2-First Code In Java

First Java Program in VS Code

Choosing a Theme

On the first launch of VS Code, you will be asked to choose a theme according to your preference. We will choose the dark theme. After opening the editor, you will see several icons on the left panel of the screen:

1. **Explorer Icon**: Used for locating files and folders.
2. **Search Icon**: Used to search for the desired file or folder.
3. **Source Control Icon**: Used for sharing or pushing our code to a repository or any version control system.
4. **Run & Debug Icon**: Used to run our program and debug it later if any error occurs.
5. **Extensions Icon**: Used to add suitable and required extensions.

Creating a Project

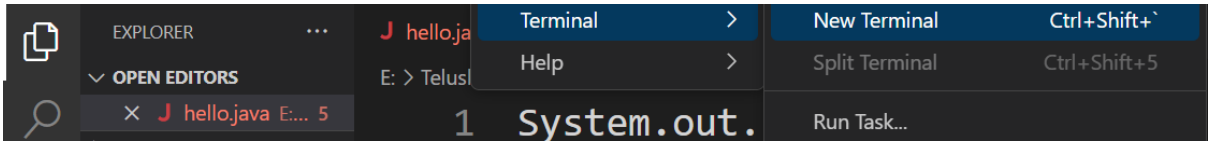
1. Click on **New Project**.
    - A project is a group of multiple files, including multiple external libraries and dependencies.
    - As this is our first project, we are not using any external library.
    - Choose a location where you will store your project files.
2. **Checking Java Version**:
    - Open the terminal in VS Code by clicking on the + icon and selecting **Open Terminal**.
    - Type the following commands to check the Java and Javac versions:

      ```
      java -version
      javac -version
      ```

    - Ensure you have Java 17, which is an LTS version.

Creating Your First Java File

1. **Create a New File**:
    - Name the file Hello.java.
    - As we use .js extension for JavaScript and .c for C files, we use .java extension for Java files.
2. **Writing the Code**:
    - To print "Hello, World!" on the screen as output, we need to write at least 3-4 lines of code because Java is a structured language.
    - However, from Java 9, we have JShell, which was introduced for beginners to understand the syntax and for experimental purposes. Using JShell, we can print this in a single line.
3. **Using JShell**:
    - Open JShell by typing jshell in the command prompt.
    - In VS Code, click on the + icon and select **Open Terminal** to open the terminal.



-
- Type the following command in JShell:

  System.out.println("Hello, World!");

- Whenever we write text inside the braces, we need to put it in double quotes. For numbers, we just write them as they are.
- Press Enter, and you will see the console displaying "Hello, World!".

```
jshell> System.out.print("Hello World");
Hello World
```

Running the Code in a Java File

If we try to compile and run the single line System.out.println("Hello, World!"); in a Java file, it will give errors.

```
PS E:\Telusko> javac .\hello.java
.\hello.java:1: error: class, interface, enum, or record expected
System.out.print("hello world");
^
1 error
```

```
PS E:\Telusko> java .\hello
Error: Could not find or load main class .\\hello
Caused by: java.lang.ClassNotFoundException: /\\hello
```

We will explore the errors and their meanings in the next chapter.

03-How Java Works

When we write a Java program, we typically use tools like JShell to test our code. However, running a complete program requires understanding the role of the Java Virtual Machine (JVM) and the Java Runtime Environment (JRE). Let's delve into how JVM operates and the control flow of a Java program.

The Role of JVM

The JVM is an integral part of the Java ecosystem, allowing Java code to be executed on any platform. Each operating system (OS) has a specific JVM designed for it, making Java platform-independent at the source code level but platform-dependent at the binary level.

Diagram

Layers

Control Flow of Java Code

Java code undergoes several stages from writing to execution. Here's a step-by-step breakdown of the process:

1. **Writing Code**: Java code is written in simple, understandable English terms.
2. **Compilation**: The Java compiler converts the source code (.java files) into bytecode (.class files).
3. **Execution by JVM**: The JVM executes the bytecode. It specifically looks for the main method as the entry point for any Java program.

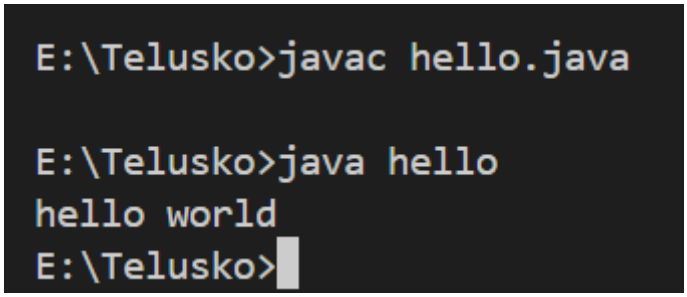
Main Method:

- The main method has a specific signature: public static void main (String[] args) {}.
- This method is crucial as it marks the starting point of program execution.

Steps to Write and Execute Java Code

1. **Write the Code**: Ensure all executable code is within the main method.
2. **Follow OOP Principles**: Java is object-oriented; thus, everything is treated as an object, and classes act as blueprints.
3. **Compile the Code**: Use the Java compiler to compile your code.
    - This generates a .class file containing bytecode.
4. **Run the Code**: Use the java command followed by the class name (without the .class extension) to run your program.

```
public class Hello{
  public static void main(String[]args) {
      System.out.print("helloworld");
      }
   }
```



Understanding JRE

The JRE is essential for running Java programs. It includes the JVM and a set of libraries and other files that the JVM uses at runtime. Think of JRE as a kitchen:

- **Kitchen Analogy:**
    - The kitchen provides the environment to cook a dish.
    - Utensils and Ingredients are like the libraries and resources JRE provides.

Example:

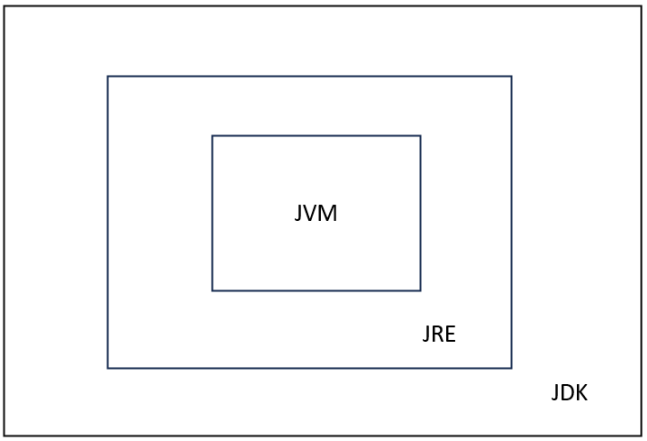
- Suppose you want to bake a cake. The JRE is like the entire kitchen setup needed for baking, including the oven, mixing bowls, and ingredients. The JVM is the oven that actually bakes the cake using the ingredients (bytecode) provided.

Visualizing Java Architecture

Imagine the Java development environment as nested boxes:

- **Outer Box (JDK)**: The Java Development Kit includes everything for development.
- **Middle Box (JRE)**: The JRE provides the runtime environment.

- **Innermost Box (JVM)**: The JVM interprets and executes the bytecode.



1.Variables

Recap:

In the previous chapter, we learned about the control flow of code execution, focusing on the concepts of JDK, JRE, and JVM and their importance. We used JDK 17, which is a Long-Term Support (LTS) version. It's important to note that a new version of the JDK is released every six months, each bringing new features and improvements.

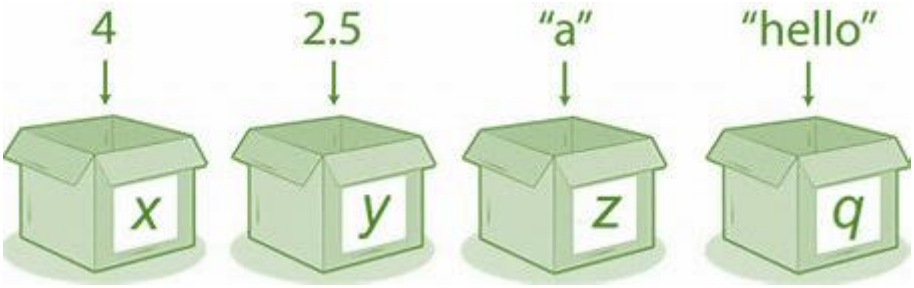
Variables:

Why do we build applications? Why do we code? These questions often arise, and the answer is simple: to solve real-world problems. For example, Amazon provides a convenient way to shop without visiting physical stores, and food delivery apps let us order meals without going to restaurants.

All these applications rely on large amounts of data to function effectively. This data is processed and stored in a persistent storage system called a database, where it can be retrieved and updated as needed. During data processing, we need to store data temporarily; this is where variables come in. Variables act as containers that temporarily hold data while we perform operations on it.

Data can come in various forms, such as text, whole numbers, integers, or decimals. To manage these different types of data, we use specific data types for our variables. For instance:

- An integer like 8 is represented by the int data type.
- Text data is represented using the String data type.



According to the image, as we can see, variables x, y, z, and q are the boxes that hold the data inside it.

Code Example:

Let's consider a simple example where we use two numbers and perform addition:

- Let num1 be the first variable with a value of 8, and num2 the second variable with a value of 5. Both are of the int data type.

- To store the result of their addition, we use another variable called result, which is also of int datatype.

```java
class hello{

    public static void main(String[] args) {
        int num1=8;
        int num2=5;
        int result=num1+num2;
        System.out.println(result);
        }
}
```

```
E:\Telusko>javac hello.java

E:\Telusko>java hello
13
```

Each time we modify our code, we must compile it before running the program. This step is crucial because it converts our code into bytecode, which the Java Virtual Machine (JVM) executes. Changes in the code affect the bytecode, which can alter the program's output.

Although we could directly use numbers in our operations, using variables allows us to store and manipulate data dynamically, making our code more flexible and maintainable.

Creating variables in Java

When we create variables in Java, there are a few key steps to follow:

1. **Specify the data type**:
   - The first step in creating a variable is to define its data type. The data type determines what kind of data the variable can hold (e.g., `int` for integers, `double` for decimals, `char` for characters).
2. **Give the variable a name**:
   - After specifying the data type, assign a suitable name to the variable. The name should be meaningful and follow Java's naming conventions (e.g., `camelCase` for variable names).
3. **Assign a value**:

You can assign a value to the variable at the time of creation by using the assignment operator `=`. This step is known as initialization. For example:

```java
int number = 10;
```

-

Alternatively, you can declare the variable without assigning a value. In this case, the variable will hold a default value, depending on its data type. For example:

```
int number;
```

-
    - Here, `number` will be initialized with a default value of `0`.

4. **Default Values**:
    - If you choose not to assign a value during variable declaration, Java automatically assigns a default value based on the data type:
        - `int`: 0
        - `double`: 0.0
        - `char`: '\u0000' (null character)
        - `boolean`: false
        - Objects (e.g., Strings): `null`

Steps to Remember While Coding:

1. **Use semicolons correctly:** Ensure each statement ends with a semicolon to avoid syntax errors.
2. **Correct Use of Curly Braces ({}):** Curly braces define code blocks. Missing braces can prevent your code from compiling successfully.
3. **Proper Indentation:** Indentation makes your code more readable and easier to understand for others. It also helps in visualizing the structure and flow of the code.

02-DataTypes

In Java, data is stored in variables, which can have different data types.

Data Types

Data types in Java are divided into two main categories:

1. **Primitive Data Types**
2. **Non-Primitive Data Types** (to be discussed in upcoming chapters)

Primitive data types: Primitive data types in Java are the foundational building blocks of data manipulation. Unlike more complex data structures, they represent simple values such as a single number or character. Java offers eight primitive types: byte, short, int, long, float, double, char, and boolean and it can be further classified into the following sub-categories:

1. **Integer types**
2. **Floating-point types**
3. **Character type**
4. **Boolean type**

Detailed Breakdown of Primitive Data Types

1. Integer

The integer data types are used for storing whole numbers, which include natural numbers, zeros, and negative numbers. Different integer data types are available based on the range and size required:

- **byte**: 1 byte (8 bits), range: -128 to 127
- **short**: 2 bytes (16 bits), range: -32,768 to 32,767
- **int**: 4 bytes (32 bits), range: -2^31 to 2^31 - 1
- **long**: 8 bytes (64 bits), range: -2^63 to 2^63 - 1 (use an 'L' suffix to specify a long literal, e.g., 123456789L)

2. Character

The `char` type is used to store a single character, such as a letter, digit, or symbol, enclosed in single quotes (e.g., `'A'`). It has a size of 2 bytes (16 bits)

because Java uses the Unicode standard, which accommodates a broader range of characters compared to ASCII.

3. Boolean

The `boolean` type represents a truth value and can hold only two possible values: `true` or `false`. It is primarily used in conditional statements. Unlike some other programming languages that might represent boolean values with 0 and 1, Java strictly uses the keywords `true` and `false`. This means that in Java, you cannot use integers to represent boolean values—only `true` or `false` are valid.

In Java, the `boolean` data type is unique compared to languages that might allow numerical representations (like 0 for false and 1 for true). Java does not support 0 and 1 as boolean values. The boolean data type strictly allows only two values:

- `true`
- `false`

This design decision helps prevent logical errors and ensures that conditions in your code are clear and unambiguous.

4. Float and Double

These data types are used for storing decimal or floating-point numbers:

- **float**: 4 bytes, single-precision, range approximately ±3.40282347E+38F (use 'f' suffix, e.g., 3.14f)
- **double**: 8 bytes, double-precision, range approximately ±1.79769313486231570E+308 (default for decimal numbers)

Sizes and Ranges of Primitive Data Types

Data Type	Size	Range	Example
byte	1 byte	-128 to 127	byte b = 100;
short	2 bytes	-32,768 to 32,767	short s = 3000;
int	4 bytes	-2^31 to 2^31 - 1	int i = 123489;
long	8 bytes	-2^63 to 2^63 - 1	long l = 123456789L;

Data Type	Size	Range	Example
float	4 bytes	±3.40282347E+38F	float f = 3.14f;
double	8 bytes	±1.79769313486231570E+308	double d = 3.14159;
char	2 bytes	0 to 65,535 (Unicode)	char c = 'A';
boolean	1 bit	true or false	boolean flag = true;

- **Code:**

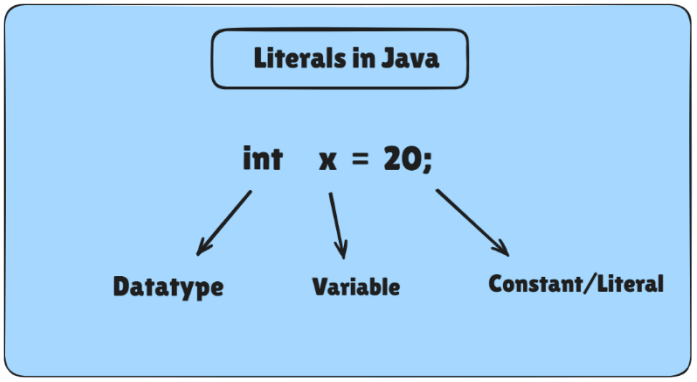
```
class hello{

    public static void main(String[] args) {
        int num1=8;

        byte by=127;

        short sh=558;

        long l= 5854l;

        float f=5.8f;

        double d=5.8;

        char c='k';

        boolean b=  true;

        }
}
```

- This Java program demonstrates the declaration and initialization of various primitive data types, including `int`, `byte`, `short`, `long`, `float`, `double`, `char`, and `boolean`. Each variable is assigned a specific value to illustrate how these data types can be used in Java.

01-Literals

Literals in Java

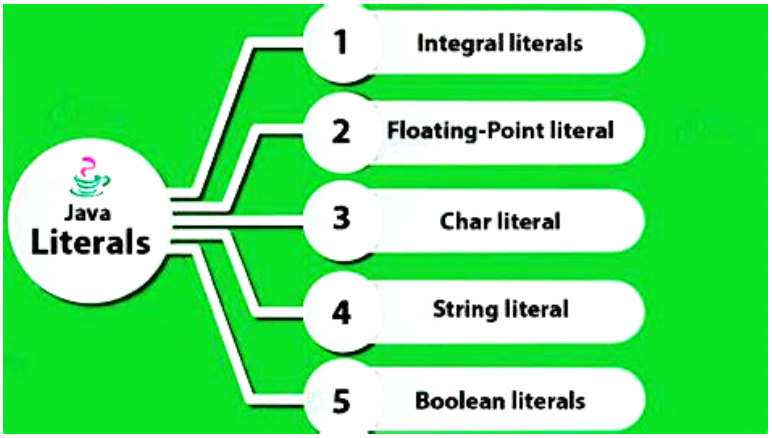
Literals in Java represent fixed values directly in the source code. They are constants that can be assigned to variables and come in various types.



Types of Literals in Java

Java supports several types of literals, including:

1. Integer Literals
2. Character Literals
3. Boolean Literals
4. String Literals
5. Floating Point Literals



Detailed Explanation of Each Literal Type

1. Integer Literals

Integer literals represent whole numbers and can be expressed in four different number systems:

- Decimal (Base 10): Uses digits 0-9.

int decimal = 101; // Decimal literal

- Octal (Base 8): Uses digits 0-7, prefixed with 0.

int octal = 0146; // Octal literal (equals 102 in decimal)

- Hexadecimal (Base 16): Uses digits 0-9 and letters a-f (or A-F), prefixed with 0x or 0X.

int hex = 0x123Face; // Hexadecimal literal

- Binary (Base 2): Uses digits 0 and 1, prefixed with 0b or 0B.

int binary = 0b1111; // Binary literal (equals 15 in decimal)

2. Character Literals

Character literals are single characters enclosed in single quotes. They can also represent special characters using escape sequences.

char letter = 'a';

char symbol = '%';

char unicodeChar = '\u000d'; // Unicode representation

3. Boolean Literals

Boolean literals represent truth values and can only be true or false.

boolean isJavaFun = true;

boolean isFishMammal = false;

4. String Literals

String literals are sequences of characters enclosed in double quotes.

String name = "Jack";

String number = "12345";

String newLine = "\n"; // Newline character

5. Floating Point Literals

Floating point literals represent numbers with fractional parts and can be of type float or double.

- Float Literals: Ends with F or f.

float price = 19.99f;

- Double Literals: Ends with D or d (optional).

double weight = 65.7;

double scientific = 1.234e2; // Exponent notation

Invalid Literals and Restrictions

Using underscores in numeric literals can enhance readability, but there are restrictions:

- Cannot start or end a number with an underscore.
- Cannot place an underscore adjacent to a decimal point in a floating-point literal.
- Cannot place an underscore adjacent to F or L suffixes.

Invalid Examples:

int invalid = 77_; // Invalid: underscore at the end

float invalidFloat = 6_.674F; // Invalid: underscore before decimal

Why Use Literals?

Literals are used to directly assign values to variables without needing to define constants separately. They simplify code by embedding constant values within the instructions.

FAQs on Literals

1. What are literals in Java?
    - Literals are fixed values assigned directly to variables in the source code.
2. Can literals be changed during program execution?

- No, literals are constants and cannot be changed once defined.

3. What is a real literal?
    - Real literals represent floating-point numbers, like 12.34 or 1.23e3.
4. What is a null literal?
    - A null literal represents the absence of an object reference, commonly assigned as null.

03-Type conversion & casting

Type Conversion (Implicit Conversion)

Type conversion, also known as implicit conversion or widening conversion, occurs automatically when:

- A smaller data type is assigned to a larger data type.
- The conversion is between compatible data types.

Characteristics:

- Performed automatically by the compiler
- No data loss occurs.
- The destination type is always larger than the source type.

Example:

When storing data, we use variables of various data types, such as boolean, int, float, byte, or short. But can we change the type of a variable? Let's explore this with an example:

byte b = 127;

int a = 256;

b = a; // This line will cause a compilation error.

In the above example, we're trying to assign an int value to a byte variable. Since int can hold a larger range of values than byte, the compiler will generate an error because this assignment may lead to data loss.

However, if we reverse the assignment:

int a = b;

This is allowed because we're storing a byte value in an int variable, which has a larger range. This type of conversion, where a smaller type is converted to a larger type, is called **implicit type conversion** or **widening**. The compiler handles these conversions automatically.

Type Casting (Explicit Conversion)

Type casting, also known as explicit conversion or narrowing conversion, is performed manually by the programmer when:

- A larger data type needs to be assigned to a smaller data type.
- The conversion may result in data loss.

Characteristics:

- requires explicit use of the casting operator
- Can be applied to both compatible and incompatible data types.
- May result in data loss or unexpected results

Code Example:

```
public class TypeCastingExample {
    public static void main(String[] args) {
        byte b = 127;  // Step 1
        int a = 257;   // Step 2
        b = (byte) a;  // Step 3

        System.out.println("Value of b after casting: " + b);  // Step 4
    }
}
```

Step 1: Declare and initialize a byte variable
byte b = 127;

b is initialized with the maximum value a byte can hold, which is 127.

Step 2: Declare and initialize an int variable
int a = 257;

a is an int variable initialized with 257.

Step 3: Cast int to byte using modulo operation
b = (byte) a;

Here, you assign a to b by casting.

The byte data type has a range from -128 to 127 (which is a total of 256 different values).

Now, when you cast a to a byte, Java calculates the equivalent value within the byte range using the modulo operation:

This explicit conversion is known as **casting** or **narrowing**, where we manually convert a larger data type to a smaller one.

Modulo Calculation:

257 % 256 = 1

After complete code execution, our output window displays:

Output:

```
Value of b after casting: 1
```

Automatic promotion in **Java** is a feature that automatically converts smaller data types to larger data types when they are used in expressions or method calls. This conversion ensures that all operands in an expression or arguments in a method call are of compatible types, allowing the operation to be performed successfully.

Type Promotion:

Type promotion refers to the process where a smaller data type is automatically converted to a larger data type. For instance, an `int` can be promoted to a `long`, `float`, `double`, and so on. The JVM performs this automatic type promotion when a method that requires a larger data type is called with a smaller data type argument.

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

        byte a=10;
        byte b=30;

        int result=a*b;

        System.out.println(result);
    }
}
```

```
E:\Telusko>javac hello.java

E:\Telusko>java hello
300
```

Code explanation:

The program calculates the product of a and b using a * b. Although both a and b are byte types, the result is automatically promoted to an int type because in Java, arithmetic operations involving bytes result in an integer value. This is why the result is stored in an int variable named result.

04-Arithmetic Operators

Operators in Java

Operators are symbols that perform specific operations on one or more operands. In Java, we commonly use operators for basic arithmetic operations like addition (+), subtraction (-), multiplication (*), and division (/), as well as for other functions like modulus (%).

Example: Basic Arithmetic Operations

```
int num1 = 7;

int num2 = 5;


// Addition

int res = num1 + num2;

System.out.println(res); // Output: 12
```

- **Addition (+)**: Adds two operands.
- **Subtraction (-)**: subtracts the second operand from the first.
- **Multiplication (*)**: Multiplies two operands.
- **Division (/)**: Divides the first operand by the second, giving the quotient.
- **Modulus (%)** : Returns the remainder after division.

Example: modulus operation

```
int num1 = 26;

int num2 = 5;

int res = num1 % num2;

System.out.println(res); // Output: 1
```

In this example, 26 % 5 equals 1 because 26 divided by 5 is 5 with a remainder of 1.

Increment and Decrement Operators

The increment (++) and decrement (--) operators always change the value by 1. These can be applied in two forms: pre and post.

- **Pre-increment (++x)/Pre-decrement (--x)**: Increases or decreases the value before the operation is performed.
- **Post-increment (x++) / post-decrement (x--)**: Increases or decreases the value after the operation is performed.

Example: Increment Operations

int num1 = 7;

num1 += 2; // Equivalent to num1 = num1 + 2;

System.out.println(num1); // Output: 9

int num2 = 16;

num2 -= 4; // Equivalent to num2 = num2 - 4;

System.out.println(num2); // Output: 12

Example: Pre and Post Increment/Decrement

Code:

```
public class IncrementDecrementExample {
    public static void main(String[] args) {
        int x = 5;
        int y = ++x;  // Pre-increment: x is incremented first, then assigned to y
        System.out.println("After pre-increment: x = " + x + ", y = " + y);

        int z = x--;  // Post-decrement: x is assigned to z, then decremented
        System.out.println("After post-decrement: x = " + x + ", z = " + z);

        int w = --x;  // Pre-decrement: x is decremented first, then assigned to w
        System.out.println("After pre-decrement: x = " + x + ", w = " + w);
    }
}
```

Output:

```
After pre-increment: x = 6, y = 6
After post-decrement: x = 5, z = 6
After pre-decrement: x = 4, w = 4
```

Explanation:

1. **Pre-increment (`++x`):** `x` is incremented first, so `x` becomes `6`, and then `y` is assigned the value of `x`, making `y = 6`.
2. **Post-decrement (`x--`)**: The current value of `x` (which is `6`) is assigned to `z`, and then `x` is decremented, making `x = 5`.
3. **Pre-decrement (`--x`):** `x` is decremented first, so `x` becomes `4`, and then `w` is assigned the value of `x`, making `w = 4`.

Key Differences:

- **Pre-increment/decrement**: Modifies the variable's value before using it in an expression.
- **Post-increment/decrement**: Uses the variable's original value in the expression before modifying it.

05-Relational Operators

Java Relational Operators are binary operators that are used to evaluate relationships between two operands, such as equality, greater than, less than, and so on. These comparisons return boolean results, and they are commonly used in looping statements, conditional if-else statements, and similar constructs. The standard format for representing a relational operator is:

Syntax:

variable1 relational_operator variable2

implementation:

Equality (==): Checks if two values are equal.

Inequality (!=): Checks if two values are not equal.

Greater than (>): checks if the left operand is greater than the right operand.

Less than (<): Checks if the left operand is less than the right operand.

Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right operand.

Less than or equal to (<=): Checks if the left operand is less than or equal to the right operand.

Examples of Each Operator:

```
int a = 10;

int b = 20;

