔹 Why Use Annotations?

- **Provide instructions** to compilers or tools (e.g., `@Override`)
- **Configure frameworks** like Spring, JPA, Hibernate, etc.
- **Reduce boilerplate code**
- Enable **custom behaviors** through reflection

✅ Built-in Annotations in Java

Annotation	Purpose
`@Override`	Ensures a method is overriding a superclass method
`@Deprecated`	Marks method/class as deprecated (not recommended for use)
`@SuppressWarnings`	Suppresses compiler warnings (e.g., `unchecked`)
`@FunctionalInterface`	Declares that an interface has only one abstract method

Example:

java Copy Edit

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

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

🔍 Meta-Annotations

These are annotations **used on other annotations**:

Meta-Annotation	Purpose
`@Retention`	Specifies if annotation is available at runtime, class file, or source
`@Target`	Specifies where the annotation can be applied (class, method, field, etc.)
`@Documented`	Includes annotation in Javadoc
`@Inherited`	Allows annotation to be inherited by subclasses

Different Type of Interface

🔹 1. Normal Interface

A standard interface with **abstract methods** and (since Java 8) **default/static methods**.

✅ Example:

java Copy Edit

```java
interface Vehicle {
    void start();              // abstract method

    default void stop() {
        System.out.println("Vehicle stopped");
    }
}
```

🔹 2. Functional Interface (Single Abstract Method - SAM)

An interface with **only one abstract method**.
Introduced in Java 8, used mainly with **lambda expressions**.

✅ Example:

java Copy Edit

```java
@FunctionalInterface
interface Printable {
    void print(String msg);
}

// Lambda usage:
Printable p = msg -> System.out.println(msg);
p.print("Hello!");
```

> ⚠️ Can have default/static methods too, but only **one abstract method** is allowed.

🔹 3. Marker Interface

An interface with **no methods or fields** — just used to **mark or tag** a class for a specific purpose.

✅ Example:

java Copy Edit

```java
interface Serializable {}  // no methods

class MyClass implements Serializable {
    // JVM uses this "marker" to allow object serialization
}
```

Lambda Expressions in Java

Lambda expressions were introduced in Java 8 to provide a concise way to represent a method using an expression. They enable functional programming features in Java and are mainly used to implement functional interfaces

🔹 Example 1: Without Lambda

java

Copy Edit

```java
interface Printable {
    void print(String msg);
}

public class Test {
    public static void main(String[] args) {
        Printable p = new Printable() {
            public void print(String msg) {
                System.out.println(msg);
            }
        };
        p.print("Hello without lambda");
    }
}
```

🔹 Example 2: With Lambda

java

Copy Edit

```java
interface Printable {
    void print(String msg);
}

public class Test {
    public static void main(String[] args) {
        Printable p = msg -> System.out.println(msg);
        p.print("Hello with lambda");
    }
}
```

🧠 Rules and Features

Rule / Feature	Description
Used with **functional interfaces**	Interface with **only one abstract method**
Can **omit parameter types**	Compiler infers types
Use `->` as the lambda operator	Separates parameters from body
Useful in **Collections, Streams**	For concise looping, filtering, etc.

Exception in Java

In Java, an exception is an event that disrupts the normal flow of a program. It typically occurs when something unexpected or erroneous happens during program execution.

🔹 Why Exception Handling?

- To avoid program crashes
- To handle errors **gracefully**
- To provide **custom messages** and logic for different failure situations
- To **maintain program flow** even if something goes wrong

🧱 Hierarchy of Exceptions

```php
Object
 └── Throwable
      ├── Error        (serious problems you shouldn't catch)
      └── Exception    (problems you can and should handle)
           ├── Checked Exceptions
           └── Unchecked Exceptions (RuntimeException)
```

✅ Types of Exceptions

1. Checked Exceptions

- Handled at **compile time**
- Must be declared using `throws` or handled with `try-catch`
- Examples: `IOException`, `SQLException`, `FileNotFoundException`

2. Unchecked Exceptions

- Occur at **runtime**
- Result of programming bugs
- Examples: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`