// Equality (==)

System.out.println(a == b); // Output: false
```

```
// Inequality (!=)
System.out.println(a != b); // Output: true

// Greater than (>)
System.out.println(a > b); // Output: false

// Less than (<)
System.out.println(a < b); // Output: true

// Greater than or equal to (>=)
System.out.println(a >= b); // Output: false

// Less than or equal to (<=)
System.out.println(a <= b); // Output: true
```

06-Logical Operators

Logical operators in Java are used to perform logical operations on boolean expressions. These operators are crucial for decision-making in programs. Here's a breakdown of the main logical operators:

1. Logical AND (& and &&)

- **& (Bitwise AND)**: Evaluates both operands.
- **&& (Short-Circuit AND)**: Evaluates the second operand only if the first is true.

Example:

```
int x = 7, y = 5, a = 5, b = 9;

boolean result = (x > y) && (a < b);

System.out.println(result); // Output: true
```

Explanation: Both conditions (x > y) and (a < b) are true, so the result is true.

2. Logical OR (| and ||)

- **| (Bitwise OR)**: Evaluates both operands.
- **|| (Short-Circuit OR)**: Evaluates the second operand only if the first is false.

Example:

```
boolean result = (x > y) || (a > b);

System.out.println(result); // Output: true
```

Explanation: The first condition (x > y) is true, so the result is true regardless of the second condition.

3. Logical NOT (!)

- **!**: Inverts the boolean value of an expression.

Example:

```
boolean result = !(x < y);
System.out.println(result); // Output: true
```

Explanation: Since (x < y) is false, !(x < y) becomes true.

4. Logical XOR (^)

- If precisely one operand is true, the function returns true; if not, it returns false.

Example:

```
boolean result = (x > y) ^ (a < b);
System.out.println(result); // Output: false
```

Explanation: Both (x > y) and (a < b) are true, so the XOR result is false.

Example Program Combining All Logical Operators

```
public class LogicalOperatorsExample {
    public static void main(String[] args) {
        int x = 7, y = 5, a = 5, b = 9;
```

```
        // AND operation
        boolean andResult = (x > y) && (a < b);
        System.out.println("AND Result: " + andResult); // Output: true

        // OR operation
        boolean orResult = (x > y) || (a > b);
        System.out.println("OR Result: " + orResult); // Output: true

        // NOT operation
        boolean notResult = !(x < y);
        System.out.println("NOT Result: " + notResult); // Output: true

        // XOR operation
        boolean xorResult = (x > y) ^ (a < b);
        System.out.println("XOR Result: " + xorResult); // Output: false
    }
}
```

Key Points

- **Logical AND (&&)** and **Logical OR (||)**

  are short-circuit operators, meaning they can skip evaluating the second operand if the result is already determined by the first operand.

- **Logical NOT (!)**

  negates the boolean value of an expression.

- **Logical XOR (^)**

  is true only if exactly one of the operands is true.

Sec-06-01-If-Else Statement

Conditional statement:

Conditional statements in programming are essential for controlling the flow of a program based on specific conditions. These statements enable the execution of different blocks of code depending on whether a given condition evaluates to true or false. This mechanism is fundamental for decision-making in algorithms.

Example:

Consider an automatic car or vehicle, which adjusts its speed based on road conditions. Here's how it might work:

1. **Free Road/Highway:** The car increases its speed when the road is clear.
2. **Heavy Traffic:** The car reduces its speed automatically when there is heavy traffic.
3. **Obstacle Detection:** The car stops if an obstacle or person comes in front of it.

These actions are based on conditions, demonstrating the practical use of conditional statements.

if-else Statement

The if-else statement extends the if statement by adding an else clause. If the condition is true, the program executes the code in the if block or else it is false then statement in else block is executed.

To start with, let's explore the if-else statement:

```
int x = 8;
System.out.println("hello");
System.out.println("bye");
```

// Applying an if condition

1. if (x > 10) {

```
   // The condition requires a boolean value (true or false).
   System.out.println("hello");
}
```

// No output for this condition as x is not greater than 10

2. int x = 18;

```
if (x > 10 && x <= 20) {  //using logical operations with two expressions.
```

```
    System.out.println("hello"); // Output: hello
}
```

Syntax of if-else

In Java, the syntax for an if-else statement is as follows:

- If the condition is true, the block of code within the if statement is executed.
- If there is only one statement to execute, curly braces {} are optional.
- If there are multiple statements, curly braces are mandatory to group them together.

```
3. int x = 15;
if (x > 10) {
    System.out.println("Value is greater than 10");
} else {
    System.out.println("Value is 10 or less");
}
```

Output:

Value is greater than 10

Key Points to Remember

- Indentation in Java does not affect the execution but helps in making the code clean and easy to understand.
- if conditions always evaluate to a boolean values (true or false).

02-If Else If

If else if

The if-else if statement allows for multiple conditions to be checked in sequence. If the if condition is false, the program checks the next else if condition, and so on.

Syntax of If-Else if Conditional Statement:

```
if (condition1) {
    // code to execute if condition1 is true
} else if (condition2) {
    // code to execute if condition2 is true
} else {
    // code to execute if all conditions are false
}
```

Example:

To determine the greatest value among three variables (x, y, and z), we can use if-else if statements along with logical operators. This is a common programming scenario where conditional logic helps in decision-making based on multiple criteria.

Example 1: Basic Comparison

```
int x = 8, y = 7, z = 6;
if (x > y && x > z) {
    System.out.println(x); }
else {
    System.out.println(y);
}
```

o/p:8

Example 2: Adjusted Values

```
z = 9;
if (x > y && x > z) {
    System.out.println(x);
} else {
```

```
    System.out.println(y); // Output: 7
}
```

Example 3: Additional Condition

```
y = 17;
if (x > y && x > z) {  // Checking if x is the greatest
    System.out.println(x);
} else if (y > x && y > z) {  // Checking if y is the greatest
    System.out.println(y); // Output: 17
}
```

Example 4: Complete Comparison with else Block

```
x = 8;
y = 7;
z = 9;

if (x > y && x > z) {  // Checking if x is the greatest
    System.out.println(x);
} else if (y > z) {  // Checking if y is the greatest
    System.out.println(y);
} else {
    System.out.println(z); // Output: 9
}
```

Key Points

- Logical Operators: Used to combine multiple conditions (e.g., && for "and").
- Conditional Blocks: Use curly braces {} for clarity, especially with multiple statements.
- Complete Conditions: Always cover all possible scenarios, including the final else block.

03-Ternary

The **ternary operator** is a concise way to write an if-else statement. It involves three operands: a condition, a result for when the condition is true, and a result for when the condition is false. This operator can help reduce the lines of code required for simple conditional assignments.

Syntax of Ternary Expression

condition ? result_if_true : result_if_false;

Implementation of Ternary Expression

Checking if a Number is Even or Odd: Using Simple if-else Approach

```
int n = 4;
int res = 0;

if (n % 2 == 0) {
    res = 10;
} else {
    res = 20;
}

System.out.println(res); // Output: 10
```

Checking if a Number is Even or Odd: Using Ternary Operator

```
int n = 4;
int res = (n % 2 == 0) ? 10 : 20;
System.out.println(res); // Output: 10
```

Example with Different Input:

```
int n = 5;
int res = (n % 2 == 0) ? 10 : 20;
```

System.out.println(res); // Output: 20

The ternary operator provides a one-line solution for conditional expressions, making the code more compact. However, it may not always be suitable for complex if-else statements that require multiple actions or complex logic.

Key Points

- **Conciseness:** The ternary operator reduces the number of lines needed for simple condition checks.
- **Readability:** While the ternary operator can make the code more compact, it should be used carefully to maintain readability, especially for more complex conditions.
- **Limitations:** It is best suited for straightforward if-else scenarios and may not work well for cases involving multiple statements or complex logic.

04-Switch Statement

The switch statement is used to evaluate a variable against a series of values, offering a more readable alternative to lengthy if-else-if chains. Each block in a switch statement is represented by a case, and typically, the execution of a case block is terminated using the break keyword.

Syntax of Switch Conditional Statement:

```
switch (variable) {
    case value1:
        // code to execute if variable equals value1
        break;
    case value2:
        // code to execute if variable equals value2
        break;
    default:
        // code to execute if variable doesn't match any case
}
```

Use Cases of Switch Statement:

- **Selecting a Code Block**: Efficiently choose one among many code blocks to execute based on the value of a variable.
- **Handling Multiple Cases**: Manage multiple scenarios clearly and concisely.

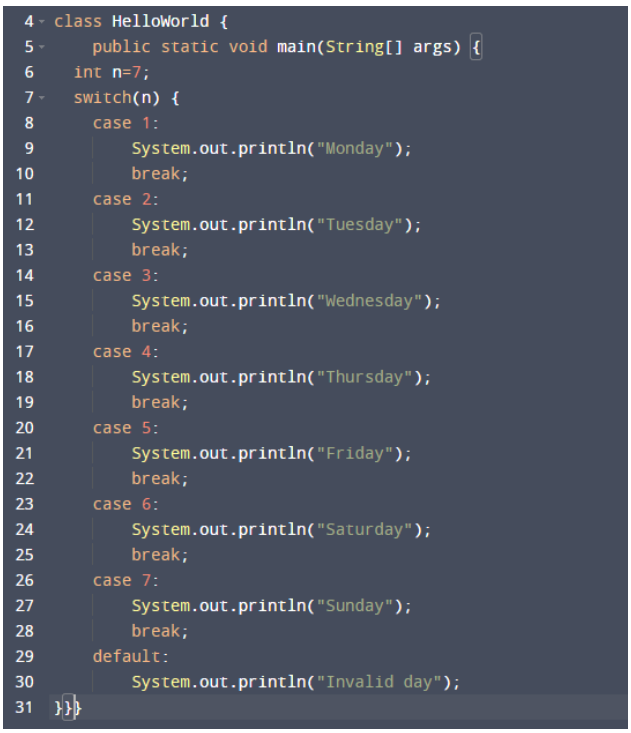
Example: Printing Weekdays

Let's consider an example where we want to print the name of the day based on a number:

- **Using if-else-if approach**: If n = 1 to 7, we need to use multiple if-else blocks to print the corresponding weekday.
- **Using switch statement**: We can replace the complex if-else chain with a switch statement for clarity and efficiency.

```
switch(n) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
```

```
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
    case 7:
        System.out.println("Sunday");
        break;
    default:
        System.out.println("Invalid day");
}
```



```
class HelloWorld {
    public static void main(String[] args) {
   int n=7;
   switch(n) {
     case 1:
         System.out.println("Monday");
         break;
     case 2:
         System.out.println("Tuesday");
         break;
     case 3:
         System.out.println("Wednesday");
         break;
     case 4:
         System.out.println("Thursday");
         break;
     case 5:
         System.out.println("Friday");
         break;
     case 6:
         System.out.println("Saturday");
         break;
     case 7:
         System.out.println("Sunday");
         break;
     default:
         System.out.println("Invalid day");
}}}
```

Output:

```
Output
java -cp /tmp/rhgRKF5CpV/HelloWorld
Sunday

=== Code Execution Successful ===
```

In this example, if n is 2, the output will be "Tuesday".

```
class HelloWorld {
    public static void main(String[] args) {
  int n=2;
  switch(n) {
    case 1:
        System.out.println("Monday");
        //break;
    case 2:
        System.out.println("Tuesday");
       // break;
    case 3:
        System.out.println("Wednesday");
       // break;
    case 4:
        System.out.println("Thursday");
       // break;
    case 5:
        System.out.println("Friday");
       // break;
    case 6:
        System.out.println("Saturday");
      //  break;
    case 7:
        System.out.println("Sunday");
     //   break;
    default:
        System.out.println("Invalid day");
}}}
```

Output:

```
java -cp /tmp/x9N2MjZZQr/HelloWorld
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Invalid day

=== Code Execution Successful ===
```

The break statement prevents the execution from falling through to subsequent cases. Without the break statement, the program would continue executing the following cases until it encounters a break or reaches the end of the switch block.

The default case is executed if none of the specified cases match the variable's value, ensuring that the program handles unexpected values gracefully.

Note: The break keyword is crucial in Java to prevent fall-through behaviour.

05-Need For Loops

Need For Loops.

Loops, also known as iterative statements, are crucial in programming for executing a block of code repetitively. They allow for the repeated execution of a set of instructions or a code block as long as a specified condition is met. Loops are fundamental to the concept of iteration, enhancing code efficiency, readability, and promoting the reuse of code logic.

Example: Iterative Statements Without and With Loops

Without Loops: Consider a scenario where you want to print numbers from 1 to 5. Without using loops, you would need to write multiple lines of code:

```
System.out.println(1);

System.out.println(2);

System.out.println(3);

System.out.println(4);

System.out.println(5);
```

This approach is verbose and not scalable.

With Loops: The same task can be accomplished using a loop, which is more compact and efficient:

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i);
}
```

This loop iterates from 1 to 5, printing each number, demonstrating the power of loops in reducing code complexity.

Types of Loops

Loops can be categorized based on the control mechanism into two main types: entry-controlled loops and exit-controlled loops.

1. **Entry-Controlled Loops:**
    - The test condition is checked before entering the main body of the loop.
    - Examples: for loop, while loop.

Example of an Entry-Controlled Loop:

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

In this example, the condition i < 10 is checked before executing the loop body.

2. **Exit-Controlled Loops:**
    - The test condition is evaluated at the end of the loop body, ensuring that the loop body will execute at least once, regardless of whether the condition is true or false.
    - Example: do-while loop.

Example of an Exit-Controlled Loop:

```
int i = 0;

do {
   System.out.println(i);
   i++;
} while (i < 10);
```

In this example, the do-while loop executes the loop body first and then checks the condition i < 10.

06-While Loop

While Loop

A while loop is an entry-controlled control flow structure in programming that repeatedly executes a block of code as long as a specified condition remains true. The loop continues to iterate while the condition is true and terminates once the condition evaluates to false.

Basic Syntax of While Loop:

```
while (condition) {
    // code block to be executed
}
```

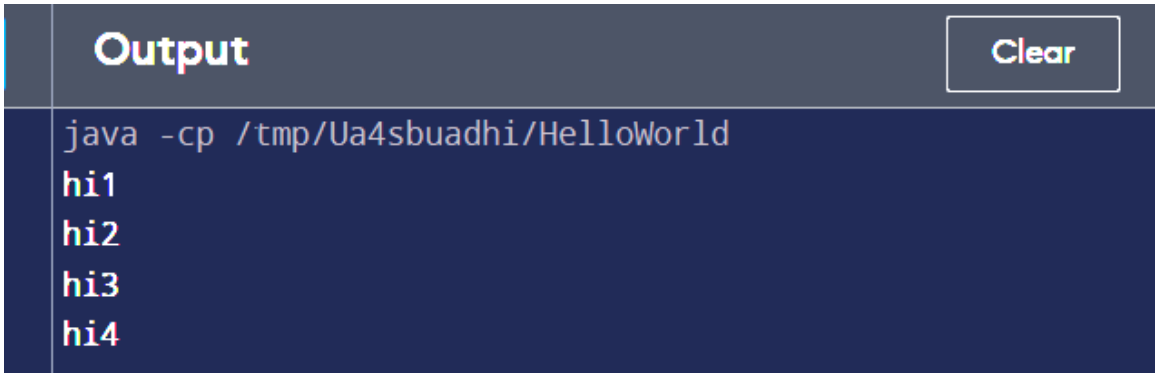
In this structure, the condition is checked before each iteration. If the condition evaluates to true, the loop body (code block) is executed. If the condition evaluates to false, the loop terminates.

Example: Printing "Hi" Multiple Times

To print "Hi" four times using a while loop, you can use a counter variable that increments with each iteration. This counter helps control the number of times the loop executes, preventing infinite loops.

Example Code:

```
int i = 0; // Initialize counter variable
while (i < 4) { // Loop continues while i is less than 4
    System.out.println("Hi");
    i++; // Increment counter
}
```



In this example:

- The counter variable i is initialized to 0.

- The condition i < 4 ensures the loop runs only four times.
- The System.out.println("Hi"); statement prints "Hi" during each iteration.
- The counter i increments by 1 after each iteration, eventually making the condition false and terminating the loop.

Nested While Loops: Printing "Hello" After "Hi"

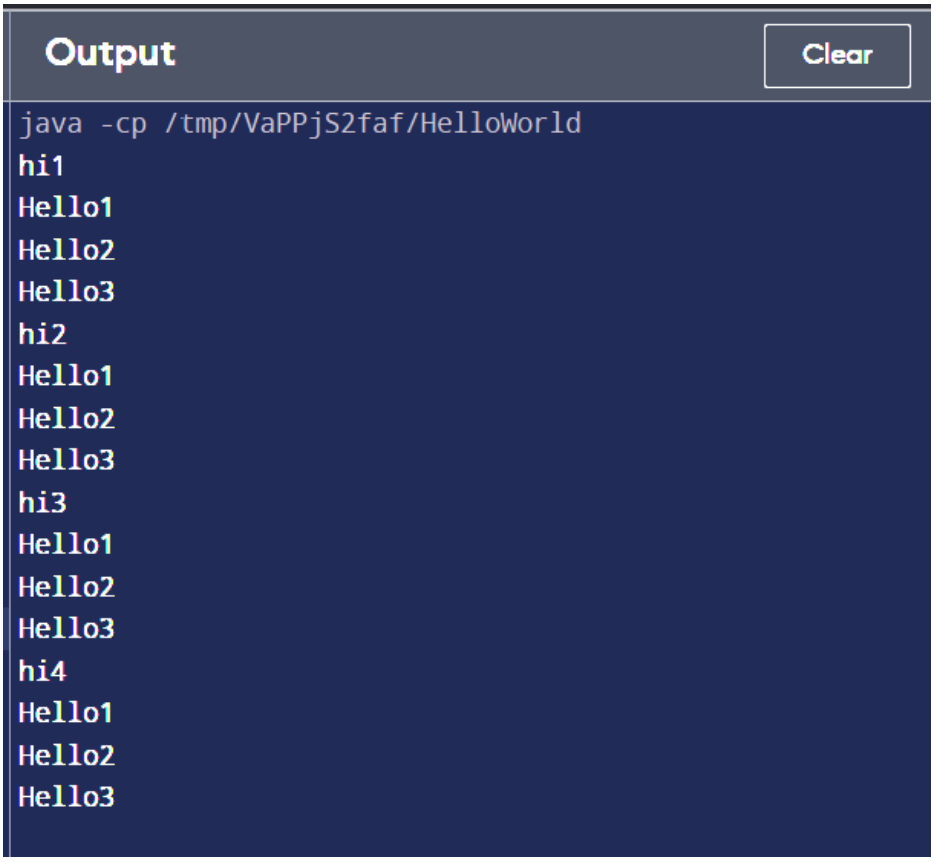
To print "Hello" three times after each "Hi", you can use nested while loops. A nested loop runs inside another loop, with the inner loop completing all its iterations before the outer loop proceeds to the next iteration.

Example Code with Explanation:

```
int i = 0; // Outer loop counter
while (i < 4) {
    System.out.println("Hi");
    int j = 0; // Inner loop counter
    while (j < 3) {
        System.out.println("Hello " + (j + 1)); // Preceding with the number of times printed
        j++; // Increment inner loop counter
    }
    i++; // Increment outer loop counter
}
```

In this example:

- The outer loop (controlled by i) prints "Hi" four times.
- For each iteration of the outer loop, the inner loop (controlled by j) prints "Hello" three times, preceded by the number indicating the print count.

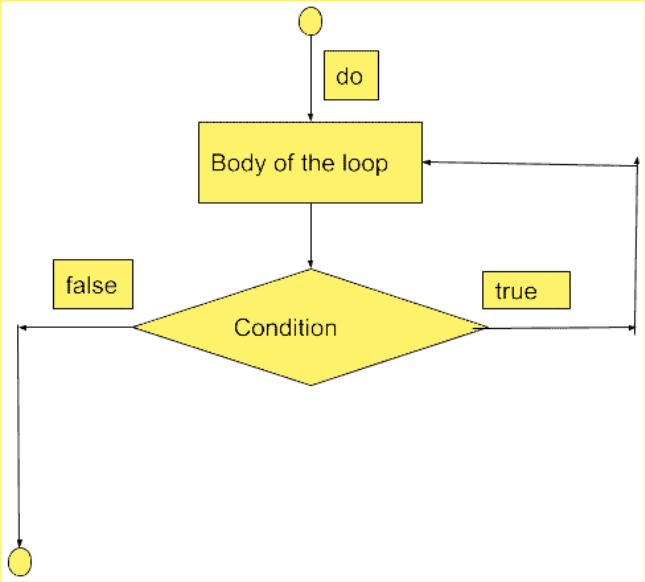


07-Do While Loop

A do-while loop is an exit-controlled control flow structure in programming. It ensures that a block of code is executed at least once and then continues to execute the block as long as a specified condition remains true. The key feature of a do-while loop is that the condition is evaluated after the code block, guaranteeing at least one execution, even if the condition is initially false.

Syntax of Do-While Loop:

```
do {
   // code block to be executed
} while (condition);
```



In this structure, the code block inside the do section is executed first. After execution, the condition in the while statement is evaluated. If the condition evaluates to true, the loop iterates again; if false, the loop terminates.

Example: Do-While Loop Usage

Let's illustrate this with an example where we want to ensure that a message is printed at least once, regardless of the condition.

Example Code:

```
public class Main {
   public static void main(String[] args) {
      int i = 1;
```

```
        do {
            System.out.println(i);
            i++;
        } while (i <= 4);
    }
}
```

In this example:

- The variable i is initialized to 1.
- The do block contains the code to print the value of i and increment it by 1.
- The loop continues to run while i <= 4. As a result, numbers from 1 to 4 are printed.
- Even if i was initially greater than 4, the message would still print once because the condition is checked after the code block execution.

Differences Between While and Do-While Loops

- **Execution Guarantee**: The do-while loop guarantees that the code block is executed at least once, even if the condition is false from the beginning. In contrast, a while loop checks the condition before execution and may not run at all if the condition is false initially.
- **Condition Check**: In a while loop, the condition is checked before the loop body; in a do-while loop, the condition is checked after the loop body.

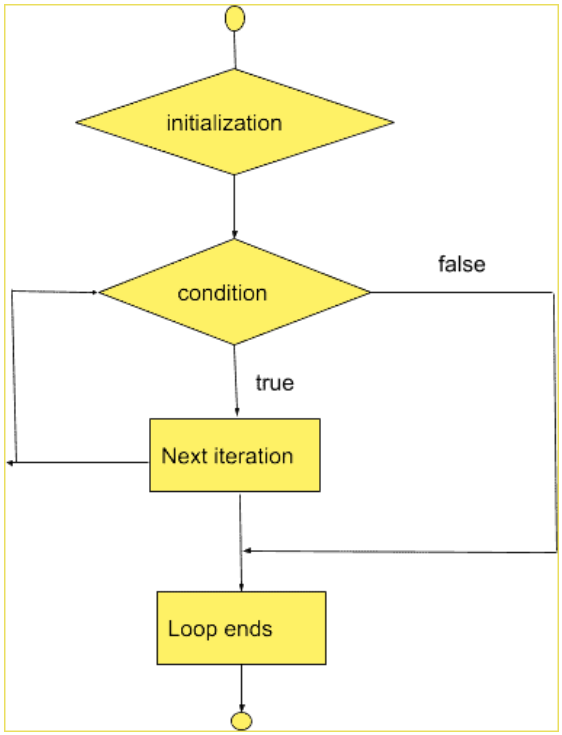
08-For Loop

A for loop is a control flow structure in programming that iterates over a sequence of elements, such as a range of numbers, items in a list, or characters in a string. It is an entry-controlled loop, meaning it determines the number of iterations before entering the loop. The for loop provides a concise way of writing the loop control structure in a single line, unlike other loops where initialization, condition, and increment/decrement steps are specified separately.

Syntax of For Loop:

```
for (initialization; condition; update) {
    // code block to be executed
}
```

- **Initialization**: This step sets the starting value of the loop control variable.
- **Condition**: This is the test condition that must be true for the loop to continue executing.
- **Update**: This modifies the loop control variable after each iteration.

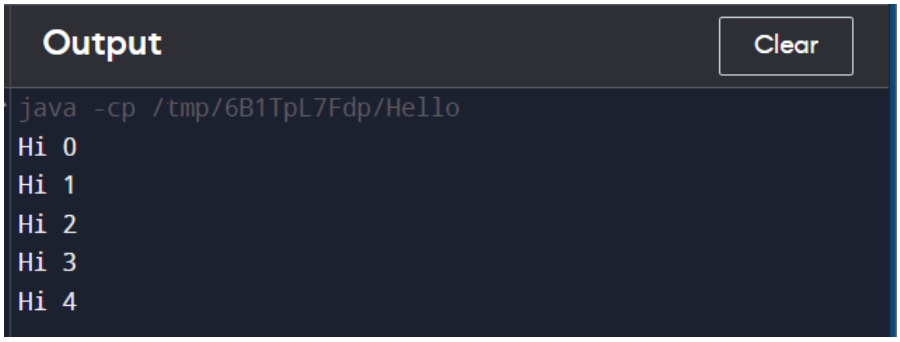


Example and Output

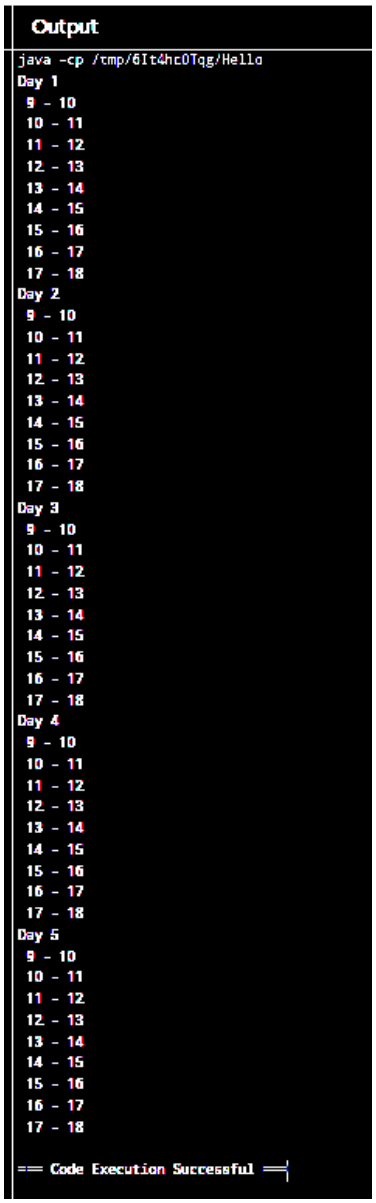
Code Example:

```
public class Hello {
```

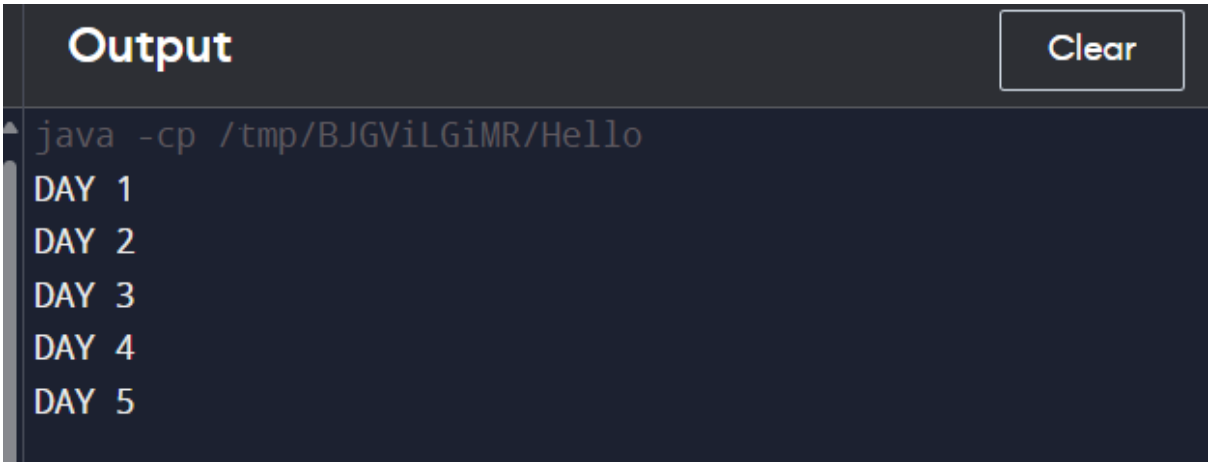
```
public static void main(String[] args) {
    // Example 1: Printing "Hi" with indices
    for (int i = 0; i <= 4; i++) {
        System.out.println("Hi " + i);
    }
```




```
    // Example 2: Nested for loops to display days and working hours
    for (int i = 1; i <= 7; i++) {
        System.out.println("Day " + i);
        for (int j = 1; j <= 9; j++) {
            System.out.println(" " + (j + 8) + " - " + (j + 9));
        }
    }
```




```
        // Example 3: For loop with initialization outside the loop
        int i = 1;
        for (; i <= 5;) {
            System.out.println("DAY " + i);
            i++;
        }
    }
}
```


Output
Clear
java -cp /tmp/BJGViLGiMR/Hello
DAY 1
DAY 2
DAY 3
DAY 4
DAY 5

6.9-Which loop to use

Recommended Loops to Use in Programs

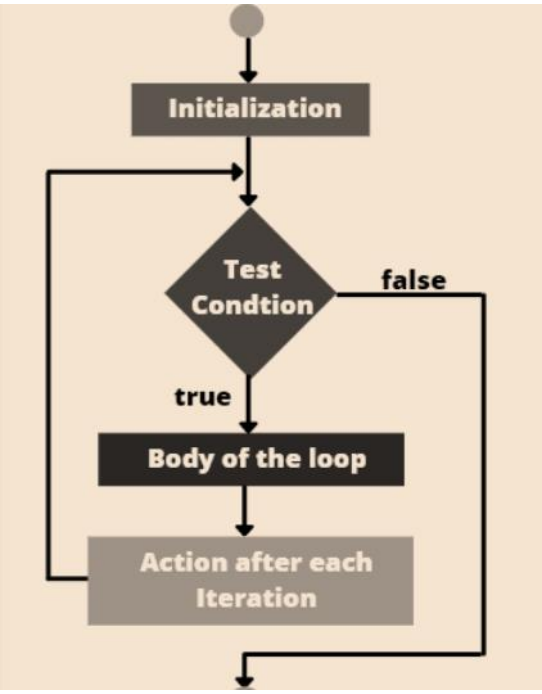
In Java, there are three primary loops that we use to perform iterations:

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

Each of these loops is useful in different scenarios, depending on the conditions and requirements of the task at hand. While all loops perform iterations, the choice of which loop to use depends on the specific situation. Let's break it down:

1. for Loop

- **When to use:**
  A for loop is ideal when **you know the exact number of iterations** in advance. It combines the initialization, condition checking, and increment/decrement in one statement, making it concise and clear.
- **Common usage:**
  For tasks like counting, iterating over arrays or lists, or performing a set number of iterations, the for loop is highly recommended.

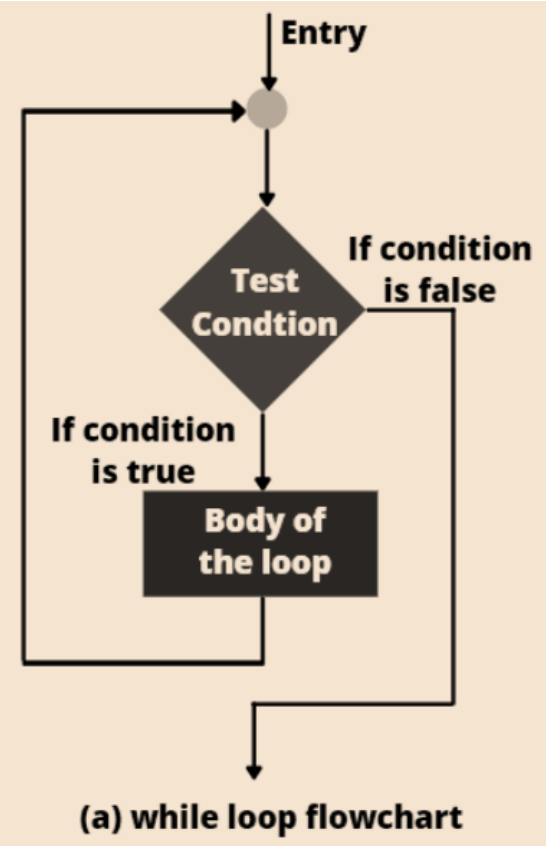


- **Example:**

```java
for (int i = 0; i < 5; i++) {
    System.out.println("Iteration: " + i);
}
```

2. while Loop

- **When to use:**
  A while loop is preferred when **you don't know in advance how many iterations** are needed, but the condition will determine when the loop should terminate. This is common in scenarios like reading from a file, where the number of iterations depends on external factors (e.g., the content of the file).
- **Analogy:**
  Think of it as **"As long as"** – as long as the condition holds true, the loop will keep running.



(a) while loop flowchart

-

- **Example:**

```
while (file.hasNextLine()) {
    String line = file.nextLine();
    System.out.println(line);
}
```

3. do-while Loop

- **When to use:**
  A do-while loop is useful when you **need to execute the loop at least once**, even if the condition is initially false. This is because the condition is checked after the first iteration.

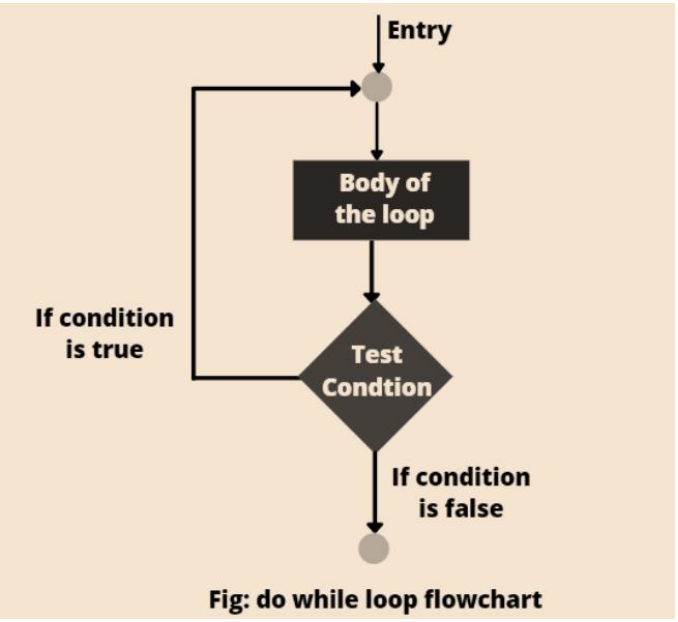


Fig: do while loop flowchart

- **Example:**

```
int count = 0;
do {
    System.out.println("Iteration: " + count);
    count++;
} while (count < 5);
```

Choosing the Right Loop

- **Use a for loop** when you know the exact number of iterations.
- **Use a while loop** when the number of iterations is unknown and depends on a condition being met (e.g., reading a file or fetching data from a database).
- **Use a do-while loop** when you want the loop to run at least once, regardless of the condition.

Real-World Usage

- In **enterprise-level applications**, while loops are often used for tasks like fetching data from a database, reading files, or processing user input. These tasks usually require loops that continue until a condition (e.g., end of file or no more data) is met.
- Although **all loops perform the same function** (i.e., iteration), choosing the right loop enhances code readability and efficiency. In practice, for loops are commonly used when the number of iterations is clear, while while loops are favored for data-driven operations.

02-Class and Object Theory

Introduction

- Java is primarily an object-oriented programming language. While it extensively uses objects and classes to structure code, it also includes primitive data types. Classes and objects are fundamental concepts in Java's implementation of Object-Oriented Programming (OOP), used to represent real-world concepts and entities.

Understanding Objects

- An object has properties (attributes) and functions (methods)—it knows something and does something. For example, a pen has attributes such as height, dimensions, and color, and it has functions such as writing.

Java Classes

- A class in Java is a blueprint for creating objects. It defines common behaviors and properties shared by all objects of that class. Classes are user-defined data types that act as templates for objects.

Java Objects

- An object in Java is an instance of a class. It is a basic unit of Object-Oriented Programming and represents real-life entities.

Object-Oriented Programming (OOP)

- In OOP, an object has properties (attributes) and behaviors (methods). To create an object, the first step is to create a class. The class serves as the blueprint for the object.
- Once the class is created, the Java compiler compiles the class file into bytecode, which is then used to generate the object. The JVM executes the bytecode and manages object creation at runtime.

03-Class and Object Practical

We will first write a simple program to add two numbers in the Demo class. Then, we will create a separate Calculator class to handle the addition functionality. This will make our code modular and reusable.

Simple Addition in Demo Class

```
public class Demo {
    public static void main(String[] args) {
        int num1 = 4;
        int num2 = 5;

        int result = num1 + num2;
        System.out.println(result);
    }
}
```

Creating the Calculator Class

We will create a Calculator class with an add method to handle the addition. This will allow us to reuse the Calculator class for various addition operations.

```
public class Calculator {

    // Method to add two numbers
    public int add(int n1, int n2) {
        System.out.println("In add method");
        int res = n1 + n2;
        return res;
    }
}
```

Accessing the Calculator Class Method in Demo Class

To use the add method from the Calculator class in the Demo class, we need to create an object of the Calculator class. This is done using the new keyword.

Complete Code

Calculator Class

```
public class Calculator {


    // Method to add two numbers
    public int add(int n1, int n2) {
        System.out.println("In add method");
        int res = n1 + n2;
        return res;
    }
}
```

Demo Class

```
public class Demo {
    public static void main(String[] args) {
        int num1 = 4;
        int num2 = 5;
        // Creating an object of the Calculator class
        Calculator calc = new Calculator();


        // Calling the add method of Calculator class
        int result = calc.add(num1, num2);
        // Printing the result
        System.out.println("The sum is: " + result);
    }
}
```

Explanation of Code Using Steps

1. Define the Calculator Class
    - Class Definition: public class Calculator { }
    - Method to Add Two Numbers:

```
public int add(int n1, int n2) {
    System.out.println("In add method");
    int res = n1 + n2;
    return res;
}
```

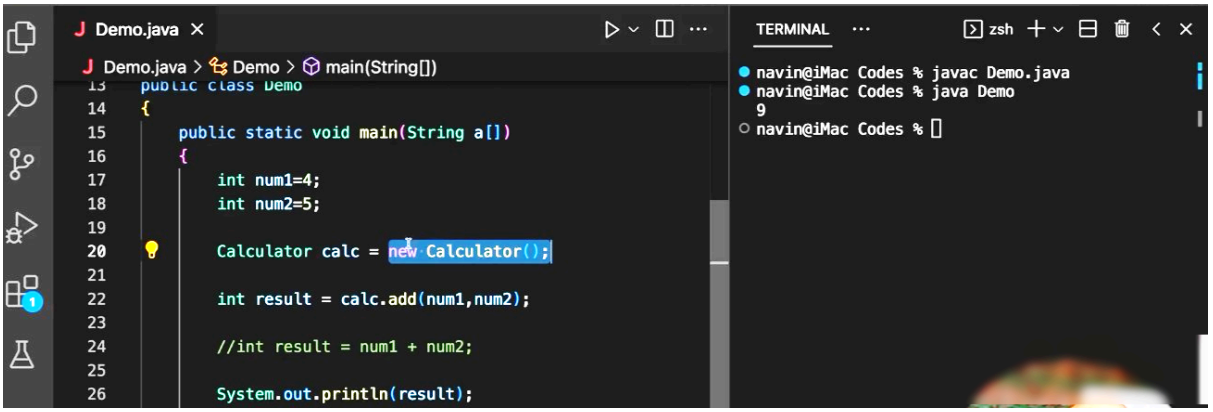
2. Create the Demo Class
    - Class Definition: public class Demo { }
    - Main Method: public static void main(String[] args) { }
3. Initialize Variables in Main Method
    - Define Variables:

```
int num1 = 4;
int num2 = 5;
```

4. Create an Object of the Calculator Class
    - Object Creation: Calculator calc = new Calculator();
        - Explanation:
            - Calculator is the class name.
            - calc is a reference variable (similar to how num1 and num2 are variables, but it references an object).
            - new Calculator() creates a new object of the Calculator class.
            - The JVM handles the creation of the object and allocates memory for it.
5. Call the Add Method Using the Calculator Object

- Method Call: int result = calc.add(num1, num2);
  - Explanation:
    - calc.add(num1, num2) calls the add method of the Calculator class using the calc object.
    - The method adds num1 and num2, prints "In add method", and returns the result.

6. Print the Result
   - Output: System.out.println("The sum is: " + result);
     - This line prints the sum of num1 and num2.



Summary

By separating the addition functionality into the Calculator class, we make our code more modular and reusable. The Demo class then creates an object of the Calculator class and calls the add method to perform the addition. This approach follows the principles of object-oriented programming and promotes better code organization and reuse.

04-JDK JRE JVM

JDK, JRE, and JVM

Understanding the Java Development Kit (JDK), Java Runtime Environment (JRE), and Java Virtual Machine (JVM) is crucial for Java development.

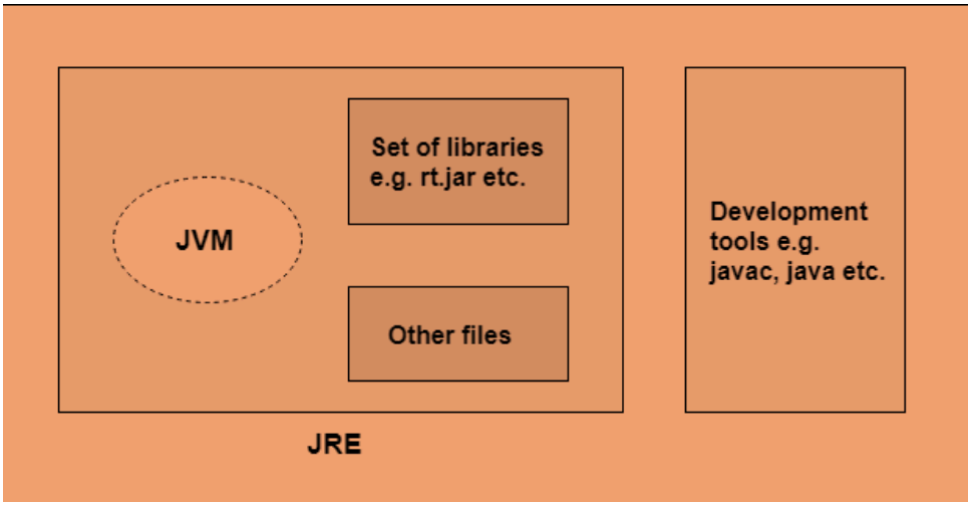
Java Development Kit (JDK)

Definition: The JDK is a software development environment used for developing Java applications and applets. It is a comprehensive package containing tools necessary for Java development.

Components:

- JRE (Java Runtime Environment): provides the libraries and components needed to run Java applications.
- JVM (Java Virtual Machine): Executes Java bytecode.
- Development Tools: Includes a launcher (java) for running Java applications, a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools necessary for Java development.

Purpose: The JDK is required to write, compile, and debug Java programs.



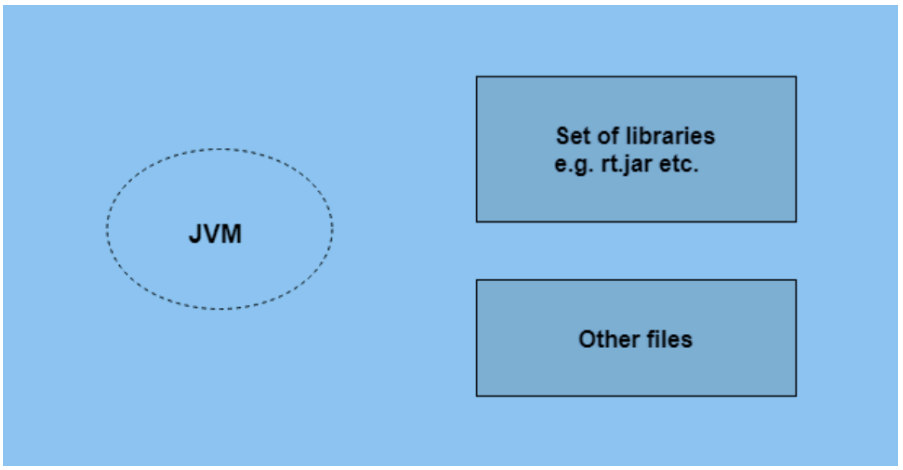
Java Runtime Environment (JRE)

Definition: The JRE is a set of software tools for running Java applications. It includes the JVM and other components that support the execution of Java programs.

Components:

- JVM: The engine that executes Java bytecode.
- Libraries: A set of class libraries and other files used by the JVM at runtime.

Purpose: The JRE provides the necessary environment to run Java applications by offering libraries and other required files.



Java Virtual Machine (JVM)

Definition: The JVM is an abstract machine that provides a runtime environment in which Java bytecode can be executed. It is called a virtual machine because it doesn't physically exist as hardware but is instead a specification that is implemented in software.

Key Functions:

- Class Loader: loads class files.
- Bytecode Verifier: Ensures bytecode is valid and adheres to security constraints.
- Interpreter: Converts bytecode into machine code for execution.
- Just-In-Time (JIT) Compiler: Improves performance by compiling bytecode to native machine code at runtime.

Purpose: The JVM executes Java bytecode and can also run programs written in other languages compiled to Java bytecode.

Relationship between JDK, JRE, and JVM

- JDK: The topmost layer, containing both JRE and JVM along with development tools.
- JRE: sits within the JDK, providing the libraries and JVM required to run Java applications.
- JVM: Part of the JRE, responsible for executing Java bytecode.

05-Methods

Methods in Java

Definition: Methods are functions or behaviours within a class that perform specific tasks. A method in Java is a collection of statements that execute certain operations and can return a result to the caller. However, not all methods must return a value. Java methods allow for code reuse, reducing the need to rewrite the same code multiple times. Unlike some other programming languages like C, C++, and Python, in Java, every method must belong to a class.

A method can be compared to a function in that it exposes the behaviour of an object. It consists of a set of code that performs a particular task. For example, the main method is the starting point of a Java program, where the execution begins.

Syntax of a Method:

```
<access_modifier> <return_type> <method_name>(<list_of_parameters>)
{
    // method body
}
```

Advantages of Using Methods:

- **Code Reusability:** Methods allow the reuse of code, eliminating the need to write the same code multiple times.
- **Code Optimization:** By using methods, the code becomes more organized and easier to manage.

Example of Methods in Real Life

Consider the process of building a product. The product is made up of various components, and these components are created using modules. In Java, a class represents a component, and the functions or behaviours of that component are represented by methods.

Example:
If we consider a car as the final product, its components are the wheels, engine, dashboard, etc. Each of these components corresponds to a class, and the behaviour or functions of each component (like the rotation of wheels or the ignition of the engine) are represented by methods within those classes.

Types of Methods in Java

There are two primary types of methods in Java:

1. **Predefined Methods:**

- These methods are already defined in the Java class libraries and can be used directly in your programs. They are also known as standard library methods or built-in methods.
- **Example:** System.out.println() is a predefined method in Java.

2. **User-Defined Methods:**
   - These are methods created by the programmer to perform specific tasks as per the requirements.
   - **Example:**

```
public void playMusic() {
    System.out.println("Playing music");
}

public String giveMePen(int cost) {
    if (cost >= 10)
        return "Pen";
    return "Nothing";
}
```

   - **Usage:**

```
public static void main(String[] args) {
    Computer obj = new Computer();
    obj.playMusic();
    String str = obj.giveMePen(10);
    System.out.println(str);
}
```

Static Method

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword **static** before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the **main()** method.

Example:

```
public class Display
{
public static void main(String[] args)
{
show();
}
static void show()
{
System.out.println("It is an example of static method.");
}
}
```

Instance Method

The method of the class is known as an **instance method**. It is a **non-static** method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

Example:

```
public class Calculator
{
public static void main(String [] args)
{
//Creating an object of the class
Calculator   obj = new Calculator   ();
//invoking instance method
System.out.println("The sum is: "+obj.add(12, 13));
}
int s;
//user-
defined method because we have not used static keyword
public int add(int a, int b)
{
s = a+b;
//returning the sum
return s;
}
}
```

Abstract Method

The method that does not has method body is known as abstract method. In other words, without an implementation is known as abstract method. It always declares in the **abstract class**. It means the class itself must be abstract if it has abstract method. To create an abstract method, we use the keyword **abstract**.

Example:

```
abstract class Demo //abstract class
{
//abstract method declaration
abstract void display();
}
public class MyClass extends Demo
{
//method impelmentation
void display()
{
System.out.println("Abstract method?");
}
public static void main(String args[])
{
//creating object of abstract class
Demo obj = new MyClass();
//invoking abstract method
obj.display();
}
}
```

Final Method:

The **final** is a keyword in java. The final keyword is used to denote the constants.
The final keyword is usually used with methods, classes, and variables. Whenever we define any variable or class, or method as final, then the particular entity cannot be changed after it has been assigned the final method in java is the method that cannot be changed or overridden once defined. Hence, we cannot modify a final method from the subclass.

Example:

```
class Calculator {
    // defining a final method inside the parent class.
    public final void display() {
        System.out.println("This is a final method.");
    }
}

public class Demo extends Calculator {
    // overriding the final method of the parent class.
    public final void display() {
        System.out.println("Overriding the final method.");
    }

    public static void main(String[] args) {
        // creating an object of the child class (Test)
        Demo obj = new Demo ();
```

```
        // calling the final method from the child class object.
        obj.display();
    }
}
```

Output:

```
Demo.java:8: error: display() in Demo cannot override
display() in Calculator
    public final void display() {
                      ^
  The overridden method is final
1 error
```

FAQs on Methods in Java

Q1. What is a method in Java programming?
A method in Java is a collection of statements that perform a specific task and return the result to the caller.

Q2. What are the parts of a method in Java?
The main components of a method in Java include:

- **Modifiers:** Keywords that define the access level and other properties of the method.
- **Return Type:** The data type of the value the method returns (e.g., void if it returns nothing).
- **Method Name:** The name of the method used to call it.
- **Parameters:** A list of input values the method accepts.
- **Method Body:** The block of code that defines what the method does.

06-Method Overloading

Method Overloading in Java

Definition:

Method Overloading in Java allows multiple methods to share the same name but differ in their signatures. A method's signature can vary by:

- The number of input parameters
- The type of input parameters
- A combination of both the number and type of input parameters

This concept is also known as **Compile-time Polymorphism**, **Static Polymorphism**, or **Early Binding**. When a method is overloaded, Java gives priority to the method that matches the most specific signature during compilation.

Example:

Let's consider a class Calculator with a method that adds two numbers and returns the result as an integer:

```
class Calculator {
    public int add(int n1, int n2) {
        return n1 + n2;
    }

    public static void main(String[] args) {
        Calculator obj = new Calculator();
        int result = obj.add(2, 5);
        System.out.println(result); // Outputs: 7
    }
}
```

Suppose the requirements change, and you now need to add three numbers instead of two. Without method overloading, you would need to modify the existing method or create a new one with a different name. However, if requirements frequently change, this approach becomes cumbersome.