3. Errors

- Represent **serious problems**
- Not handled in normal code
- Examples: `OutOfMemoryError`, `StackOverflowError`

🔧 Exception Handling Syntax

java Copy Edit

```java
try {
    // code that may throw exception
} catch (ExceptionType e) {
    // code to handle exception
} finally {
    // optional, always executes
}
```

🔹 Example: Divide by Zero

java Copy Edit

```java
public class Demo {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Error: Cannot divide by zero.");
        } finally {
            System.out.println("This block always runs.");
        }
    }
}
```

Output:

vbnet Copy Edit

```
Error: Cannot divide by zero.
This block always runs.
```

🛠️ Keywords

Keyword	Use
`try`	Define a block of code to test for errors
`catch`	Handle specific exception
`finally`	Cleanup code (always runs)
`throw`	Throw an exception manually
`throws`	Declare exception for a method

✍️ Custom Exception Example

java Copy Edit

```java
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}

public class Test {
    public static void main(String[] args) throws MyException {
        throw new MyException("Custom error occurred!");
    }
}
```

Ducking Exception in Java

Ducking an exception in Java refers to the practice of passing the responsibility of handling an exception to the caller method instead of handling it within the method itself. This is done using the throws keyword in the method declaration

📘 Why "Ducking"?

The term "ducking" comes from the idea of **"ducking under the problem"** — you're not solving it in the current method, you're saying:

➡️ *"I can't handle it here — the caller must handle it."*

✅ Syntax of Ducking an Exception

java Copy Edit

```java
void myMethod() throws IOException {
    // risky code that may throw IOException
}
```

```java
import java.io.*;

public class Demo {
    public static void main(String[] args) throws IOException {
        readFile();  // we're ducking the exception to main()
    }

    static void readFile() throws IOException {
        BufferedReader br = new BufferedReader(new FileReader("data.txt"));
        System.out.println(br.readLine());
        br.close();
    }
}
```

In the above example:

- `readFile()` may throw `IOException`, but it doesn't catch it.
- It **ducks** the exception using `throws IOException`.
- Now the calling method (`main`) either needs to catch or also duck it.

❗ When to Duck

✅ Duck an exception when:

- You can't handle it at the current level
- You're writing a **low-level method** and want to give control to higher-level code

❌ Don't duck **RuntimeExceptions** unnecessarily (unchecked ones)

Difference between throw and throws in Java

🔹 `throw` — **Used to actually throw an exception**

- Used **inside** a method or block
- Throws **a specific exception object**
- Only **one exception** can be thrown at a time

✅ **Syntax:**

```java
throw new ExceptionType("message");
```

✅ **Example:**

```java
public class Demo {
    public static void main(String[] args) {
        throw new ArithmeticException("Division by zero not allowed");
    }
}
```

🔹 `throws` — Used to **declare** an exception

- Used in a **method declaration**
- Tells the **caller** that this method may throw exceptions
- Can declare **multiple exceptions**, separated by commas

✅ **Syntax:**

```java
void methodName() throws IOException, SQLException {
    // method code
}
```

✅ **Example:**

```java
public void readFile() throws IOException {
    FileReader fr = new FileReader("file.txt");
}
```

🧱 Key Differences Table

Feature	`throw`	`throws`
Purpose	Actually **throws** an exception	**Declares** an exception may be thrown
Location	Inside a method	With method signature
Used for	Throwing **one** exception	Declaring **one or more** exceptions
Object required	Yes, must throw an **instance**	No object needed
Example	`throw new IOException("file missing");`	`void method() throws IOException { }`

BufferedReader vs Scanner in Java

Both BufferedReader and Scanner are used to read input in Java, but they serve different use-cases and have different performance and features.