With method overloading, you can create multiple methods with the same name but different parameters:

```
class Calculator {
    public int add(int n1, int n2) {
        return n1 + n2;
    }

    public int add(int n1, int n2, int n3) {
        return n1 + n2 + n3;
```

```
    }

    public double add(double n1, int n2) {
        return n1 + n2;
    }
}
```

Different Ways of Method Overloading in Java:

1. **Changing the Number of Parameters:**
    - Method overloading can be achieved by altering the number of parameters passed to different methods.
    - Example:

```
public int add(int n1, int n2) {
    return n1 + n2;
}

public int add(int n1, int n2, int n3) {
    return n1 + n2 + n3;
}
```

2. **Changing the Data Types of the Arguments:**
    - Methods can be overloaded if they have the same name but different parameter types.
    - Example:

```
public int add(int n1, int n2) {
    return n1 + n2;
}

public double add(double n1, double n2) {
    return n1 + n2;
}
```

3. **Changing the Order of the Parameters:**
    - Method overloading can also be implemented by rearranging the order of parameters in two or more methods with the same name.
    - Example:

```
public void display(String name, int rollNo) {
    System.out.println("Name: " + name + ", Roll No: " +
rollNo);
}

public void display(int rollNo, String name) {
    System.out.println("Roll No: " + rollNo + ", Name: " +
name);
}
```

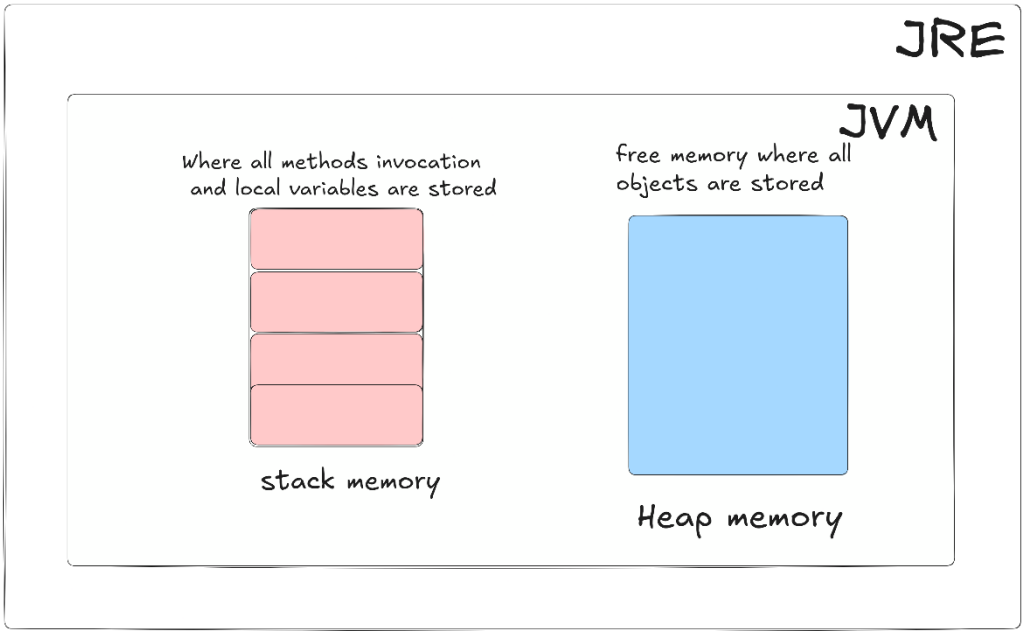
Advantages of Method Overloading:

- **Improved Readability:** Overloaded methods allow the use of the same method name for similar tasks, making the code easier to understand.
- **Code Reusability:** The same method name can be used for different operations, reducing redundancy in code.
- **Reduced Complexity:** Overloading allows related methods to be grouped together, simplifying the program structure.
- **Efficiency:** Programmers can perform tasks more effectively by using method overloading to handle different types or numbers of arguments with minimal code changes.
- **Flexible Object Initialization:** Objects of a class can be initialized in different ways using overloaded constructors.

07-Stack and Heap

Understanding Stack and Heap Memory in Java

In Java, memory management is crucial for efficient application performance. The Java Virtual Machine (JVM) divides memory into two primary areas: **Stack Memory** and **Heap Memory**. Understanding how these memory areas work is key to writing optimized Java programs.



What's Inside the JVM?

The JVM manages the execution of Java code, including memory allocation and deallocation. Here's an overview of how memory is structured within the JVM:

1. Stack Memory

- **What is Stack Memory?**
    - Stack memory is allocated to each thread at runtime and is used to store:
        - Local variables
        - Method call details (like the order of execution)
        - References to objects and partial results
- **How Does Stack Memory Work?**

- It follows the **Last-In-First-Out (LIFO)** principle, meaning the last item added to the stack is the first one to be removed. Each method in Java has its own stack, and the memory is automatically freed up when the method completes.

- **Code Example**:

```
public class Calculator {
    public static void main(String[] args) {
        int data = 10; // Local variable stored in stack memory

        Calculator calc = new Calculator(); // Reference stored in stack, object in heap
        int result = calc.add(3, 4); // Method call, adds to stack

        System.out.println("Result: " + result);
    }

    public int add(int num1, int num2) {
        int sum = num1 + num2; // Local variables num1, num2, and sum are in stack memory
        return sum;
    }
}
```

Explanation:

- In the main method, the variable data is stored in the stack.
- When the add method is called, a new stack frame is created for this method, and local variables num1, num2, and sum are stored in this stack frame.
- The reference to the Calculator object (calc) is stored in the stack, but the actual Calculator object is stored in the heap.

Output:

Result: 7

Explanation of Output:

- The add method is called with the arguments 3 and 4. These values are stored in the stack, added together, and returned as 7. This result is then printed to the console.

2. Heap Memory

- **What is Heap Memory?**
    - Heap memory is used for dynamic memory allocation. It stores:
        - Objects
        - Classes loaded by the JVM
- **How Does Heap Memory Work?**
    - Heap memory is not bound by a specific order like the stack. Memory allocation and deallocation are dynamic, meaning objects can grow and shrink as needed.
- **Code Example**:

```
public class Calculator {
    int num=5; // Instance variable stored in heap memory

    public static void main(String[] args) {
        Calculator calc = new Calculator(); // Reference in stack, object in heap
        int result = calc.add(3, 4); // Method call
        System.out.println("Result: " + result);
    }

    public int add(int num1, int num2) {
        return num1 + num2 + num; // Accessing instance variable from heap
    }
}
```

Explanation:

- The instance variable num is stored in the heap memory because it belongs to an instance of the Calculator class.
- The reference calc is stored in the stack, pointing to the object in the heap.
- The add method uses the instance variable num, demonstrating how the stack and heap interact.

Output:

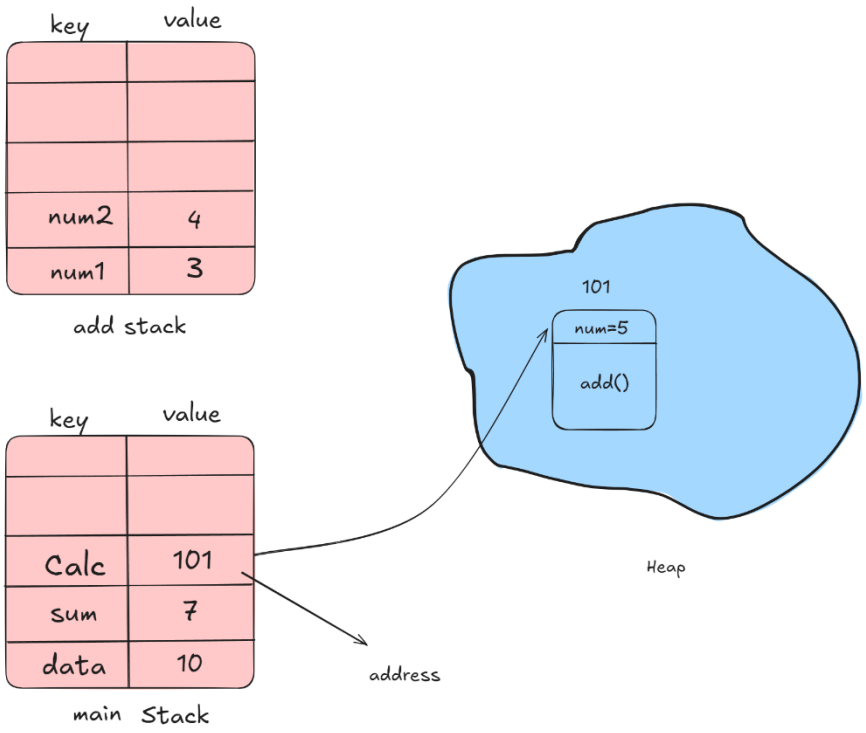
Result: 12

Explanation:

- The instance variable num is initialized to 5 in the heap memory. When add is called, num1 and num2 (with values 3 and 4) are added along with num (which is 5), giving a total of 12, which is printed.

3. Interaction Between Stack and Heap

- The stack and heap work together to manage memory efficiently. When a method uses an object, the reference is stored in the stack, and the actual object is in the heap. The JVM uses this reference to access and manipulate the object in the heap.



9.1-Need of an array

Introduction

When we need to store a single integer value, we can easily do so by using a variable. For example:

int a = 1;

This works well when dealing with a small, fixed number of values. However, what if the number of values increases over time, or if we need to store multiple values at once? Declaring a new variable for each value is not practical, as it becomes cumbersome to manage and document, especially as the number of values grows.

The Problem

Let's say you need to store multiple integer values. You could declare each value separately:

```
int a = 1;

int b = 2;

int c = 3;

// and so on...
```

This approach quickly becomes inefficient and hard to maintain. Each variable requires separate memory allocation, and managing a large number of variables can lead to errors and confusion.

The Solution: Arrays

An array is a more efficient way to handle multiple values. Instead of declaring separate variables, you can store a group of values in a single array. For example:

```
int num[] = {5, 6, 7};
```

Here, the num array can store multiple integer values in a single variable. This makes it easier to manage, access, and manipulate large sets of data.

Simple Analogy

Imagine you have guests arriving at your home. Initially, only two guests show up, and you serve them tea with the two cups you have in your hands. Later, more guests arrive, and serving each one individually becomes difficult. Instead, you use a tray that can hold several cups at once, making it easier to serve everyone. Similarly, an array acts like a tray, allowing you to store and manage multiple values efficiently.

Why Use Arrays?

- **Efficient Storage:** Arrays allow you to store multiple values in a single variable, reducing the need for multiple declarations.
- **Easy Access:** You can access any element in the array using its index, making it easier to retrieve and manipulate data.
- **Reduced Complexity:** Arrays simplify code by reducing the number of variables you need to manage.
- **Scalability:** Arrays can grow in size, making them flexible to accommodate more data as needed.

Common Use Cases

Arrays in Java are commonly used to store collections of data, such as:

- A list of numbers
- A set of strings
- A series of objects

By using arrays, you can access and manipulate collections of data more efficiently than using individual variables.

9.2-Creation of Array

Introduction

An array is a collection of elements of the same type, stored in contiguous memory locations. In Java, an array is an object that can hold multiple values of the same data type. This data structure is useful when you need to store a fixed set of elements, as it allows efficient data storage and retrieval.

Advantages of Arrays

- **Code Optimization:** Arrays help optimize code by allowing efficient data retrieval and sorting.
- **Random Access:** You can directly access any element in an array using its index, making data manipulation straightforward.

Disadvantages of Arrays

- **Size Limit:** Arrays have a fixed size; once declared, their size cannot change during runtime. To overcome this limitation, Java provides a collection framework that allows dynamic resizing of data structures.

Types of Arrays in Java

There are two main types of arrays in Java:

1. **Single-Dimensional Array:** A linear array with elements stored in a single row.
2. **Multidimensional Array:** An array of arrays, where elements are stored in a matrix form, like a table with rows and columns.

Syntax to Declare an Array in Java

You can declare an array in Java using any of the following syntaxes:

```
dataType[] arr;  // Example: int[] arr;

dataType []arr;  // Example: int []arr;

dataType arr[];  // Example: int arr[];
```

Implementation of Arrays

Example 1: Declaring and Initializing an Array

If you know the number of elements you want to store, you can directly initialize the array as follows:

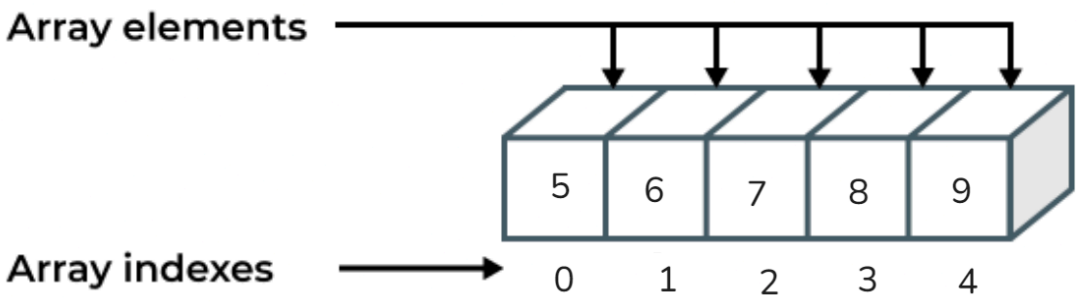
```
int num[] = {5, 6, 7, 8, 9};
```

Alternatively, you can declare the array first and then allocate memory:

```
int num[] = new int[4];  // This creates an array with 4 elements, initially set to the default value (0 for int).
```

Accessing Array Elements

In arrays, elements are stored in indexed positions, starting from 0. For example:



One-dimensional array with five elements

System.out.println(num[0]); // Output: 5

This code will print the value at index 0, which is 5. You can access other elements similarly.

Example 2: Updating an Array Element

Suppose you want to change the value at index 1 (currently 6) to 0:

num[1] = 0;

System.out.println(num[1]); // Output: 0

This updates the value at index 1 to 0 and then prints it.

Example 3: Working with Arrays of Unknown Size

If you don't know how many values you need to store initially, you can declare an array and allocate memory for a fixed number of elements:

int nums[] = new int[4]; // Allocates memory for 4 elements, all initially set to 0.

Later, you can assign values to these elements:

nums[0] = 3;

nums[1] = 4;

nums[2] = 5;

nums[3] = 6;

You can print the values individually:

System.out.println(nums[0]); // Output: 3

System.out.println(nums[1]); // Output: 4

System.out.println(nums[2]); // Output: 5

System.out.println(nums[3]); // Output: 6

Example 4: Using a Loop to Access Array Elements

Instead of manually accessing each element, you can use a loop to make the task easier, especially when dealing with larger arrays:

```
for(int i = 0; i <= 3; i++) {
    System.out.print(nums[i] + " ");  // Output: 3 4 5 6
}
```

This loop prints all the elements in the array in a single line.

9.3-Multi Dimensional Array

Understanding multi-dimensional arrays

A multi-dimensional array in Java is essentially an array of arrays. This structure allows us to store data in a matrix format, where data is organized in rows and columns. Think of it as a table where each cell can hold a value. For example, if you want to store a grid of numbers, a multi-dimensional array is the perfect choice.

Example of a Single-Dimensional Array

```
int nums[] = {3, 7, 2, 4};
```

In this example, nums is a single-dimensional array containing four elements. Now, imagine combining several such arrays. This combination leads to a multi-dimensional array, which can be visualized as a table with rows and columns.

Syntax to Declare a Multi-Dimensional Array in Java

There are several ways to declare a multi-dimensional array in Java:

```
dataType[][] arrayRefVar;     // Commonly used
dataType [][]arrayRefVar;
dataType arrayRefVar[][];
dataType []arrayRefVar[];
```

Example:

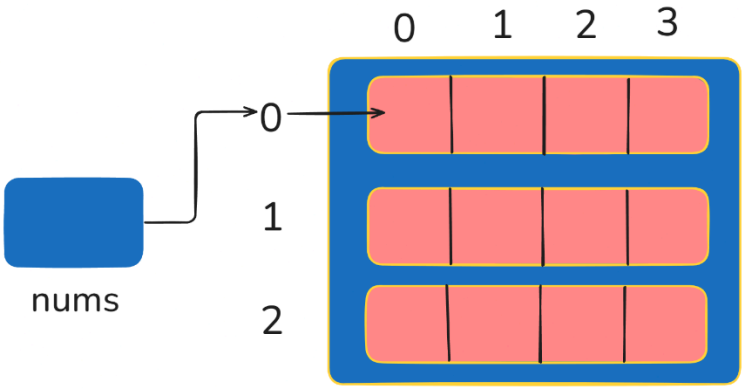
```
int nums[][] = new int[3][4];
```

This declaration creates a 2D array with 3 rows and 4 columns. Each row is an array containing 4 elements

In Java, arrays can have unequal rows, meaning not all rows need to have the same number of columns.

Understanding the Structure

In the example **int nums[][] = new int[3][4];,** the first bracket (i.e., 3) represents the number of rows, and the second bracket (i.e., 4) indicates the number of columns in each row.



.

Accessing and Iterating Over Multi-Dimensional Arrays

Using Nested Loops: To access elements in a multi-dimensional array, nested loops are often used. The outer loop iterates over the rows, while the inner loop iterates over the columns.

```
for (int i = 0; i < 3; i++) {     // Iterate over rows
    for (int j = 0; j < 4; j++) { // Iterate over columns
        System.out.print(nums[i][j] + " ");
    }
    System.out.println();         // Move to the next line after each row
}
```

This code snippet will print out the elements of the 2D array row by row. Initially, all values will be 0 since the default value of an int in Java is 0.

Generating Random Values in a 2D Array

We can populate the array with random values using the Math.random() method. Since this method returns a double, it needs to be cast to int, and the values can be scaled as needed.

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        nums[i][j] = (int) (Math.random() * 10);
        System.out.print(nums[i][j] + " ");
    }
    System.out.println();
}
```

This will fill the array with random numbers between 0 and 9 and print the array's contents.

Enhanced For Loop

You can also use an enhanced for loop to iterate over multi-dimensional arrays. However, in a 2D array, the outer loop returns entire arrays (i.e., rows), and the inner loop returns individual elements within those rows.

```
for (int[] row : nums) {
    for (int element : row) {
        System.out.print(element + " ");
    }
    System.out.println();
    }
```

This approach is cleaner and often easier to read, especially when dealing with arrays that don't require complex indexing.

9.4-Jagged and 3D Array

Jagged Arrays in Java

A **jagged array** in Java is a collection of arrays where each array can contain a different number of elements. Unlike a two-dimensional array, where every row must have the same length, a jagged array allows for rows with varying lengths. This flexibility makes jagged arrays useful for scenarios where the data structure is irregular or non-rectangular.

Jagged arrays are also known as "ragged arrays" or "irregular arrays". They can be created by specifying the number of rows in the array and then individually specifying the length of each row.

Example:

Imagine a jagged array with three rows:

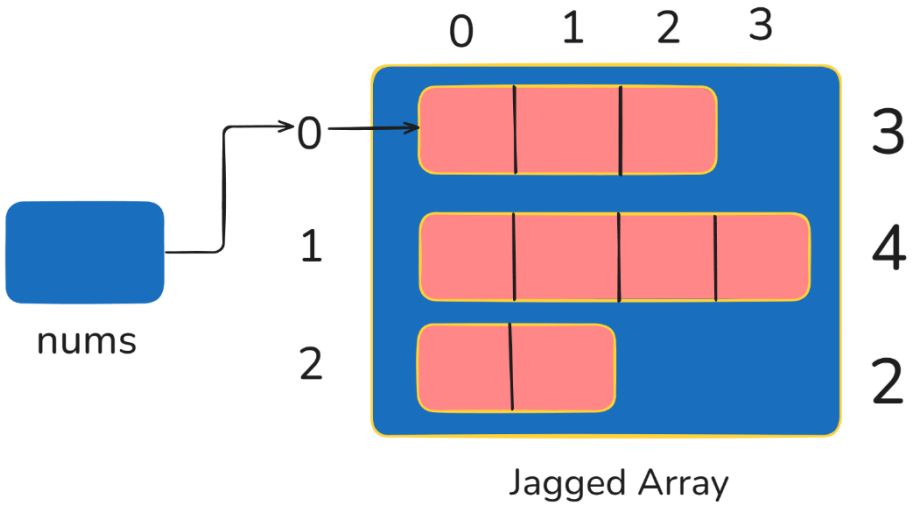
- The first row has three elements.
- The second row has two elements.
- The third row has four elements.

Syntax to Declare a Jagged Array in Java:

Here's how you can declare and initialize a jagged array in Java:

```
int nums[][] = new int[3][];  // Declaring a jagged array with 3 rows


nums[0] = new int[3];  // First row with 3 elements
nums[1] = new int[2];  // Second row with 2 elements
nums[2] = new int[4];  // Third row with 4 elements
```

In this example, the nums array has three rows, but each row has a different number of elements.



Jagged Array

Populating and Accessing Jagged Arrays:

To populate a jagged array with random values and print them, you can use nested loops:

```
for (int i = 0; i < nums.length; i++) {
    for (int j = 0; j < nums[i].length; j++) {
        nums[i][j] = (int) (Math.random() * 10);
        System.out.print(nums[i][j] + " ");
    }
    System.out.println(); } // Move to the next line after each row
```

This code will fill the jagged array with random integers between 0 and 9.

You can also use an enhanced for loop to iterate over the jagged array:

```
for (int[] row : nums) {
    for (int element : row) {
        System.out.print(element + " ");
    }
    System.out.println();
}
```

Advantages of Jagged Arrays

Jagged arrays offer several advantages over fixed-size multi-dimensional arrays:

- **Memory Efficiency:** They are more memory-efficient because memory is only allocated for the elements that are actually needed. In a fixed-size multi-dimensional array, memory is allocated for all elements, even if some remain unused.
- **Flexibility:** Jagged arrays allow each row to have a different size, which is useful for representing non-rectangular data structures.
- **Ease of Initialization:** You can easily initialize jagged arrays by specifying the size of each row individually. This makes them suitable for storing data that varies in size or is generated dynamically.
- **Enhanced Performance:** They can offer better performance in scenarios where you need to perform operations on each row independently or dynamically **add/remove** rows.
- **Ease of Manipulation:** Manipulating jagged arrays can be simpler because they allow for direct access to individual elements without complex indexing.

- **Dynamic Allocation:** They support dynamic memory allocation, enabling you to determine the size of each sub-array at runtime rather than at compile-time.
- **Space Utilization:** Jagged arrays save memory when sub-arrays are of different lengths, unlike rectangular arrays where all sub-arrays must have the same size, potentially leading to unused memory.

3D Arrays in Java

A **3D array** in Java is an extension of the two-dimensional array concept, where data is stored in multiple layers or matrices. You can think of it as a stack of 2D arrays, each representing a different layer or depth.

Example:

```
int nums[][][] = new int[3][4][5];  // Declaring a 3D array with 3 layers, 4 rows, and 5 columns
```

Here, the nums array represents a 3D structure with:

- **3 layers** (depth)
- **4 rows** per layer
- **5 columns** per row

Populating and Accessing 3D Arrays:

To populate and print a 3D array with random values:

```
for (int i = 0; i < nums.length; i++) {
    for (int j = 0; j < nums[i].length; j++) {
        for (int k = 0; k < nums[i][j].length; k++) {
            nums[i][j][k] = (int) (Math.random() * 10);
            System.out.print(nums[i][j][k] + " ");
        }
        System.out.println();
    }
    System.out.println();  // Separate layers with a blank line
}
```

Summary

Both jagged arrays and 3D arrays offer unique advantages and are suitable for different use cases. Jagged arrays provide flexibility and memory efficiency when dealing with irregular data structures, while 3D arrays are useful for representing data in multiple layers or dimensions.

9.5-Drawbacks of Array:

👉 Introduction

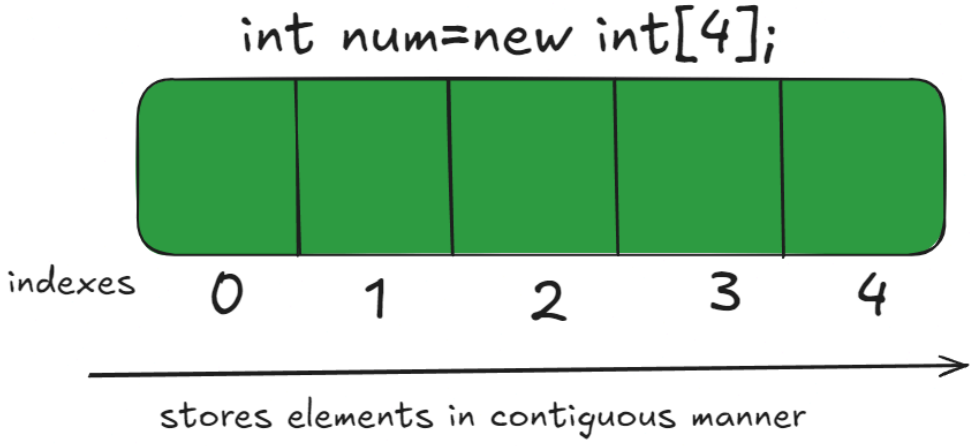
In Java, an array is used to store multiple values as a group at the same time. Arrays can be one-dimensional or multi-dimensional, such as 2D arrays or 3D arrays. Java treats arrays as objects, making them a fundamental data structure in the language. However, despite their usefulness, arrays come with several drawbacks that developers should be aware of.

👉 Memory Storage in Arrays

When we declare an array in Java, we need to specify its size. The values stored in the array are placed in contiguous memory format. This means that the elements of the array are stored sequentially in memory, which ensures faster access but comes with certain limitations.

Example:

```
public class Demo {
    public static void main(String[] args) {
        int num[] = new int[4];  // Heap memory
stores values in a contiguous manner
    }
}
```



👉 Fixed Size

Once you declare the size of an array, it becomes **fixed**. This can be limiting in the following ways:

- **Inability to Grow:** If later you decide to add more elements, you cannot expand the array. The size remains constant, and you will need to create a new array to accommodate more elements.
- **Wasted Space:** If you declare an array of size 5 but store only 4 elements, one space will remain unused, leading to wasted memory. This unused space cannot be reclaimed or resized dynamically.
- **Workaround:** A non-optimal solution to the fixed size problem is to create a new, larger array, copy the existing elements into it, and then add the new elements. However, this approach consumes more memory and may lead to inefficient performance.

👉 Inefficient Insertions and Deletions

Arrays are not ideal for scenarios where frequent insertions and deletions are needed. Here's why:

- **Insertion:** When you insert an element into an array (e.g., in the middle), all subsequent elements must be shifted by one position to make space for the new element. This shifting is time-consuming, especially in large arrays.
- **Deletion:** Similarly, when you delete an element, all subsequent elements must be shifted back to fill the gap. This makes deletions inefficient, particularly for large datasets.

For tasks that require frequent insertions and deletions, using other data structures such as ArrayList, LinkedList, or collections from the Java Collection Framework is a more efficient choice.

👉 Lack of Flexibility

Homogeneous Data:

- Arrays in Java can only store values of a single data type. This can be restrictive in cases where you need to store heterogeneous data (i.e., data of different types).

Example:

- Suppose you want to store the details of an employee, including their name (String), age (int), and phone number (long). These are different types of data, but an array can only store one type at a time (e.g., all int or all String values).
- **Alternative:** Although you can store heterogeneous data by using an array of Object, this is not an ideal solution because you lose the benefits of type safety and performance. A better approach would be to use collections such as ArrayList or HashMap, which offer more flexibility.

👉 Conclusion

While arrays are a crucial part of Java programming, it is important to understand their limitations. Arrays are useful when:

- You need a simple, fixed-size data structure.
- You require fast access to elements using an index.

However, when dealing with scenarios that require flexibility, frequent insertions, deletions, or the storage of heterogeneous data, it is better to use more versatile data structures such as ArrayList, LinkedList, or HashMap. These alternatives offer dynamic sizing, efficient memory usage, and flexibility in handling different data types.

By understanding the drawbacks of arrays, developers can make informed decisions about when to use arrays and when to switch to more advanced data structures that enhance the efficiency and scalability of their applications.

9.6-Array of Objects

Length Property of an Array

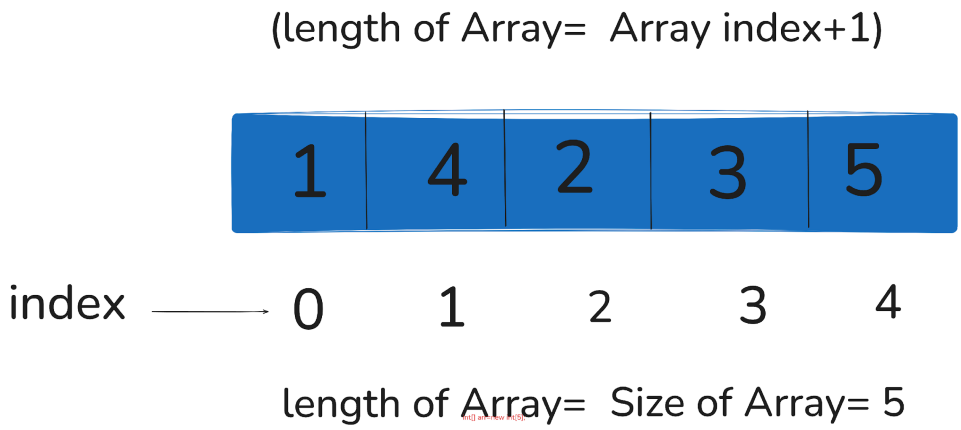
In Java, the length of an array refers to the number of elements it can hold. Unlike other data structures, Java does not provide a predefined method to obtain the length of an array. Instead, the array's length can be accessed using the length property, an attribute built into every array.

- **Understanding length:**
  - The length property gives the total number of elements that an array can contain.
  - This property is accessed using the dot . operator, followed by the array's name.

Example:

```
int[] arr = new int[5];

int arrayLength = arr.length;  // arrayLength will be 5
```

- **Logical Size vs. Array Index:**
  - The logical size usually refers to the number of elements that actually hold meaningful data, not necessarily to the highest index of the array, which is `(length - 1)`.
  - Therefore, if you want to refer to the highest index, it would be **(arrayLength – 1)**.



Example:

```
int logicalSize = arr.length - 1;  // logicalSize will be 4
```

Code Example:

```
public class MyClass {
   public static void main(String args[]) {
                int[] num = new int[6];
         num[0] = 3;
         num[1] = 4;
         num[2] = 5;
         num[3] = 8;

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

Output:

```
 3 4 5 8 0 0
```

- **Explanation:**
    - Only the first four elements of the array are assigned values.
    - The remaining two elements are automatically assigned the default value for int, which is 0.
- **Caution:**
    - When iterating through an array, it's crucial to loop only within the bounds of the array's length. Exceeding the array's length will result in an

ArrayIndexOutOfBoundsException, meaning you are trying to access an element beyond the array's limits.

- Using the length property helps ensure that your loop iterates safely within the array's bounds.

Array of Objects

Java is an object-oriented programming language, meaning that classes and objects are fundamental concepts. Typically, when we need to store a single object, we use a variable of the object's type. However, when dealing with multiple objects, an array of objects becomes more practical.

- **Definition:**
  - An array of objects stores objects as its elements, as opposed to traditional arrays, which store primitive data types like int, String, or boolean.
  - The array is created using the class name, making use of Java's Object class, the root class of all classes.
- **Syntax:**

```
Class_Name[] objectArrayReference;
```

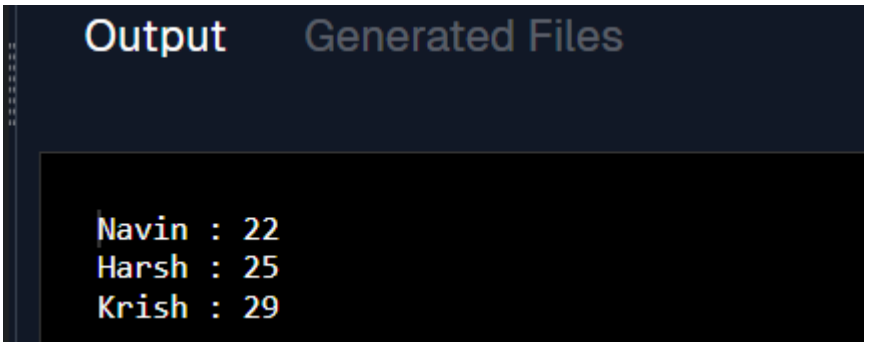
Code Example:

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


public class MyClass {
  public static void main(String args[]) {
                      Student s1 = new Student();
        Student s2 = new Student();
        Student s3 = new Student();

        s1.name="Navin";
        s2.name="Harsh";
        s3.name="Krish";

         s1.rollNo=22;
        s2.rollNo=25;
        s3.rollNo=29;

        // Array of Student objects
        Student[] students = new Student[3];
        students[0] = s1;
        students[1] = s2;
        students[2] = s3;

        for (int i = 0; i<students.length; i++) {
            System.out.println(students[i].name + " : " + students[i].rollNo);
        }
    }
}
```

Explanation:

- In this example, students is an array that can hold references to Student objects.
- We then assign references of three Student objects to the array.
- The loop accesses each student's data, such as name and rollNo, via the array.

Output:



Important Note:

The array students holds **references** to Student objects, not the actual objects themselves.

This means the array elements point to where the objects are stored in memory, rather than containing the objects directly.

9.7-Enhanced For Loop

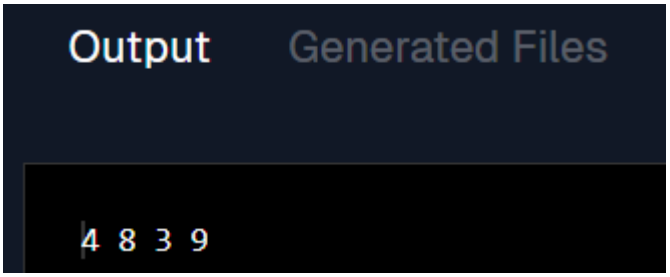
- **Introduction:**

  The enhanced for loop, also known as the **for-each loop**, was introduced in Java 1.5. It offers a more concise and readable way to iterate over arrays and collections compared to the traditional for loop. This type of loop is especially useful for traversing elements without needing to manage an index or counter.

- **Traditional For Loop Example**

```
public class MyClass {
  public static void main(String args[]) {
    int[] nums = new int[4];
nums[0] = 4;
nums[1] = 8;
nums[2] = 3;
nums[3] = 9;

for (int i = 0; i < nums.length; i++) {
System.out.print(nums[i]+" ");

}

  }
}
```

- **Output:**



- **Explanation:**

  In the traditional for loop example above, the loop uses a counter variable (i) to iterate over the array elements.

  The loop runs from i = 0 to i < nums.length and prints each element.

While this method works fine, it requires managing the index (i) and the array length, which can lead to errors like **ArrayIndexOutOfBoundsException** if the loop exceeds the array's length.

- **Enhanced For Loop**

The enhanced for loop simplifies this process. It is easier to use and reduces the risk of common programming errors.

- **Syntax:**

```
for (data_type variable : array) {
// body of for-each loop
}
```

data_type is the type of the elements in the array or collection.

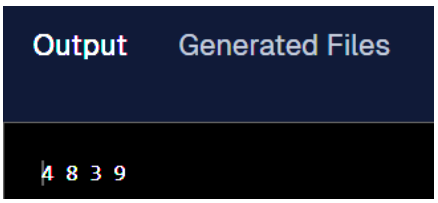
variable is the loop variable that holds the current element.

array is the array or collection you want to iterate over.

- **Enhanced For Loop Example**

```
public class MyClass {
  public static void main(String args[]) {
    int[] nums = new int[4];
nums[0] = 4;
nums[1] = 8;
nums[2] = 3;
nums[3] = 9;

for (int n : nums) {
System.out.print(n+ " ");
}


  }
}
```

- **Output:**



- **Explanation:**

In this example, the loop automatically iterates over each element in the nums array.

No need for an index variable or manually managing the loop's range.

The code is cleaner, easier to read, and less prone to errors.

- **Advantages of the Enhanced For Loop**

1. **Readability:** The enhanced for loop is more readable and concise.
2. **Error Reduction:** It reduces the chances of errors, such as going out of bounds, because there's no need to manage an index.

- **Limitations of the Enhanced For Loop**

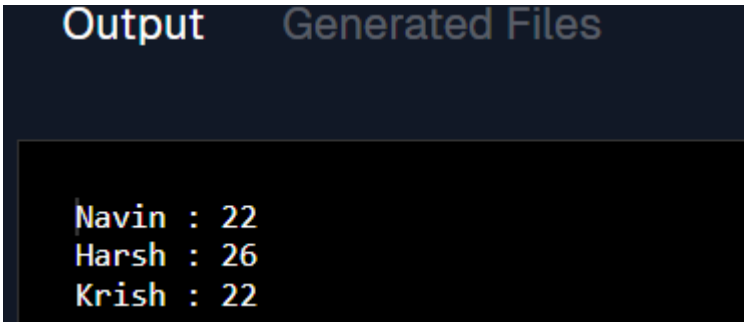
1. **No Reverse Traversal:** The enhanced for loop cannot traverse elements in reverse order.
2. **No Index-Based Access:** You cannot skip elements or access specific elements based on an index within the loop.
3. **Fixed Order:** The loop always iterates from the first element to the last.

- **Example: Enhanced For Loop with Objects**

The enhanced for loop can also be used to iterate over an array of objects.

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

public class MyClass {
  public static void main(String args[]) {
    Student s1 = new Student();
s1.name = "Navin";
s1.rollNo = 22;

Student s2 = new Student();
s2.name = "Harsh";
s2.rollNo = 26;

Student s3 = new Student();
s3.name = "Krish";
s3.rollNo = 22;

Student[] students = {s1, s2, s3};

for (Student stud : students) {
System.out.println(stud.name + " : " + stud.rollNo);
}
}
}
```

- **Output:**



- **Explanation:**

Here, we have an array of Student objects.

The enhanced for loop iterates over each Student object in the students array and prints their names and roll numbers.

This loop is efficient and requires minimal code to achieve the desired result.

08-What is String

What is a String in Java?

Overview of Strings

A **String** is generally understood as a sequence of characters. In Java, however, a string is an object that represents a sequence of characters. The ***java.lang.String*** class is used to create and manipulate string objects.

- **Data Type**: Just as there are data types for storing variables (like int, char, etc.), String is a data type used to store a sequence of characters or words within double quotes ("").
- **Character Array**: You can also use a character array to store a series of characters.

Creating a String Object

There are two main ways to create a String object in Java:

1. **By String Literal**
2. **By Using the new Keyword**

1. String Literal

- A string literal is created by placing the characters within double quotes.

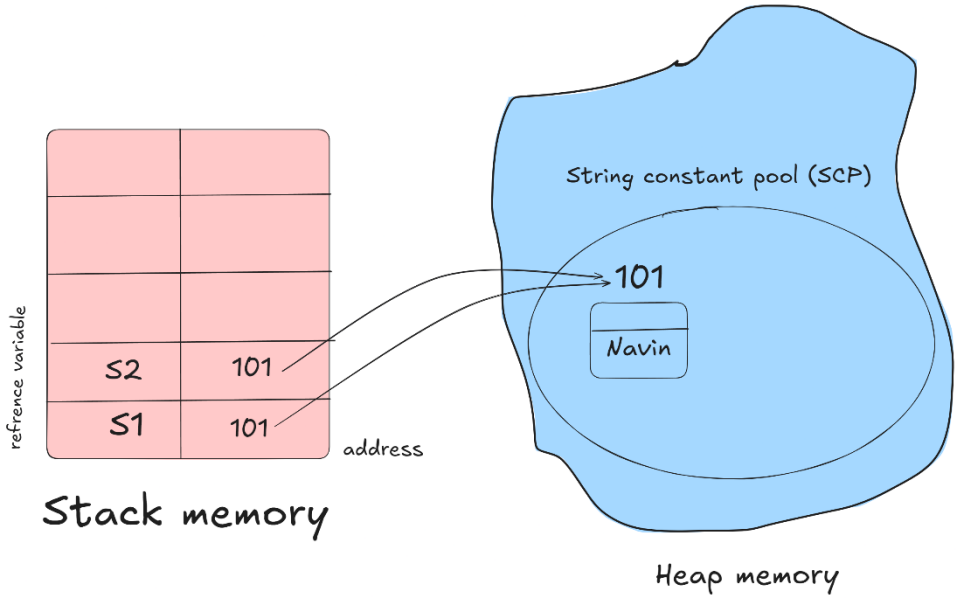
```
String s = "Navin";
```

- When a string literal is created, the JVM checks the **String Constant Pool**:
    - If the string already exists in the pool, a reference to the existing object is returned.
    - If it doesn't exist, a new string object is created and added to the pool.

Example:

```
String s1 = "Navin";

String s2 = " Navin "; // Does not create a new object, reuses the one in the pool
```

- **Explanation**: Only one object is created in this case. The JVM first checks if " Navin " is already in the pool. If it is, both s1 and s2 will reference the same object.



2. Using the new Keyword

- Strings can also be created using the new keyword:

```
String s = new String("Navin ");
```

- This approach creates two objects:
  - One in the heap memory (outside the string constant pool).
  - Another in the string constant pool.

Example:

```
String name = new String(); // Creates an empty String object
```

- **Explanation**: name is a reference variable that points to the String object. This syntax, however, is less commonly used.

Internal Architecture

- **Concatenation**: You can concatenate two strings using the + operator.

Example:

```
String name = new String("Navin");

System.out.println("Hello " + name);
```

Output:

Hello Navin

String Methods

The String class provides several methods to perform various operations on strings. Here are some of the most commonly used methods with examples:

1. **length()**: Returns the length of the string.

```
String str = "Hello";
int len = str.length(); // Returns 5
```

2. **concat(String s)**: Concatenates the specified string to the end of the current string.

```
String s1 = "Hello";
String s2 = "World";
String s3 = s1.concat(s2); // Returns "HelloWorld"
```

3. **equals(Object obj)**: Compares the string to the specified object for equality.

```
String s1 = "Hello";
String s2 = "Hello";
boolean isEqual = s1.equals(s2); // Returns true
```

4. **substring(int beginIndex)**: Returns a new string that is a substring of this string.

```
String str = "HelloWorld";
String subStr = str.substring(5); // Returns "World"
```

5. **replace(char oldChar, char newChar)**: Replaces occurrences of the old character with the new character.

```
String str = "Hello";
String replacedStr = str.replace('l', 'p'); // Returns "Heppo"
```

6. **split(String regex)**: Splits the string around matches of the given regular expression.

```
String str = "one,two,three";
String[] parts = str.split(","); // Splits into ["one", "two", "three"]
```

7. **compareTo(String anotherString)**: Compares two strings lexicographically.

```
String s1 = "Apple";
String s2 = "Banana";
int result = s1.compareTo(s2); // Returns a negative value because "Apple" is less than "Banana"
```

8. **intern()**: Returns a canonical representation for the string object.

```
String s = new String("Hello");
String internedStr = s.intern(); // Interns the string
```

09-Mutable vs Immutable string

Mutable vs. Immutable Strings in Java

Memory Allocation for Strings

In Java, memory management is crucial, especially when dealing with strings. Understanding how strings are allocated and managed in memory can help in optimizing performance and avoiding unnecessary overhead.

Example:

```
class Example {
   public static void main(String[] args) {
      String name = "navin";
      System.out.println(name + " reddy");

      String s1 = "navin";
      String s2 = "navin";
      System.out.println(s1 == s2); // Checking if both are referring to the same object or not
   }
}
```

Output:

```
navin reddy
true
```

Explanation

At first glance, you might assume that s1 and s2 refer to different objects. However, both s1 and s2 point to the same object in memory. This is due to how Java handles string literals.

Memory Allocation in JVM

Within the JVM, memory is divided into two main regions: **Heap Memory** and **Stack Memory**. When it comes to strings, there is a special area within the heap known as the **String Constant Pool (SCP)**.

- **String Constant Pool**: This pool is responsible for managing string literals. Whenever a string literal is created, the JVM checks the SCP. If the string already exists in the pool, the same object is reused, and no new memory is allocated. If the string is not found in the SCP, a new memory allocation occurs.

What is the String Pool?

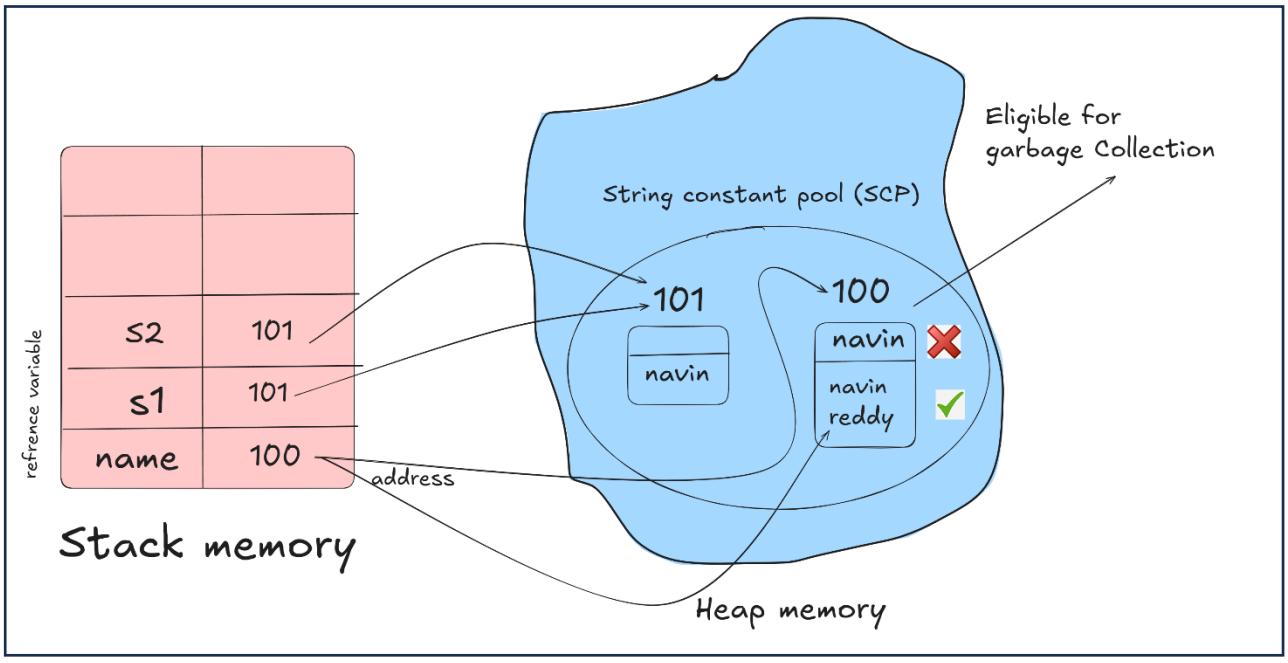
The String Pool (or String Intern Pool) is a special storage area in the Java heap where string literals are stored. It's an optimization mechanism to reduce memory usage and improve performance. By default, the String Pool is empty and is managed by the Java String class.

- **Process**:
    - When a string literal is created, the JVM checks the String Pool.
    - If the literal exists, the JVM returns a reference to the existing object.
    - If the literal doesn't exist, a new string object is created and added to the pool.

Example Breakdown

In the example provided:

- The string name is initially set to "navin". When "navin" is appended with " reddy", a new memory allocation occurs because strings in Java are immutable.
- The original "navin" object becomes eligible for garbage collection since it's no longer referenced.



This demonstrates **string immutability**—once a string object is created, its value cannot be changed. Instead, any modification creates a new string object.

Mutable Strings

For scenarios where string modification is required without creating new objects, Java provides:

- **StringBuilder**

- **StringBuffer**

Both of these classes allow for mutable strings, meaning you can modify the string without creating new objects in memory.

Example of Mutable Strings

```
StringBuilder sb = new StringBuilder("navin");

sb.append(" reddy");

System.out.println(sb.toString());
```

Output:

```
navin reddy
```

In this case, StringBuilder allows the string to be modified without creating a new object.

10-StringBuffer and StringBuilder

Java provides three classes to represent a sequence of characters: **String**, **StringBuffer**, and **StringBuilder**. The String class is immutable, while StringBuffer and StringBuilder classes are mutable.

- StringBuilder was introduced in JDK 1.5.
- StringBuffer was introduced in Java 1.0.

StringBuffer

StringBuffer represents growable and writable character sequences. Unlike String, which represents fixed-length, immutable character sequences, StringBuffer allows characters and substrings to be inserted in the middle or appended to the end. It will automatically grow to accommodate these additions and often reallocates more characters than needed to allow room for growth.

Advantages of StringBuffer:

- **Mutable and Thread-Safe:** Suitable for multi-threaded environments where string operations need to be performed safely across multiple threads.
- **Efficient for Most String Manipulations**: Reliable in scenarios where immutability is not a requirement.

Disadvantages of StringBuffer:

- **Slower Performance**: The synchronization overhead makes StringBuffer slower than StringBuilder.
- **Not Ideal for Single-Threaded Scenarios**: When immutability and maximum performance are required, StringBuffer may not be the best choice.

Example:

```
StringBuffer sb = new StringBuffer("navin");

System.out.println("Capacity: " + sb.capacity()); // Output: 21

System.out.println("Length: " + sb.length());     // Output: 5
```

```
sb.append(" reddy");

System.out.println(sb); // Output: navin reddy
```

Explanation:

- The initial capacity of StringBuffer is 16 bytes. When "navin" (5 characters) is added, the total capacity becomes 21.
- The append() method adds "reddy" to the existing string.

StringBuilder

StringBuilder is similar to StringBuffer but is not thread-safe, meaning it is not synchronized. It was introduced in JDK 1.5 and provides an efficient way to handle mutable strings in single-threaded environments.

Advantages of StringBuilder:

- **Mutable:** Allows efficient modification of string content without creating new objects.
- **High Performance:** Ideal for tasks that involve frequent and extensive string manipulations.
- **Efficient Memory Usage**: Minimizes memory overhead by reusing its internal buffer.

Disadvantages of StringBuilder:

- **Not Thread-Safe:** Requires manual handling of thread safety if used in multi-threaded environments.
- **Not Suitable for Immutable Scenarios:** If immutability is needed, StringBuilder is not the right choice.

Example:

```
StringBuilder sb = new StringBuilder("navin");

sb.append(" reddy");

System.out.println(sb); // Output: navin reddy

Comparison of StringBuffer and StringBuilder
```

Feature	StringBuffer	StringBuilder
Thread Safety	Synchronized (thread-safe)	Not synchronized (not thread-safe)
Performance	Slower due to synchronization overhead	Faster due to the absence of synchronization
Introduced in	Java 1.0	JDK 1.5

Conclusion

In conclusion, the choice between String, StringBuilder, and StringBuffer in Java depends on the specific requirements and constraints of your project. Each of these classes has its own set of advantages and disadvantages, making them suitable for different scenarios.

11-Static Variable

What is a Static Variable?

The static keyword in Java is primarily used for memory management. It can be applied to variables, methods, blocks, and nested classes. The static keyword belongs to the class itself rather than to instances of the class, meaning it's shared across all instances.

The static keyword can be used with:

1. **Variable** (also known as a class variable)
2. **Method** (also known as a class method)
3. **Block**
4. **Nested class**

Java Static Variable

When you declare a variable as static, it is known as a static variable. A static variable is used to refer to the common property of all objects of a class (which is not unique for each object), such as the company name of employees, or the college name of students.

- The static variable gets memory only once in the class area, at the time of class loading.

Advantages of Static Variables:

- **Memory Efficiency:** It saves memory because static variables are allocated memory only once, and the memory is shared across all instances of the class.