🔹 BufferedReader

- Comes from: `java.io`
- Reads **text** from a character-based input stream efficiently
- **Faster** than `Scanner` for large input
- Reads **entire lines** or characters
- Does **not parse input** (e.g., integers, words)

✅ Example:

java Copy Edit

```java
import java.io.*;

public class BRDemo {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter your name: ");
        String name = br.readLine();  // reads one line of input
        System.out.println("Hello, " + name);
    }
}
```

🔹 Scanner

- Comes from: `java.util`
- More **flexible** — reads and parses different data types (int, double, String, etc.)
- Slower than BufferedReader for large input
- Uses **regex-based tokenizing**
- Best suited for **console-based programs**

✅ Example:

java Copy Edit

```java
import java.util.*;

public class ScannerDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your age: ");
        int age = sc.nextInt();  // reads an int
        System.out.println("You are " + age + " years old.");
        sc.close();
    }
}
```

try-with-resources in Java

try-with-resources is a feature in Java that allows you to automatically close resources like files, sockets, or database connections after use — without writing explicit finally blocks.

🔧 What is a Resource?

A **resource** is an object that must be **closed after the program is finished using it**, like:

- `BufferedReader`
- `FileReader`
- `Scanner`
- `Connection` (JDBC)
- `FileInputStream`

To be used in try-with-resources, the object must implement the `AutoCloseable` interface (or `Closeable`, which extends `AutoCloseable`).

🔹 Example 1: Reading a File with BufferedReader

java Copy Edit

```java
import java.io.*;

public class TryWithResourceExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
            String line = br.readLine();
            System.out.println("First line: " + line);
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }
        // br is auto-closed here
    }
}
```

✅ No need to call `br.close()`. It is done **automatically**, even if an exception is thrown.

Threads in Java

A thread in Java is a lightweight sub-process — the smallest unit of execution. Using threads, Java supports multithreading, which means multiple parts of a program can run concurrently.

🧠 Why Use Threads?

- To perform **multiple tasks at the same time**
- To keep the application **responsive** (e.g., UI + background task)
- To make better use of **multi-core processors**

🔹 Ways to Create a Thread in Java

1. Extending `Thread` class

java Copy Edit

```java
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}

public class Demo {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();  // start() calls run() internally
    }
}
```

2. Implementing `Runnable` interface

java Copy Edit

```java
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread is running...");
    }
}

public class Demo {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start();
    }
}
```

3. **Using Lambda (Java 8+)**

java Copy Edit

```java
public class Demo {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            System.out.println("Thread using lambda");
        });
        t1.start();
    }
}
```

🔁 Thread Life Cycle

1. **New** – Thread is created
2. **Runnable** – Ready to run
3. **Running** – Actively executing
4. **Blocked/Waiting** – Waiting for a resource or signal
5. **Terminated** – Finished execution

🔧 Common Thread Methods

Method	Description
`start()`	Starts thread, calls `run()` method
`run()`	Code to be executed by thread
`sleep(ms)`	Puts thread to sleep for given milliseconds
`join()`	Waits for thread to die
`isAlive()`	Checks if thread is still running
`yield()`	Hints that other threads can be scheduled
`interrupt()`	Interrupts a sleeping/waiting thread

java

Copy Edit

```java
class A extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++)
            System.out.println("A: " + i);
    }
}

class B extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++)
            System.out.println("B: " + i);
    }
}

public class Demo {
    public static void main(String[] args) {
        new A().start();
        new B().start();
    }
}
```

Race Condition in Java

A race condition is a bug that occurs in multithreaded programs when two or more threads access shared data at the same time, and the final result depends on the timing of their execution.

Because threads run concurrently, the order of execution is not guaranteed, leading to unpredictable results.

🔥 Real-Life Analogy

Imagine two people trying to **update the same bank account** balance at the same time:

- Person A reads balance = ₹1000
- Person B reads balance = ₹1000
- Person A deposits ₹500 → writes ₹1500
- Person B deposits ₹300 → writes ₹1300 (overwriting Person A's update!)

❌ Final balance should be ₹1800
❌ But it's ₹1300 due to **race condition**

```java
class Counter {
    int count = 0;

    public void increment() {
        count++;  // NOT thread-safe
    }
}

public class RaceDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) c.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) c.increment();
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final count: " + c.count);  // 🔥 unpredictable result (often < 2000)
    }
}
```

🧠 Why It Happens

- `count++` is **not atomic** — it actually does:
  1. Read `count`
  2. Increment it
  3. Write back

If two threads do this simultaneously, they might **read the same value** and overwrite each other's updates.