Example: Understanding Static Variables

```
class Mob {
    String brand;
    int price;
    String name;

    public void show() {
        System.out.println(brand + "," + price + "," + name);
    }
}
class Demo {
    public static void main(String[] args) {
        Mob obj = new Mob();
```

```
        Mob obj1 = new Mob();
        obj.brand = "Apple";
        obj.price = 1500;
        obj.name = "SmartPhone";

        obj1.brand = "Samsung";
        obj1.price = 1700;
        obj1.name = "SmartPhone";

        obj.show();
        obj1.show();
    }
}
```

Output:

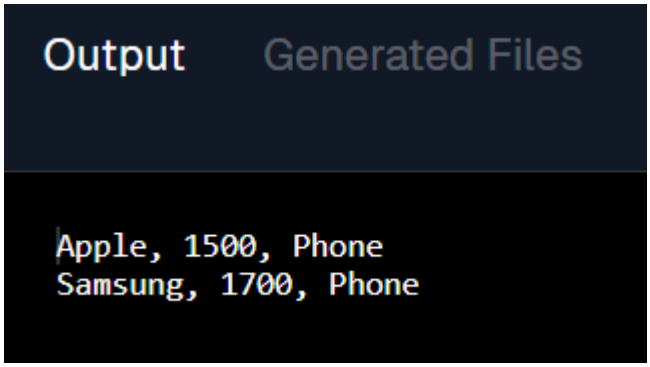
```
Apple, 1500, SmartPhone
Samsung, 1700, SmartPhone
```

In the example above, the name of the device is common for both objects. However, space is consumed each time the name is stored. Wouldn't it be better if we had a way to declare it once and have it remain common for all instances? Yes, we can achieve this by using the static keyword.

Refactoring with Static Variables

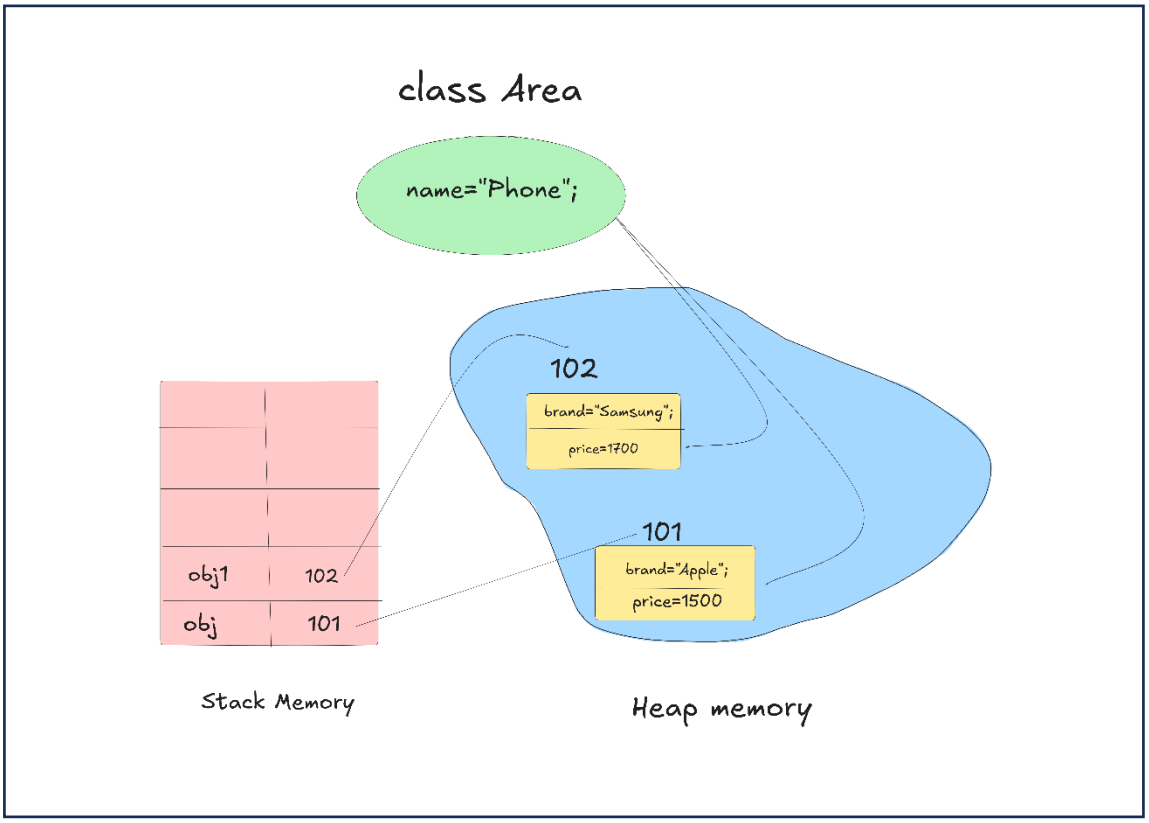
By declaring the name variable as static, it becomes a class variable, meaning memory will be allocated only once for it. When the class is loaded, memory is assigned to all static members of the class. If any variable is declared as static, its value will be shared by all objects of the class.

```java
class Mob {
    String brand;
    int price;
    String network;
    static String name;

    public void show() {
        System.out.println(brand + ", " + price + ", " + name);
    }
}

public class Demo {
    public static void main(String[] args) {
        Mob obj = new Mob();
        Mob obj1 = new Mob();

        obj.brand = "Apple";
        obj.price = 1500;
        Mob.name = "SmartPhone";

        obj1.brand = "Samsung";
        obj1.price = 1700;
        Mob.name = "SmartPhone";

        obj.name="Phone";

        obj.show();
        obj1.show();
    }
}
```



Explanation:

- The static keyword ensures that the name variable is shared across all instances of the class. Therefore, when the name is changed for one object, it reflects for all objects.
- The correct way to access static members is by using the class name, as shown with Mob.name.



13-Static Method

What is a Static Method?

In Java, the static keyword is used to create methods that belong to the class rather than to instances of the class. This means that static methods can be called without creating an object of the class. Any method declared with the static keyword is referred to as a static method.

Features of Static Methods

- **Class-Level Method**: A static method is associated with the class itself, not with any particular object instance.
- **Access by All Instances**: Every instance of a class can access a static method.
- **No Need for Object Reference**: Static methods can access static variables directly, without needing to refer to an instance of the class.
- **Limited Access**: A static method can only access other static data (static variables and static methods). It cannot access instance variables directly.
- **Direct Access**: Static methods can be accessed directly within both static and non-static methods.

Syntax to Declare a Static Method

```
Access_modifier static void methodName() {
    // Method body
}
```

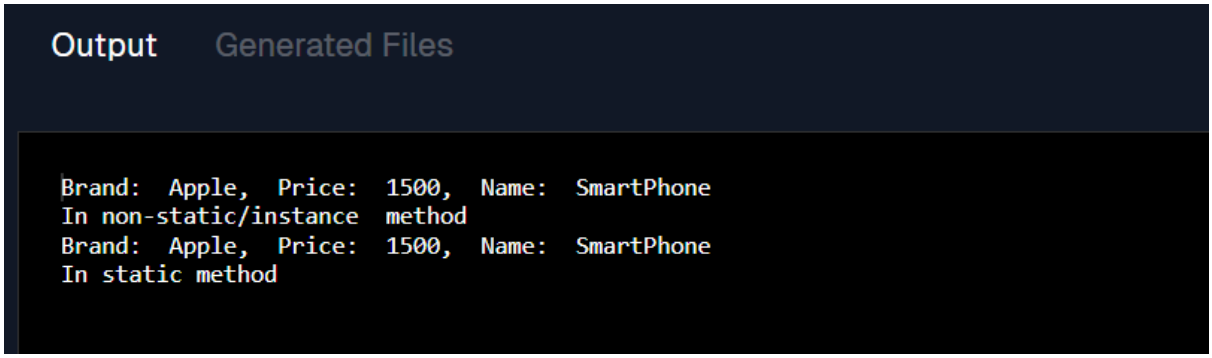
Example: Using Static Methods in a Class

```
Class Mobile
ass Mobile {
  // Instance variable
  String brand;
  int price;
  static String name;

  // Instance method
  public void show() {
      // Prints the brand, price, and name of the mobile
      System.out.println("Brand:  " + brand + ",  Price:  " + price + ",  Name:  " + name);
        System.out.println("In non-static/instance  method");
  }

  // Static method
  public static void show1(Mobile obj) {
      // Prints the brand, price, and name of the mobile
      System.out.println("Brand:  " + obj.brand + ",  Price:  " + obj.price + ",  Name:  " + name);
      // Prints a message indicating that this is a static method
      System.out.println("In static method");
  }


 Class Demo
blic class Demo {
  public static void main(String[] args) {
      // Creates a new instance of the Mobile class
      Mobile obj = new Mobile();
      // Sets the brand and price of the mobile
      obj.brand = "Apple";
      obj.price = 1500;
      // Sets the name of the mobile
      Mobile.name = "SmartPhone";

      // Calls the instance method of the Mobile class
      obj.show();
      // Calls the static method of the Mobile class with the current instance as an argument
      Mobile.show1(obj);
  }
```

Explanation of the Code

1. **Instance Method (show()):**
    - The show() method is an instance method and is called using an object (obj) of the Mobile class.
    - It can directly access instance variables (brand, price) and static variables (name).
2. **Static Method (show1()):**
    - The show1() method is declared static. It can only directly access static variables (name).
    - To access instance variables (brand, price), we pass an object of the Mobile class reference as a parameter.
    - Inside show1(), we use the passed object (obj) to access the instance variables.
3. **Main Method:**

- The main() method is also static. It is the entry point of the program and does not require an object of the class to be called.
- The main() method creates an instance of Mobile, assigns values to its fields, and calls both the instance and static methods.



Why is the Main Method in Java Static?

The main() method is static because it is the entry point for the program. When the program starts, there is no object of the class available. If the main() method were non-static, the JVM would need to create an object of the class before it could call the method, which would create a circular dependency and deadlock like situation. Making the main() method static avoids this issue and allows the program to start without needing to create an object.

FAQs

1. **Can a static method access non-static variables?**
   - No, a static method cannot directly access non-static (instance) variables. It can only access them through an object reference.
2. **Why do we need to pass an object to a static method?**
   - We pass an object to a static method to access instance variables and instance methods. Static methods can only interact with static data directly.
3. **Can we override static methods?**
   - No, static methods cannot be overridden. However, they can be hidden by declaring a static method with the same name in a subclass.
4. **What happens if we remove the static keyword from the main() method?**
   - If you remove the static keyword from the main() method, the program will not run because the JVM will not have a valid entry point to start the program.

12-Static Block

What is a Static Block?

A **static block** in Java is a block of code associated with the static keyword. This block is used for the static initialization of a class and is executed only once when the class is loaded into memory for the first time. Unlike other code blocks, a static block runs before any object of the class is created and even before the constructor is called.

Characteristics of a Static Block

- **Special Java Block:** A static block is enclosed within {} and marked with the static keyword. The code inside this block executes once and is used to initialize static variables or perform startup tasks for the class.
- **Initialization:** The primary use of a static block is to initialize static variables or execute code that must run only once when the class is loaded.

Syntax of a Static Block

```
static {
    // Initialization code or static variables
}
```

Example of a Static Block

Let's look at an example to understand how a static block works:

```java
class Mobile {
    String brand;
    int price;
    static String name;

    // Constructor to initialize instance variables
    public Mobile() {
        brand = "";
        price = 200;
        name = "Phone";

        System.out.println("Inside constructor");
    }

    // Static block to initialize static variables
    static {
        name = "Phone";
        System.out.println("In static block");
    }

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

public class Demo {
    public static void main(String[] args) {
        Mobile obj = new Mobile();
        obj.brand = "Apple";
        obj.price = 1500;
        Mobile.name = "SmartPhone";
        obj.show();
    }
}

```

Explanation of the Code

1. **Class Declaration:**
    - class Mobile { ... }: This declares a class named Mobile. Inside this class, we define variables and methods that describe the characteristics and behaviors of a mobile phone.
2. **Instance Variables:**
    - String brand; int price;: These are instance variables, meaning each object of the Mobile class will have its own copy of these variables.
    - static String name;: This is a static variable, meaning all objects of the Mobile class share the same value for this variable.
3. **Constructor:**
    - public Mobile() { ... }: This is a constructor method that initializes instance variables. The constructor is called every time a new object of the class is created.
    - brand = ""; price = 200; name = "Phone";: These lines set default values for the instance variables when a new Mobile object is created.
4. **Static Block:**
    - static { ... }: This block is executed when the class is loaded into memory, before any objects are created.
    - name = "Phone"; System.out.println("In static block");: This sets the static variable name and prints a message to the console.
5. **Instance Method:**
    - public void show() { ... }: This method displays the values of the brand, price, and name variables.
6. **Main Method:**
    - public static void main(String[] args) { ... }: This is the entry point of the program. Inside this method:
    - Mobile obj = new Mobile();: A new Mobile object is created, which calls the constructor.
    - obj.brand = "Apple"; obj.price = 1500; Mobile.name = "SmartPhone";: The brand and price instance variables are updated for the obj object, and the static name variable is set for the class.
    - obj.show();: This calls the show() method to display the object's properties.

Output Explanation

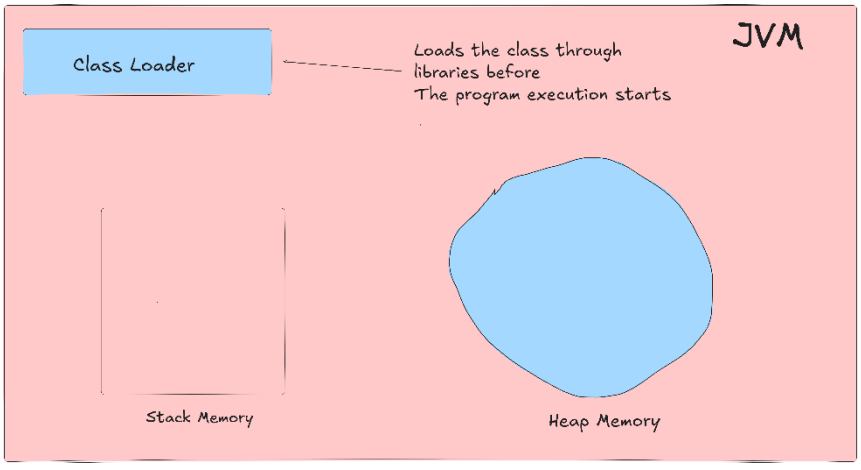
```
Output    Generated Files

In static block
Inside constructor
Brand: Apple, Price: 1500, Name: SmartPhone
```

When you run this code, you'll see the following output:

- **Static Block First:** The message "In static block" is printed first because the static block runs as soon as the class is loaded, even before the constructor.
- **Constructor Next:** The message "Inside constructor" is printed next, as the constructor is called when the object obj is created.
- **Show Method:** Finally, the show() method prints the updated values of the instance and static variables.

Understanding Static Block Execution Order

- **Static Block Before Constructor:** The static block runs before the constructor because it is associated with the class, not with individual objects. When the JVM loads the class into memory, it runs the static block to initialize any static variables.



- **Class Loading Without Object Creation:** If you want the static block to execute even without creating an object, you can force the class to load using ***Class.forName("Mobile");***. This method loads the class and triggers the static block without requiring object creation.

FAQs on Static Block

1. **Q: What is a static block in Java?**
    - A: A static block in Java is a block of code associated with the static keyword. It is used to initialize static variables or perform tasks that need to run once when the class is loaded into memory.
2. **Q: When is the static block executed?**
    - A: The static block is executed when the class is loaded into memory by the JVM, before any objects of the class are created and before the constructor is called.
3. **Q: Can a class have multiple static blocks?**
    - A: Yes, a class can have multiple static blocks. They are executed in the order they appear in the class.
4. **Q: What is the purpose of using a static block?**
    - A: The static block is primarily used to initialize static variables or execute code that should only run once when the class is loaded.
5. **Q: How can you force a static block to execute without creating an object?**
    - A: You can force the static block to execute by using Class.forName("ClassName");. This method loads the class into memory and triggers the static block.
6. **Q: Is a static block mandatory in a class?**
    - A: No, a static block is not mandatory. It is used when necessary, typically for static variable initialization.

14-Encapsulation

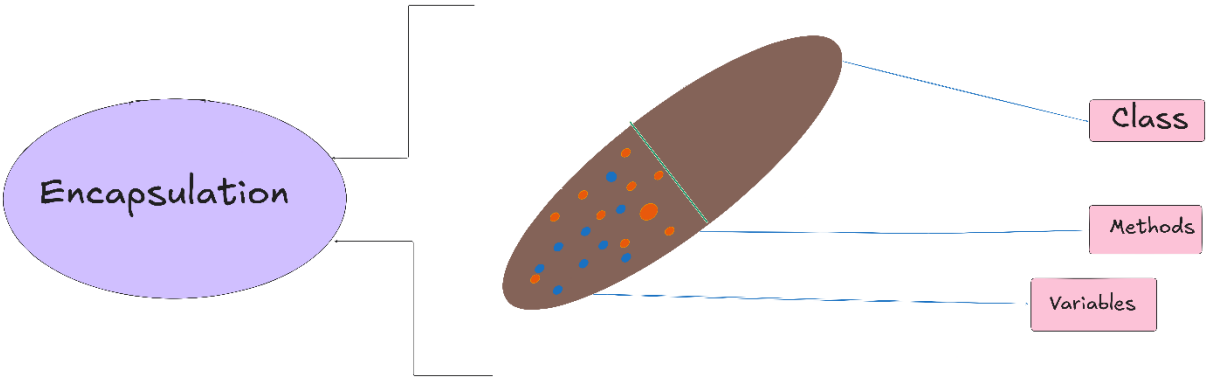
What is Encapsulation?

Encapsulation is a fundamental concept in object-oriented programming (OOP) in Java. It involves hiding the internal details of an object and exposing only the necessary parts to the outside world. This is achieved by:

- Declaring the class variables as private, which makes them accessible only within the class.
- Providing public methods (getters and setters) to access and modify these private variables.

Analogy:

Imagine a medicine capsule 💊. The internal ingredients are hidden from view. Similarly, encapsulation hides the internal state of an object and only exposes certain methods to interact with that state.



Advantages of Encapsulation

1. **Data Hiding:**
    - Hides the internal implementation details from the user.
    - Users can interact with the object through public methods without knowing the internal workings.
2. **Increased Flexibility:**
    - Variables can be made read-only or write-only depending on needs.
    - For example, omitting setter methods makes variables read-only.
3. **Reusability:**
    - Encourages code reusability and makes it easier to update and maintain.

4. **Easier Testing:**
    - Encapsulated code is easier to test and debug, especially in unit testing.
5. **Freedom for Programmers:**
    - Programmers can change the internal implementation without affecting external code, as long as the public interface remains the same.

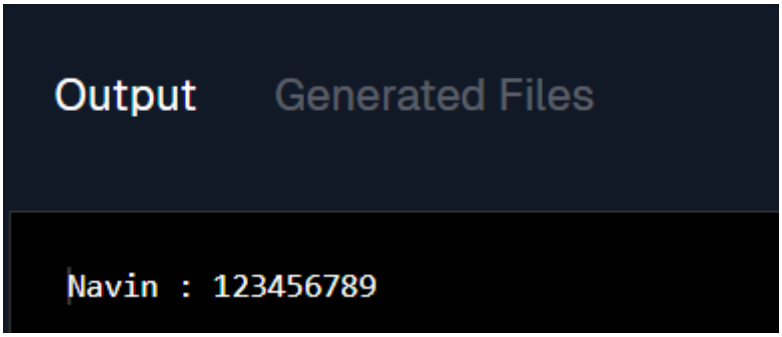
Disadvantages of Encapsulation

1. **Increased Complexity:**
    - May add complexity, especially if not implemented properly.
2. **Understanding Difficulty:**
    - Can make the system harder to understand for others who are reading or maintaining the code.
3. **Limited Flexibility:**
    - Can limit flexibility in certain scenarios where direct access to internal data might be beneficial.

Example 1: Basic Encapsulation

```java
class Human {

    private String name = "Navin";
    private long mobNo = 123456789;


    public String getName() {
        return name;
    }

    public long getMobNo() {
        return mobNo;
    }
}

public class Demo {

    public static void main(String[] args) {
        Human obj = new Human();
       System.out.println(obj.getName() + " : " + obj.getMobNo());
    }
}
```

Output:



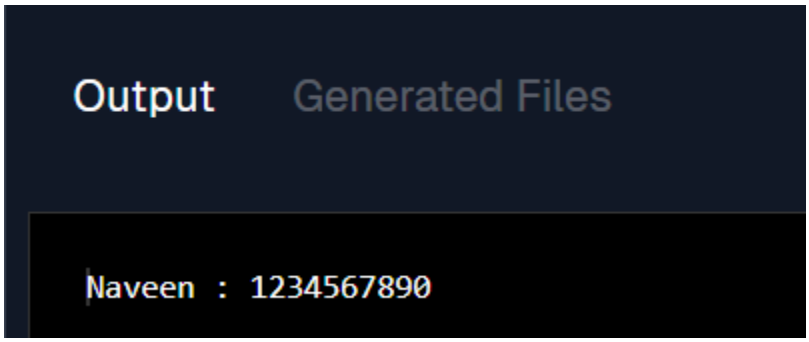
Explanation:

- **Private Variables:** name and mobNo are private, so they cannot be accessed directly from outside the Human class.
- **Getter Methods:** getName() and getMobNo() are used to retrieve these values.
- **Show Method:** Uses the getter methods to display the private data.

Example 2: Using Setters

```java
class Human {

    private String name = "Navin";
    private long mobNo = 123456789;

public void setName(String n){
name=n;
}

public void setMobNo(long mb){
mobNo=mb;
}

   public String getName() {
        return name;
    }

    public long getMobNo() {
        return mobNo;
    }
}

public class Demo {

    public static void main(String[] args) {
        Human obj = new Human();
        obj.setName("Naveen");
        obj.setMobNo(1234567890);
        System.out.println(obj.getName() + " : " + obj.getMobNo());

    }
}
```

Output:



Explanation:

- **Setter Methods:** setName() and setMobNo() allow changing the private variables.
- **Updated Values:** The main method demonstrates setting new values for name and mobNo and then retrieving and displaying these values.

Summary

Encapsulation is crucial for protecting the internal state of an object and providing a controlled way of accessing and modifying that state. By using private variables and public getter and setter methods, you can control how data is accessed and updated, which enhances the flexibility, reusability, and maintainability of your code.

7.15-Getters & Setters

In Java, **Getters** and **Setters** are special methods used to access and modify private variables within a class. Since private variables are only accessible within the class they are declared, outside classes cannot directly access them. However, by providing public getter and setter methods, you can control how these variables are accessed and modified.

Why Use Getters and Setters?

- **Encapsulation**: Getters and setters allow you to encapsulate the data within your class. This means that the internal representation of an object is hidden from the outside world, and access to this data is controlled.
- **Data Protection**: By using getters and setters, you can ensure that the data is only accessed or modified in a controlled manner, preventing unauthorized access or changes.
- **Code Maintainability**: If the internal structure of the class changes, you can modify the getter and setter methods without changing the code that uses the class.

Syntax

Both getter and setter methods start with either get or set, followed by the name of the variable with the first letter capitalized. The getter method returns the value of the variable, while the setter method assigns a new value to the variable.

```
public class Person {
    private String name; // private = restricted access

    // Getter
    public String getName() {
        return name;
    }

    // Setter
    public void setName(String newName) {
        this.name = newName;
    }
}
```

How Getters and Setters Work

- **Getter in Java**: A getter method, also known as an accessor, is used to retrieve the value of a private variable. The method returns the value of the variable, which could be of any data type like int, String, double, float, etc. The method name conventionally starts with "get" followed by the capitalized variable name.
- **Setter in Java**: A setter method, also known as a mutator, is used to update the value of a private variable. The method sets the value of the variable and conventionally starts with "set" followed by the capitalized variable name.