✅ Solution: Synchronization

1. Using `synchronized` keyword:

```java
public synchronized void increment() {
    count++;
}
```

2. Using `AtomicInteger`:

```java
AtomicInteger count = new AtomicInteger();

public void increment() {
    count.incrementAndGet();  // atomic operation
}
```

Collection API in Java

The Java Collection Framework (JCF) is a set of classes and interfaces in the java.util package that helps you store, manipulate, and retrieve groups of objects efficiently.

🌳 Hierarchy Overview

```typescript
          Collection (interface)
             /       |       \
        List        Set     Queue
        ↓           ↓         ↓
   ArrayList     HashSet    PriorityQueue
   LinkedList    TreeSet    LinkedList
   Vector        LinkedHashSet

         Map (separate interface)
            ↓
       HashMap
       TreeMap
       LinkedHashMap
```

🔑 Core Interfaces

Interface	Description
`Collection`	Root of the collection hierarchy
`List`	Ordered collection (duplicates allowed)
`Set`	Unordered, no duplicates
`Queue`	For holding elements before processing
`Map`	Key-value pairs (keys are unique)

📦 Common Collection Classes

🔹 **List** (ordered, duplicates allowed)

- Maintains insertion order
- Access by index

Class	Description
`ArrayList`	Fast for random access
`LinkedList`	Fast for insert/delete
`Vector`	Thread-safe (legacy)

java Copy Edit

```java
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
```

🔹 **Set** (no duplicates)

Class	Description
`HashSet`	Unordered, fastest
`LinkedHashSet`	Maintains insertion order
`TreeSet`	Sorted set (uses Red-Black Tree)

java Copy Edit

```java
Set<String> set = new HashSet<>();
set.add("Java");
set.add("Python");
```

⚙️ Utility Class: `Collections`

- Class with **static methods** to operate on collections:

```java
Collections.sort(list);
Collections.reverse(list);
Collections.max(list);
```

🔹 Queue / Deque

Class	Description
`PriorityQueue`	Elements processed by priority
`ArrayDeque`	Double-ended queue (stack/queue)

```java
Queue<Integer> queue = new LinkedList<>();
queue.add(10);
queue.add(20);
```

🔹 Map (key-value pairs, not part of `Collection`)

Class	Description
`HashMap`	Fast, unordered
`LinkedHashMap`	Maintains insertion order
`TreeMap`	Sorted by keys

```java
Map<String, Integer> map = new HashMap<>();
map.put("John", 90);
map.put("Alice", 80);
```

⚙️ Utility Class: `Collections`

- Class with **static methods** to operate on collections:

```java
Collections.sort(list);
Collections.reverse(list);
Collections.max(list);
```

Stream In Java

Stream was introduced in Java 8, the Stream API is used to process collections of objects. A stream in Java is a sequence of objects that supports various methods that can be pipelined to produce the desired result.

🔗 Stream Pipeline = 3 Parts:

1. **Source** → e.g., `List`, `Set`, `Array`
2. **Intermediate Operations** → e.g., `filter()`, `map()`, `sorted()`
3. **Terminal Operation** → e.g., `collect()`, `forEach()`, `count()`

🧪 Example:

```java
List<String> names = Arrays.asList("John", "Jane", "Mike", "Anna");

names.stream()                          // Source
     .filter(name -> name.startsWith("J"))  // Intermediate
     .sorted()                          // Intermediate
     .forEach(System.out::println);     // Terminal
```

📄 **Output:**

```nginx
Jane
John
```

🔧 Common Methods in Stream API

Type	Method	Description
Intermediate	`filter()`	Filter elements based on condition
	`map()`	Transform each element
	`sorted()`	Sort elements
	`distinct()`	Remove duplicates
Terminal	`collect()`	Convert to List, Set, etc.
	`forEach()`	Perform action for each element
	`count()`	Count elements
	`anyMatch()`	True if any match

🔁 Example: Square Even Numbers from List

java Copy Edit

```java
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6);

List<Integer> result = nums.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .collect(Collectors.toList());

System.out.println(result); // [4, 16, 36]
```

What is Optional in Java?