Example

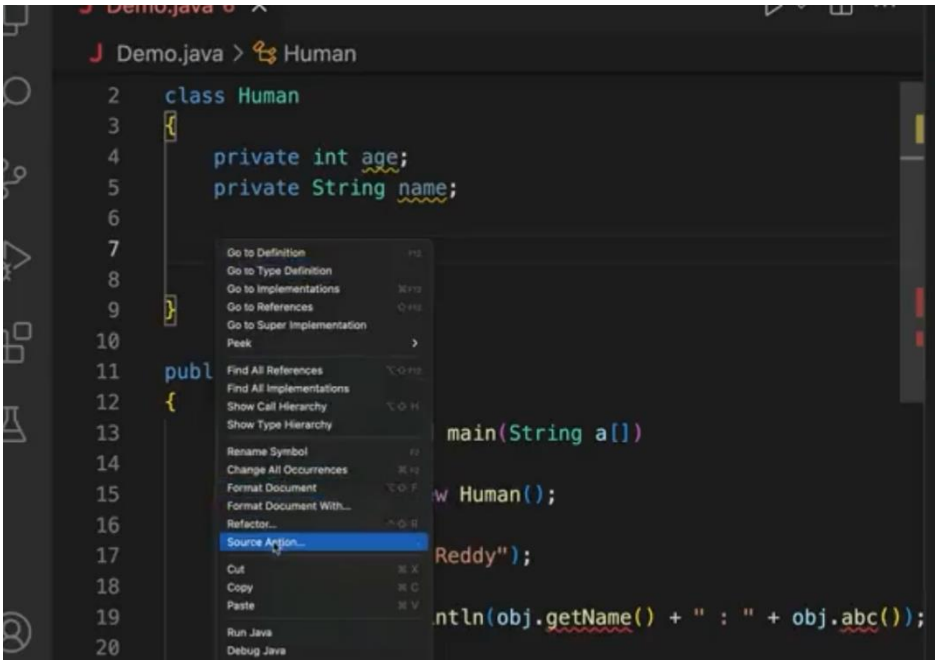
Here's an example using a Vehicle class to demonstrate getters and setters:

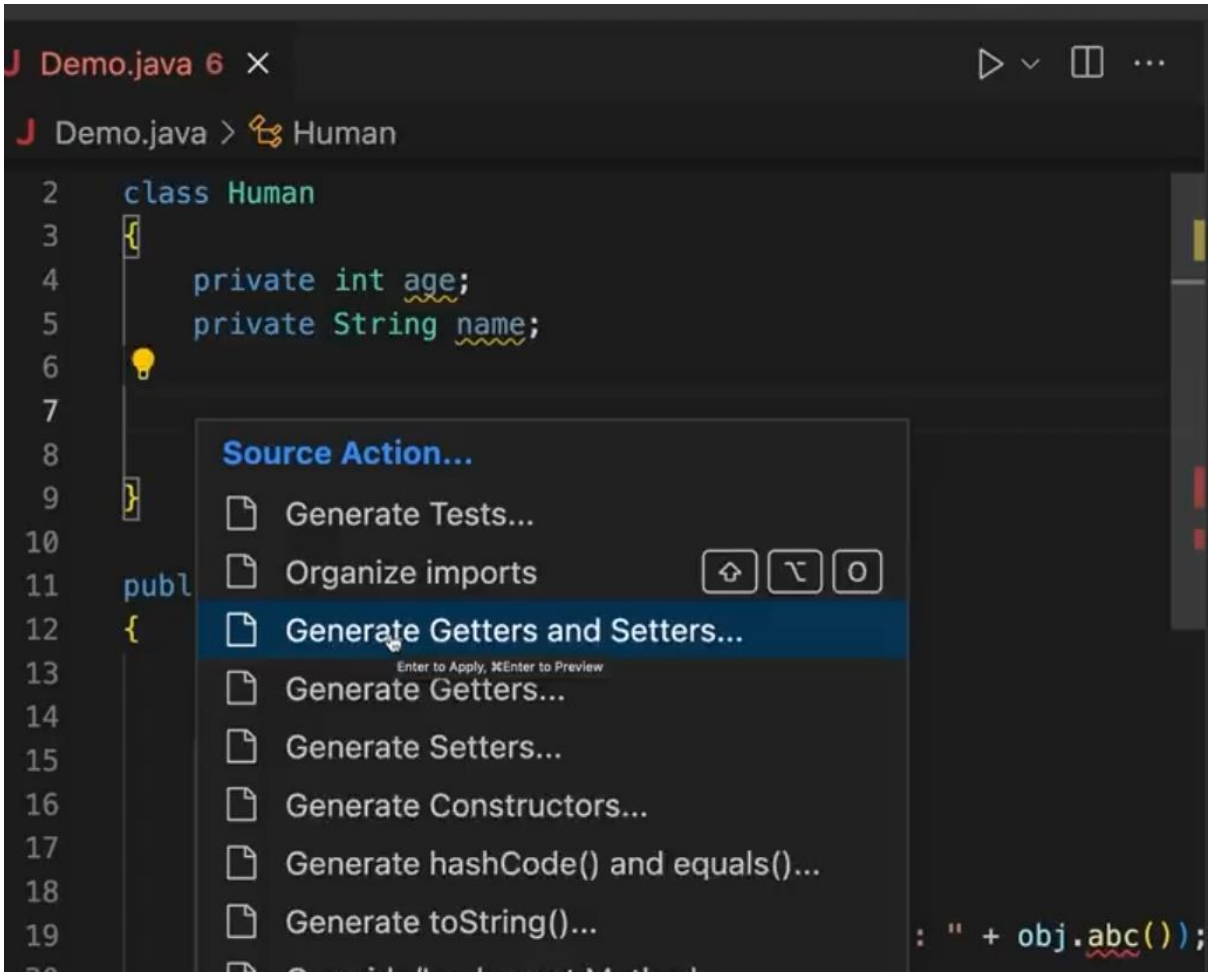
```
public class Vehicle {
    private String color;

    // Getter
    public String getColor() {
        return color;
    }

    // Setter
    public void setColor(String c) {
        this.color = c;
    }
}
```

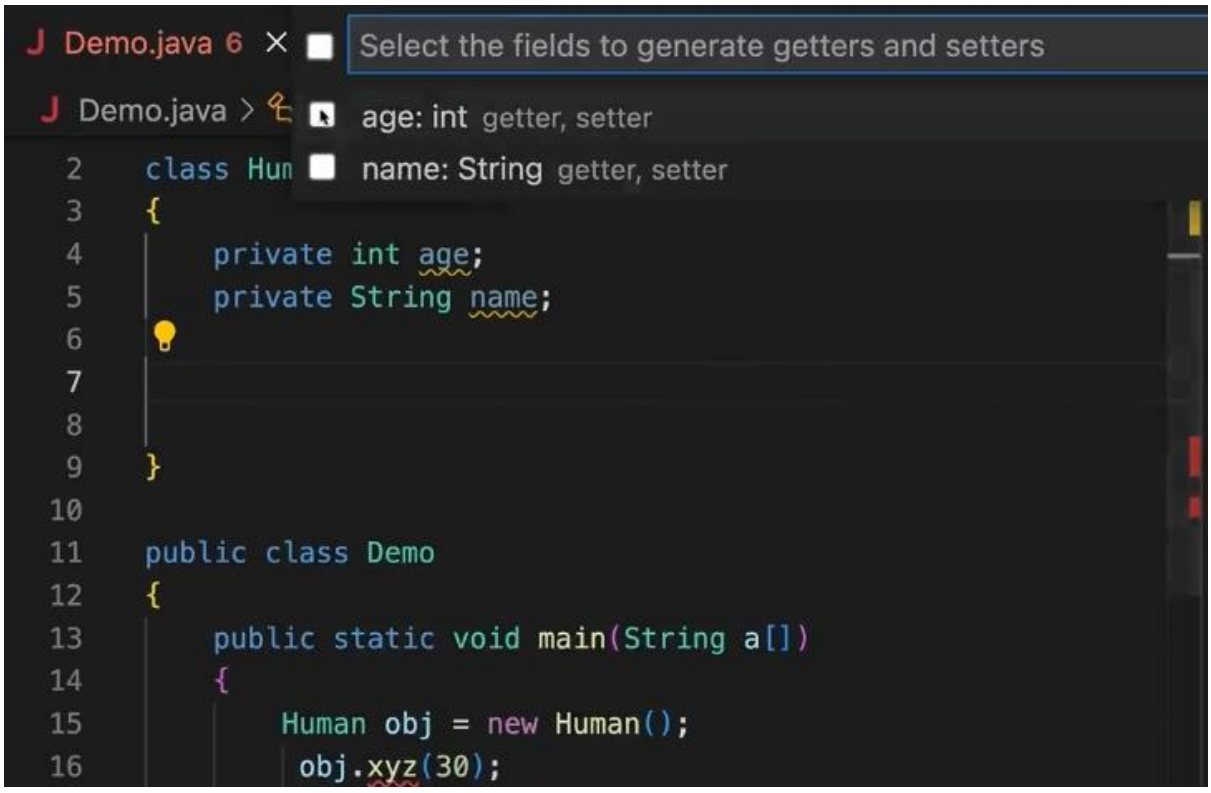
In this example:

- The getColor method returns the value of the color variable.
- The setColor method sets the value of the color variable.

Using Getters and Setters in IDEs




Demo.java 6
Demo.java > Human
class Human
{
private int age;
private String name;
Source Action...
Generate Tests...
Organize imports
Generate Getters and Setters...
Enter to Apply, ⌘Enter to Preview
Generate Getters...
Generate Setters...
Generate Constructors...
Generate hashCode() and equals()...
Generate toString()...
: " + obj.abc());


Demo.java 6
Select the fields to generate getters and setters
age: int getter, setter
name: String getter, setter
class Hun
{
private int age;
private String name;
}
public class Demo
{
public static void main(String a[])
{
Human obj = new Human();
obj.xyz(30);

TELUSKO

```java
class Human
{
    private int age;
    private String name;
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
```

Most modern Integrated Development Environments (IDEs) provide built-in support for generating getters and setters. This can save you time and ensure that the methods follow best practices. You can select the variables you want to generate methods for, and the IDE will create the code for you.

Important Considerations

- **Not Mandatory**: While it's a common practice to name these methods as get and set, it's not a strict requirement. You can name them anything, but following conventions makes the code more readable and understandable to other developers.
- **Flexibility**: Getters and setters offer flexibility in controlling how the variables are accessed and modified. For example, you can add validation logic in a setter to ensure that only valid data is assigned to a variable.

16-'this' keyword

The 'this' Keyword in Java

In Java, the '**this**' keyword is a reference variable that refers to the current object, meaning it points to the object instance on which the method or constructor is being invoked. It can be used in various ways to improve code readability and avoid naming conflicts between local variables and instance variables.

Key Uses of this Keyword:

1. **Refer to Current Class Instance Variables**:
    - When a local variable and an instance variable have the same name, this helps to differentiate between them.
2. **Invoke Current Class Methods**:
    - You can use this to call another method of the current class.
3. **Return the Current Class Instance**:
    - Useful in method chaining or builder pattern.
4. **Pass the Current Class Instance as a Parameter**:
    - You can pass the current object as an argument in a method or constructor.

Example Code

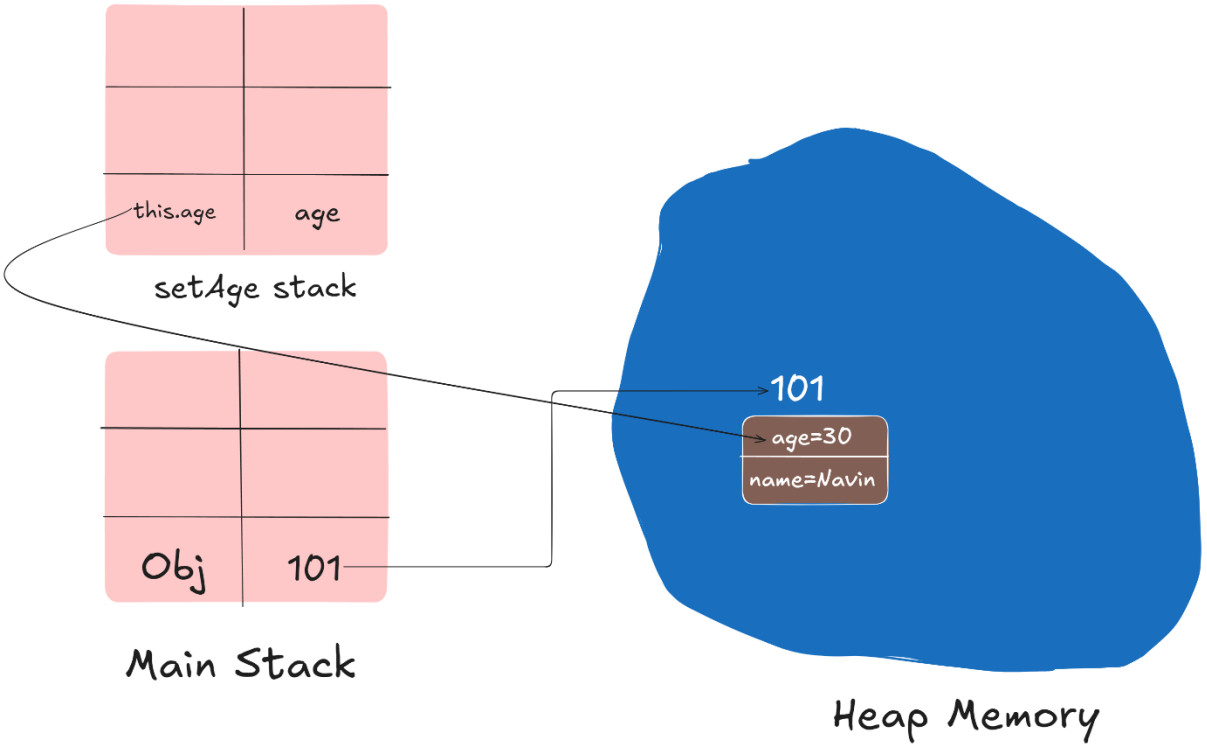
```java
class Human {
    private int age;
    private String name;

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }

    public void setAge(int age) {
        this.age = age; // 'this' refers to the current object
    }

    public void setName(String name) {
        this.name = name; // 'this' refers to the current object
    }
}

public class Demo {
    public static void main(String[] args) {
        Human obj = new Human();
        obj.setAge(30);
        obj.setName("Navin");
        System.out.println("Name: " + obj.getName() + "\nAge: " + obj.getAge());
    }
}

```

Explanation of the Code:

- **Class Human**: This class has two private instance variables, age and name. It also has getter and setter methods.
- **setAge(int age) and setName(String name) Methods**: The this keyword is used to assign the method's local variables (age and name) to the instance variables of the same name.
- **Class Demo**: This class creates an instance of the Human class, sets values using setters, and retrieves them using getters.

Output:

If we did not use this in the setter methods, the instance variables would not be correctly assigned, leading to potential bugs or unexpected results.



Methods to Use this in Java:

1. **Using this to Refer to Current Class Instance Variables**:
    - Differentiates between local and instance variables when they share the same name.
2. **Using this to Invoke Current Class Methods**:
    - Calls another method in the same class.
3. **Using this to Return the Current Class Instance**:
    - Facilitates method chaining.
4. **Using this as a Method Parameter**:
    - Passes the current object as an argument to another method or constructor.

Disadvantages of Using this:

1. **Overuse Can Reduce Code Readability**:
    - Excessive use may clutter the code and make it harder to follow.
2. **Unnecessary Overhead**:
    - Using this when not needed can add unnecessary complexity.
3. **Cannot Use this in Static Context**:
    - The '**this**' keyword cannot be used in a static method because static methods belong to the class, not any specific object.

17-Constructor

Introduction

In Java, a **constructor** is a special method used to initialize objects. The constructor is automatically called when an object of a class is created. It can be used to set initial values for object attributes, ensuring that an object is in a consistent state right from the moment it is created.

Understanding Constructors

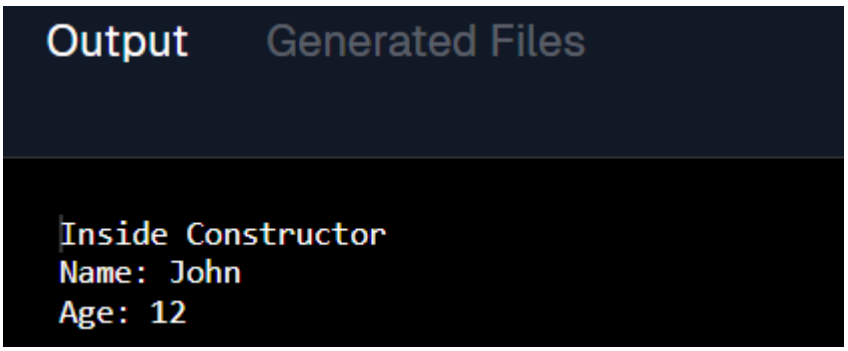
- **Constructor vs. Method**: Although a constructor is similar to a method, it differs in several ways:
    - A constructor **must** have the same name as the class.
    - Constructors **do not** have a return type, not even void.
    - A constructor is called **only once** when an object is created, whereas methods can be called multiple times.

How Constructors Work

When you create an object in Java using the new keyword, the constructor is called to initialize the object. If no constructor is explicitly defined in a class, Java provides a **default constructor** that initializes the object with default values.

```java
class Human {
    private int age;
    private String name;

    public Human() {  // Constructor
        System.out.println("Inside Constructor");
        age = 12;
        name = "John";
    }

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }
}

public class Demo {
    public static void main(String[] args) {
        Human obj = new Human();  // Constructor is called
        System.out.println("Name: " + obj.getName() + "\nAge: " + obj.getAge());
    }
}
```

In this example, when the Human object is created, the constructor is called, and the instance variables age and name are initialized with default values.



Why Use Constructors?

- **Avoid Default Values**: Without a constructor, instance variables might take default values (null for strings, 0 for integers), which may not be desired. Using a constructor, you can ensure that your object is initialized with meaningful values.
- **Custom Initialization**: Constructors allow you to set up an object exactly how you want it right when it's created.

Types of Constructors in Java

Java supports two main types of constructors:

1. **Default Constructor**: Provided automatically if no constructor is defined. Initializes object with default values.
2. **Parameterized Constructor**: Allows passing parameters to assign custom values during object creation.

3. Constructor vs. Method - A Quick Comparis

Feature	Constructor	Method
Purpose	Initializes the state of an object	Exposes the behavior of an object
Return Type	None	Must have a return type or void
Invocation	Implicitly invoked during object creation	Explicitly invoked using an object
Provided by Compiler	Yes, if no constructor is defined	No
Name	Same as the class name	Can be any valid identifier

FAQs on Java Constructors

1. **What is a constructor in Java?**
    - A constructor in Java is a special method used to initialize objects.

2. **Can a Java constructor be private?**
    - Yes, a constructor can be declared private, usually to restrict object creation from outside the class.
3. **Can a constructor be static, final, or abstract?**
    - No, a constructor cannot be static, final, or abstract because constructors are meant to initialize instances, and these modifiers conflict with the purpose of a constructor.
4. **Why can't we override constructors in Java?**
    - Constructors are not inherited, so they cannot be overridden. Overriding applies to methods that are inherited by subclasses.
5. **Is it mandatory to define a constructor in a class?**
    - No, it’s not mandatory. If you don’t define any constructor, the Java compiler provides a default constructor.

18-default vs parameterized constructor

In Java, constructors are special methods used to initialize objects. When an object of a class is created, the constructor is automatically called. Constructors can either set default values or accept parameters to set custom values for object attributes.

Types of Constructors in Java

Constructors in Java can be broadly categorized into two types:

1. **Default Constructor:**
    - A constructor that has no parameters.
    - It is automatically provided by the Java compiler if no constructor is explicitly defined.
    - **Syntax:**

<class_name>() {}

    - If a class has no constructors, the compiler automatically creates a default constructor.
2. **Parameterized Constructor:**
    - A constructor that accepts parameters to initialize an object with specific values.
    - Allows for greater flexibility in object creation.
    - **Example:**

```
public Human(int age, String name) {
    this.age = age;
    this.name = name;
}
```

Key Differences Between Default and Parameterized Constructors

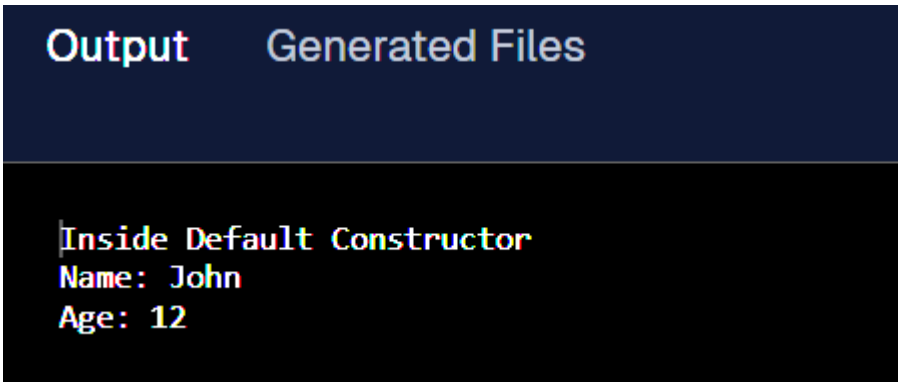
- **Default Constructor:**
    - Has no parameters.
    - Provides default values to object attributes.
    - Automatically generated by the compiler if not explicitly defined.
- **Parameterized Constructor:**
    - Accepts arguments to initialize object attributes with specific values.

- Must be explicitly defined by the programmer.

Example: Default Constructor

```java
class Human {
    private int age;
    private String name;

    public Human() {  // Default Constructor
        System.out.println("Inside Default Constructor");
        age = 12;
        name = "John";
    }

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }
}

public class Demo {
    public static void main(String[] args) {
        Human obj = new Human();  // Default Constructor is called
        System.out.println("Name: " + obj.getName() + "\nAge: " + obj.getAge());
    }
}
```

Output:



In this example, every time an object of the Human class is created using the default constructor, the same default values (age = 12, name = "John") are assigned to the object's attributes.

Example: Parameterized Constructor

```java
class Human {
    private int age;
    private String name;

    public Human() {  // Default Constructor
        System.out.println("Inside Default Constructor");
        age = 12;
        name = "John";
    }

    public Human(int a, String n) {  // Parameterized Constructor
        System.out.println("Inside Parameterized Constructor");
        age = a;
        name = n;
    }

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }
}

public class Demo {
    public static void main(String[] args) {
        Human obj = new Human();  // Default Constructor is called
        Human obj1 = new Human(18, "June");  // Parameterized Constructor is called
        System.out.println("Name: " + obj.getName() + "\nAge: " + obj.getAge());
        System.out.println("Name: " + obj1.getName() + "\nAge: " + obj1.getAge());
    }
}
```

Output:

```
Output    Generated Files

Inside Default Constructor
Inside Parameterized Constructor
Name: John
Age: 12
Name: June
Age: 18
```

Here, the default constructor assigns values for obj, while the parameterized constructor allows different values to be assigned to obj1.

Additional FAQs on Java Constructors

1. **What happens if I define a constructor with parameters and no default constructor?**
    - If you define a parameterized constructor and omit the default constructor, the compiler will not provide a default constructor. Therefore, attempting to create an object without arguments will result in a compile-time error.
2. **Can a constructor be private?**
    - Yes, a constructor can be private. This is often used in the Singleton design pattern to restrict object creation and ensure only one instance of a class exists.
3. **Can constructors be overloaded in Java?**
    - Yes, constructors can be overloaded by defining multiple constructors with different parameter lists within the same class.
4. **Why do we need a parameterized constructor?**
    - A parameterized constructor is useful when you want to create objects with different initial values, providing more control over object creation.

8.1-Naming Convention

In Java, it's considered acceptable practice to name classes, variables, and methods in a way that clearly describes their purpose. This approach, rather than using arbitrary names, helps improve the readability and maintainability of code. When naming these elements, Java developers typically follow CamelCase conventions, which involve capitalizing the first letter of each subsequent word in a compound word.

Advantages of Naming Conventions in Java

- Readability: Well-named identifiers make the code easier to read and understand for both the author and others who may use it later.
- Maintenance: Clear naming reduces the time required to understand the code, making it easier to update and maintain.

CamelCase Naming Conventions

- Classes and Interfaces: The first letter is uppercase. For example: Thread, Runnable, Reflection, Collections.
- Methods: Start with a lowercase letter, with subsequent words capitalized. For example: concat(), toString(), wait(), notify().
- Constants: Written in all uppercase letters, often with words separated by underscores. For example: PIE, SUNDAY, MAX_VALUE.

Popular Naming Conventions

Identifier Type	Naming Rules	Examples
Class	- Start with an uppercase letter. - Should be a noun. - Use meaningful words and avoid acronyms.	`public class Employee { }`
Interface	- Start with an uppercase letter. - Should be an adjective. - Avoid acronyms.	`interface Printable { }`
Method	- Start with a lowercase letter. - Should be a verb. - Use camelCase for multi-word names.	`void draw() { }`
Variable	- Start with a lowercase letter. - Avoid special characters and one-character names. - Use camelCase for multi-word names.	`int id;`
Package	- Use all lowercase letters. - Separate multi-word names with dots.	`package com.telusko;`
Constant	- Use all uppercase letters. - Separate multi-word names with underscores. - May contain digits but not as the first character.	`static final int MIN_AGE = 18;`

19-Anonymous Object

Anonymous Object in Java

Object Creation: The Traditional Way

In Java, the traditional way of creating an object involves two main steps:

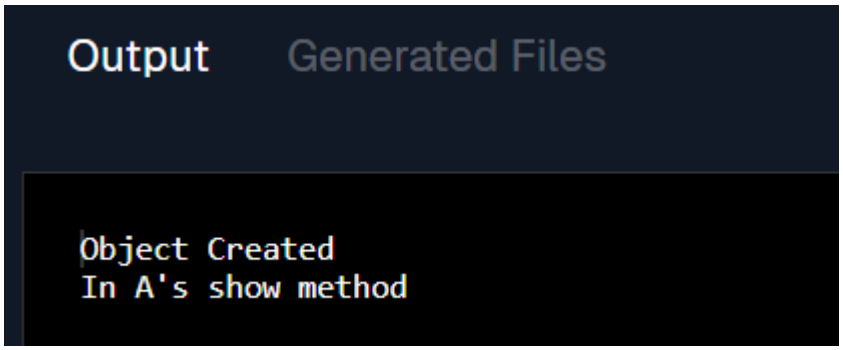
1. **Declaration of an Object Reference**: First, you declare a reference variable that will point to the object.
2. **Object Creation**: Then, you use the **new keyword** to create the object in the heap memory and assign it to the reference variable.

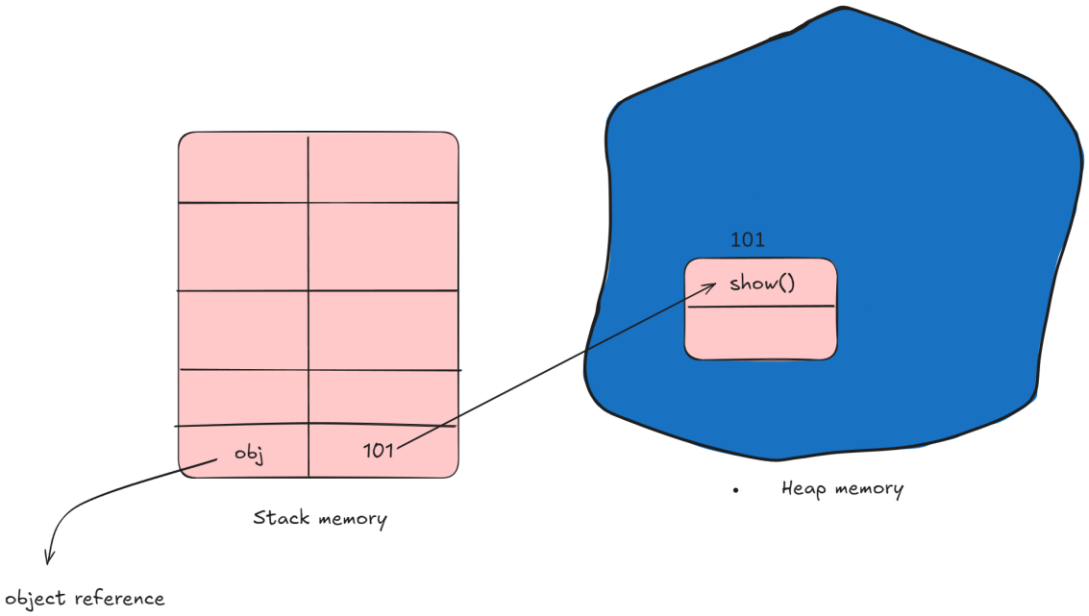
This approach involves allocating memory in both the stack (for the reference variable) and the heap (for the actual object).