Optional<T> is a container object introduced in Java 8 that may or may not contain a non-null value. It's a safer alternative to returning null and helps avoid NullPointerException.

🔥 Why Use `Optional`?

Traditionally, Java methods that might return no value return `null`:

```java
String getName() {
    return null; // risky!
}
```

Later usage like `name.length()` throws a `NullPointerException`.

✅ Instead, use `Optional`:

```java
Optional<String> getName() {
    return Optional.ofNullable(null);
}
```

Now the caller is **forced to handle** the absence of a value.

🔧 Creating `Optional` Objects

```java
Optional<String> opt1 = Optional.of("Java");         // value must NOT be null
Optional<String> opt2 = Optional.ofNullable(null);   // value may be null
Optional<String> opt3 = Optional.empty();            // empty optional
```

✅ Common Methods in `Optional`

Method	Description
`isPresent()`	Returns true if value is present
`get()`	Returns the value (if present) — ⚠️ risky
`orElse(value)`	Returns value if present; else returns default
`orElseGet(Supplier)`	Lazy default value
`orElseThrow()`	Throws exception if no value
`ifPresent(Consumer)`	Executes block if value is present
`map()`	Transforms the value inside Optional
`filter()`	Filters the value if condition matches

🧪 Example: Using Optional

java

```java
Optional<String> name = Optional.of("Tison");

// Safe check
if (name.isPresent()) {
    System.out.println(name.get());
}

// One-liner
name.ifPresent(n -> System.out.println(n));

// Default value
String result = name.orElse("Default");
System.out.println(result);  // Tison
```

⚠️ Bad Practice — Don't Do This

java

```java
Optional<String> name = Optional.of("Giya");
if (name != null) {          // ❌ NO! Optional is never null
    System.out.println(name.get());
}
```

Method Reference & Constructor Reference in Java (Java 8+)

Method reference and constructor reference are shortcuts in lambda expressions to directly refer to existing methods or constructors.

They make the code cleaner and more readable.

🔹 1. **Method Reference**

A **method reference** is used to refer to a **method without executing it**.

✅ Syntax:

```cpp
ClassName::methodName
```

📘 Types of Method References:

Type	Syntax	Example	Used When...
Static method	`ClassName::staticMethod`	`Math::max`	You are calling a static method
Instance method of a particular object	`instance::method`	`System.out::println`	You have an instance and want to call its method
Instance method of an arbitrary object of a particular type	`ClassName::instanceMethod`	`String::toLowerCase`	Used with streams or mapping
Constructor reference	`ClassName::new`	`ArrayList::new`	You want to create objects in a lambda

📌 Example 1: **Static Method Reference**

```java
class Utility {
    public static void print(String s) {
        System.out.println(s);
    }
}

// Lambda way
Consumer<String> lambda = s -> Utility.print(s);

// Method reference
Consumer<String> methodRef = Utility::print;
```

📌 Example 2: **Instance Method Reference**

```java
List<String> names = Arrays.asList("John", "Anu", "Tison");

// Using Lambda
names.forEach(name -> System.out.println(name));

// Using method reference
names.forEach(System.out::println);
```

📌 Example 3: **Reference to an Instance Method of a Class**

java Copy Edit

```java
Function<String, String> lambda = s -> s.toUpperCase();
Function<String, String> methodRef = String::toUpperCase;
```

🔹 2. **Constructor Reference**

Used when you want to create a new object using a **lambda expression**.

✅ Syntax:

cpp Copy Edit

```cpp
ClassName::new
```

📌 Example:

java Copy Edit

```java
class Employee {
    String name;
    Employee(String name) {
        this.name = name;
    }
}

// Using lambda
Function<String, Employee> lambda = name -> new Employee(name);

// Using constructor reference
Function<String, Employee> constructorRef = Employee::new;
```

🧠 When to Use Method or Constructor Reference?

Use it when:

- You want to replace a **lambda** that just calls a method or constructor.
- It makes the code **shorter and more expressive**.