Example:

```
1  class A {
2      public A() {
3          System.out.println("Object Created");
4      }
5
6      public void show() {
7          System.out.println("In A's show method");
8      }
9  }
10
11 public class Demo {
12     public static void main(String[] args) {
13         A obj = new A(); // Traditional object creation
14         obj.show();
15     }
16 }
```

Output:





Anonymous Object in Java

But what if you want to create an object without allocating space in the stack for the reference variable? Can you directly use an object without assigning it a name or reference?

Yes, you can create such an object, known as an **Anonymous Object**. An anonymous object is created without a reference variable and can still be used to invoke methods.

- In Java, an anonymous object is an object that is created without giving it a name. Anonymous objects are often used to create objects on the fly and pass them as arguments to methods.

Example:

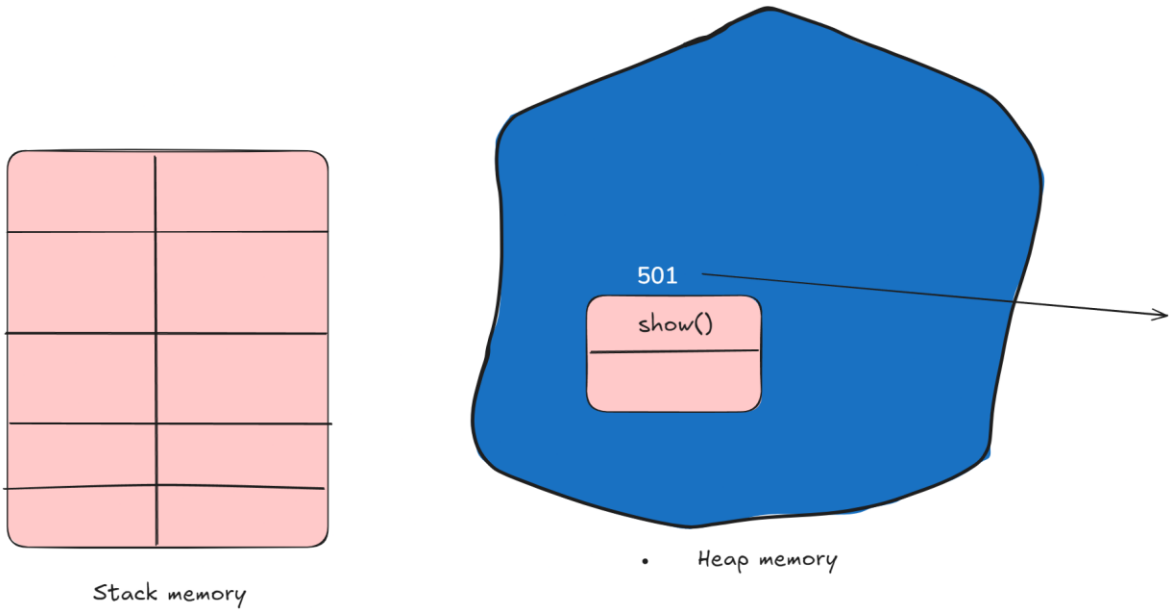
```
class A {
    public A() {
        System.out.println("Object Created");
    }

    public void show() {
        System.out.println("In A's show method");
    }
}

public class Demo {
    public static void main(String[] args) {
        new A().show(); // Anonymous object creation
    }
}
```

Output:

```
Output      Generated Files

Object Created
In A's show method
```

Explanation of the Example

In the second example, the object is created without assigning it to any reference variable. The new A().show(); statement creates the object of class A, immediately calls its show() method, and then the object becomes eligible for garbage collection once the method execution is complete. This type of object is useful when you need to create an object for one-time use, where retaining the object reference isn't necessary.



Advantages of Using Anonymous Objects

- **Memory Efficiency**: Since no reference is stored in the stack, anonymous objects save memory.
- **Cleaner Code**: Ideal for situations where the object is only used once, resulting in less clutter in the code.

Disadvantages of Using Anonymous Objects

- **No Reusability**: Since the object has no reference, it can't be reused in the program.
- **Immediate Garbage Collection**: The object is eligible for garbage collection right after its usage, which may not be desirable in some cases.

20-Need For Inheritance

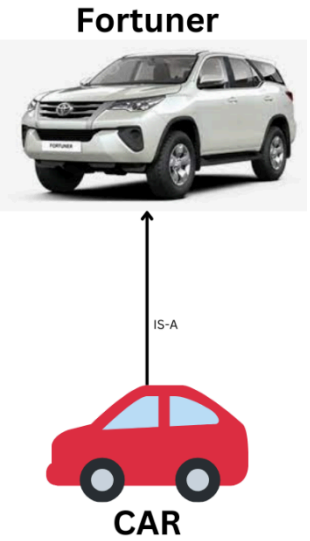
Inheritance in Java

Object-Oriented Programming (OOP):
When discussing Object-Oriented Programming (OOP), concepts like classes, objects, and encapsulation are fundamental. One of the most important concepts within OOP is **inheritance**.

The Need for Inheritance

Understanding "Is-A" and "Has-A" Relationships:

- **Has-A Relationship:**
  Consider a computer as an abstract concept. It could be a laptop, desktop, or mobile device—it's not specific, just a hardware device. When discussing a desktop, for example, we might say that the desktop *has* hardware, *has* a keyboard, and *has* a mouse. These relationships describe composition, where an object contains other objects.
- **Is-A Relationship:**
  Now, take the example of a Toyota Fortuner, a popular car in India. When referring to it, we say "Fortuner *is a* Car." The phrase "is a" emphasizes the relationship of inheritance. Here, the Fortuner inherits the general characteristics of a car but also includes specific features that distinguish it from other cars.



Real-World Example of Inheritance

In Java, inheritance is a mechanism where one class can inherit properties and methods from another class. Let's consider the example of a calculator:

- **Basic Calculator (Calc):**
  This class might include variables and methods for basic operations like addition, subtraction, and division.
- **Advanced Calculator (AdvCalc):**
  This class represents a scientific calculator with advanced features, in addition to the basic operations.

Inheritance Relationship:

- AdvCalc ***inherits*** from Calc. This means AdvCalc can use the properties and methods of Calc and add its own specialized features.

Key Terms in Inheritance

- **Parent Class / Superclass / Base Class:**
  The class from which properties and methods are inherited. In this example, Calc is the parent class.
- **Child Class / Subclass / Derived Class:**
  The class that inherits from another class. Here, AdvCalc is the child class.

- ## Inheritance and the extends Keyword in Java
- In Java, inheritance is implemented using the `extends` keyword. This keyword allows one class (the child class or subclass) to inherit the properties and methods of another class (the parent class or superclass).

- **Syntax:**

```
class ParentClass {
    // Fields and methods
}

class ChildClass extends ParentClass {
    // Fields and methods specific to ChildClass
}
```

21-What is Inheritance

What is Inheritance?

Inheritance in Java is a mechanism by which one class (called a child or subclass) can inherit the properties and behaviours (fields and methods) of another class (called a parent or superclass). This is a fundamental concept in Object-Oriented Programming (OOP), allowing for the creation of new classes based on existing ones. It promotes reusability and can help reduce redundancy in your code.

By using inheritance, you can:

- Reuse existing code from parent classes.
- Add or modify functionalities in child classes.
- Maintain a clear and organized structure in your codebase.

Key Concepts in Inheritance

1. **Class**:
    - A blueprint from which individual objects are created. It defines a group of objects with common properties.
2. **Subclass/Child Class**:
    - A class that inherits the properties and methods from another class. It is also known as a derived class, extended class, or child class.
3. **Superclass/Parent Class**:
    - The class from which a subclass inherits properties and methods. It is also referred to as a base class or parent class.
4. **Reusability**:
    - The concept of reusing fields and methods of an existing class when creating a new class. This is one of the primary benefits of inheritance, allowing for more efficient and maintainable code.

Syntax of Java Inheritance

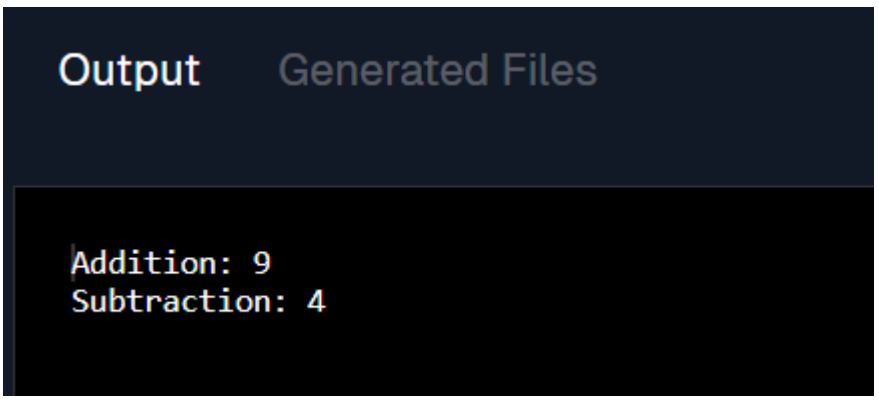
```
class SubclassName extends SuperclassName {
    // fields and methods
}
```

Example: Simple Calculator

Let's start with a simple calculator class that performs basic arithmetic operations like addition and subtraction:

```
class Calc {
    public int add(int n1, int n2) {
        return n1 + n2;
    }

    public int sub(int n1, int n2) {
        return n1 - n2;
    }
}

public class Demo {
    public static void main(String[] args) {
        Calc obj = new Calc();
        int r1 = obj.add(4, 5);
        int r2 = obj.sub(7, 3);
        System.out.println("Addition: " + r1 + "\nSubtraction: " + r2);
    }
}
```

Output:



Explanation:

This example demonstrates a simple calculator that performs addition and subtraction. The Calc class contains two methods: add and sub. The Demo class creates an instance of Calc and uses it to perform the operations.

Extending the Calculator: Advanced Calculator

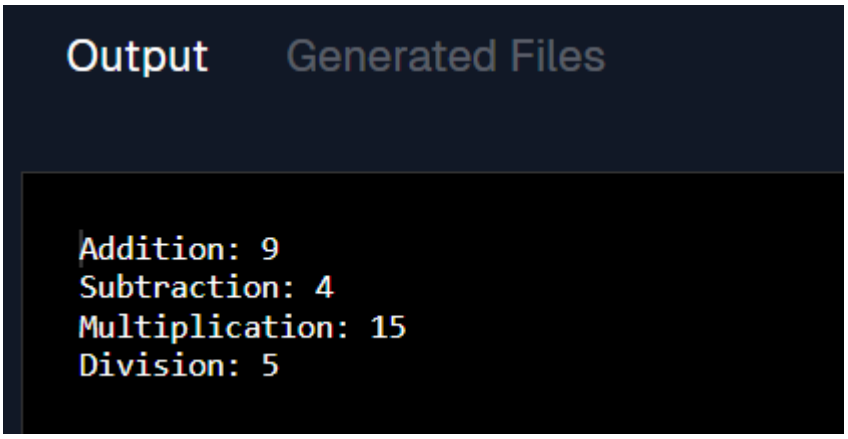
Suppose you need a calculator with advanced features like multiplication and division. You have two options:

1. Add more methods to the Calc class.
2. Create a new class (AdvCalc) that extends Calc and adds the extra features.

Option 2 is more feasible as it avoids code redundancy and keeps the code organized.

```
class Calc {
    public int add(int n1, int n2) {
        return n1 + n2;
    }

    public int sub(int n1, int n2) {
        return n1 - n2;
    }
}
class AdvCalc extends Calc {
    public int mul(int n1, int n2) {
        return n1 * n2;
    }

    public int div(int n1, int n2) {
        return n1 / n2;
    }
}

public class Demo {
    public static void main(String[] args) {
        AdvCalc obj = new AdvCalc();
        int r1 = obj.add(4, 5);  // Inherited method
        int r2 = obj.sub(7, 3);  // Inherited method
        int r3 = obj.mul(5, 3);  // New method
        int r4 = obj.div(15, 3); // New method

        System.out.println("Addition: " + r1 + "\nSubtraction: " + r2 +
                           "\nMultiplication: " + r3 + "\nDivision: " + r4);
    }
}

```

Output:



Explanation:

In this example, the AdvCalc class extends the Calc class. It inherits the add and sub methods and introduces two new methods: mul (for multiplication) and div (for division). This demonstrates the power of inheritance, where the AdvCalc class can reuse the existing methods from Calc while adding its own functionalities.

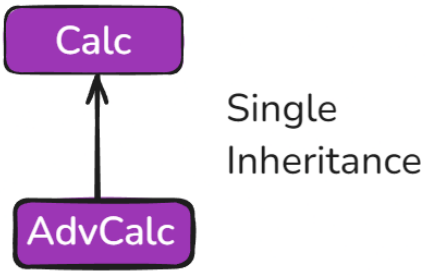
22-SIngle and Multilevel inheritance

Single and Multilevel Inheritance in Java

Inheritance is a powerful concept in Java that allows one class to inherit the properties and behaviors of another class. This concept is fundamental in Object-Oriented Programming (OOP) and helps in creating a hierarchical structure for classes.

Single Inheritance

Definition: Single inheritance occurs when a class (child class) inherits from only one parent class. This allows the child class to access the fields and methods of the parent class.



Example: Let's look at a simple example of single inheritance:

```java
class Calc {
    public int add(int n1, int n2) {
        return n1 + n2;
    }

    public int sub(int n1, int n2) {
        return n1 - n2;
    }
}

class AdvCalc extends Calc {
    public int multi(int n1, int n2) {
        return n1 * n2;
    }

    public int div(int n1, int n2) {
        return n1 / n2;
    }
}

public class Demo {
    public static void main(String[] args) {
        AdvCalc obj = new AdvCalc();
        int r1 = obj.add(4, 5);
        int r2 = obj.sub(7, 3);
        int r3 = obj.multi(5, 3);
        int r4 = obj.div(15, 4);

        System.out.println("Addition: " + r1);
        System.out.println("Subtraction: " + r2);
        System.out.println("Multiplication: " + r3);
        System.out.println("Division: " + r4);
    }
}
```

Output:

```
Output    Generated Files

Addition: 9
Subtraction: 4
Multiplication: 15
Division: 3
```

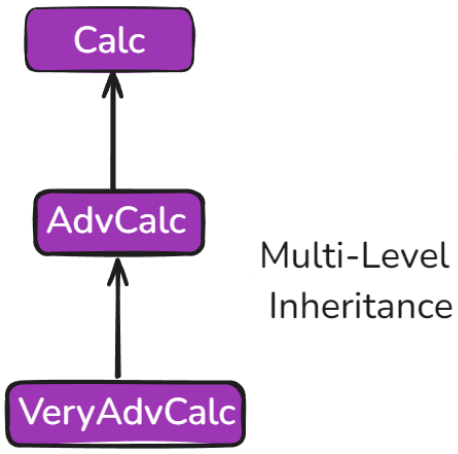
Explanation:

- This example demonstrates how AdvCalc, the child class, inherits basic arithmetic operations from Calc, the parent class.
- AdvCalc adds its own methods (multi and div) while reusing the methods from Calc (add and sub).

This approach prevents code duplication and makes the code easier to maintain and extend.

Multilevel Inheritance

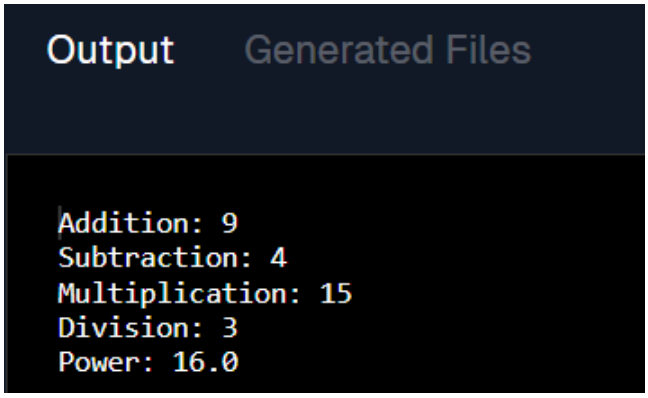
Definition: Multilevel inheritance is a scenario where a class inherits from a child class, making a chain of inheritance. This allows a subclass to inherit properties and methods from more than one level of parent classes.



Example: Consider the following example to understand multilevel inheritance:

```java
class Calc {
    public int add(int n1, int n2) {
        return n1 + n2;
    }

    public int sub(int n1, int n2) {
        return n1 - n2;
    }
}

class AdvCalc extends Calc {
    public int multi(int n1, int n2) {
        return n1 * n2;
    }

    public int div(int n1, int n2) {
        return n1 / n2;
    }
}

class VeryAdvCalc extends AdvCalc {
    public double power(int n1, int n2) {
        return Math.pow(n1, n2);
    }
}

public class Demo {
    public static void main(String[] args) {
        VeryAdvCalc obj = new VeryAdvCalc();
        int r1 = obj.add(4, 5);
        int r2 = obj.sub(7, 3);
        int r3 = obj.multi(5, 3);
        int r4 = obj.div(15, 4);
        double r5 = obj.power(4, 2);

        System.out.println("Addition: " + r1);
        System.out.println("Subtraction: " + r2);
        System.out.println("Multiplication: " + r3);
        System.out.println("Division: " + r4);
        System.out.println("Power: " + r5);
    }
```

Output:



Explanation:

- In this example, VeryAdvCalc inherits from AdvCalc, which in turn inherits from Calc. This creates a multilevel inheritance structure.

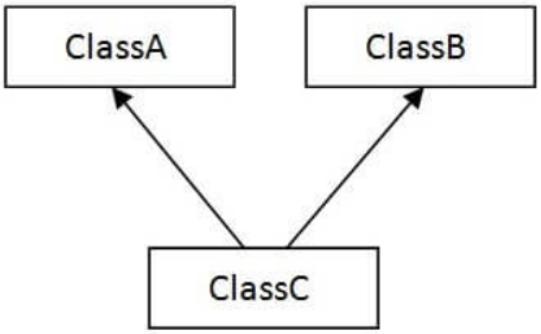
- VeryAdvCalc has access to methods from both its parent (AdvCalc) and grandparent (Calc) classes, along with its own method power.

23-Multiple Inheritance

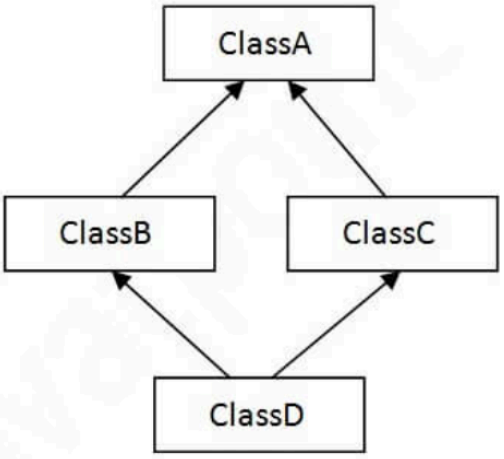
Introduction: In object-oriented programming, inheritance allows a class to inherit properties and methods from another class. However, Java does not support multiple inheritance for classes, meaning a class cannot inherit from more than one class. This is a key difference from other languages like C++, where multiple inheritance is allowed.

Why Multiple Inheritance is Not Allowed in Java:

Consider a scenario where class C inherits from two classes, A and B. If both A and B contain a method with the same name, it creates ambiguity: the compiler cannot determine whether to use the method from A or B. This issue is known as the **Diamond Problem** or **Ambiguity Problem**.



To



avoid this problem and reduce complexity, Java does not support multiple inheritance for classes. Instead, Java promotes code simplicity and consistency by disallowing it, resulting in compile-time errors rather than runtime issues.

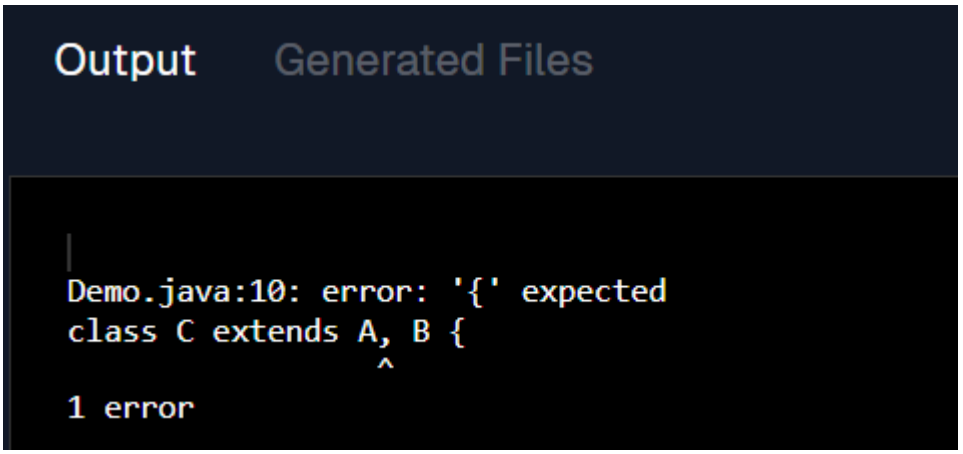
Key Points:

- **Diamond Problem:** When a class inherits from two classes that have methods with the same name, the compiler faces ambiguity, leading to confusion about which method to inherit.
- **Compile-Time Error:** Java throws a compile-time error if a class attempts to inherit from more than one class, preventing potential ambiguity and runtime errors.
- **Simplification:** Java's decision to disallow multiple inheritance simplifies the language and reduces the chances of errors.

Example of Multiple Inheritance in Java:

```java
class A {
    // Some methods and fields
}

class B {
    // Some methods and fields
}

// This code will not compile in Java
class C extends A, B {
    // Compiler error: C cannot extend from both A and B
}

public class Demo {
    public static void main(String[] args) {
        // Code to demonstrate the issue
    }
}
```

Error Message:



In this example, class C attempts to inherit from both A and B. However, this is not allowed in Java, and the code will result in a compile-time error. If you try to compile this code in an IDE, you'll see a warning or error indicating that multiple inheritance is not supported.

24-this and super method

this and super Methods in Java

In Java, the super() and this() keywords are crucial for managing constructor invocation and establishing relationships between classes. Understanding how to use these keywords effectively is vital for mastering object-oriented programming in Java.

this Keyword

The this keyword is a **reserved keyword** in Java that refers to the current instance of a class. It is primarily used for calling one constructor from another within the same class. Here are some contexts where this can be used:

- **To invoke the current class constructor**: This is commonly used when you want to call one constructor from another within the same class.
- **To pass the current class instance as an argument**: this can be passed as an argument in method or constructor calls.

Example:

Consider a scenario where class B has both a default constructor and a parameterized constructor. If you want both constructors to be executed when the parameterized constructor is called, you can use the this keyword to call the default constructor from the parameterized one.

```java
class A {
    public A() {
        System.out.println("In A's Constructor");
    }
}

class B extends A {
    public B() {
        super();  // This is present by default
        System.out.println("In B's Constructor");
    }

    public B(int n) {
        this(); // Calls the default constructor of the current class
        System.out.println("In B's parameterized constructor: " + n);
    }
}

public class Demo {
    public static void main(String[] args) {
        B obj = new B(10);
    }
}
```

Output:

super Keyword

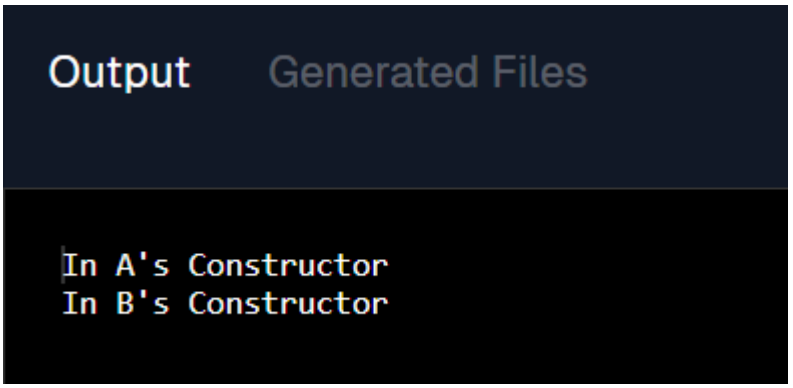
The super keyword is another reserved keyword in Java, used to refer to the superclass (parent class) of the current object. It has several important uses:

- **To call superclass constructors**: super() is often used to call a superclass constructor from a subclass. If a superclass has a parameterized constructor, you can call it explicitly using super().
- **To refer to superclass members**: super can also be used to refer to fields, methods, or constructors of the superclass, which is particularly useful when the subclass has members with the same names as those in the superclass.

Example:

```java
class A {
    public A() {
        System.out.println("In A's Constructor");
    }
}

class B extends A {
    public B() {
        super();  // Calls the superclass (A's) constructor
        System.out.println("In B's Constructor");
    }
}

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

Output:



In the above example, when an object of class B is created, the constructor of A is called first, followed by the constructor of B. This happens because the super() keyword, which calls the parent class constructor, is implicitly included in every constructor.

Important Points to Remember

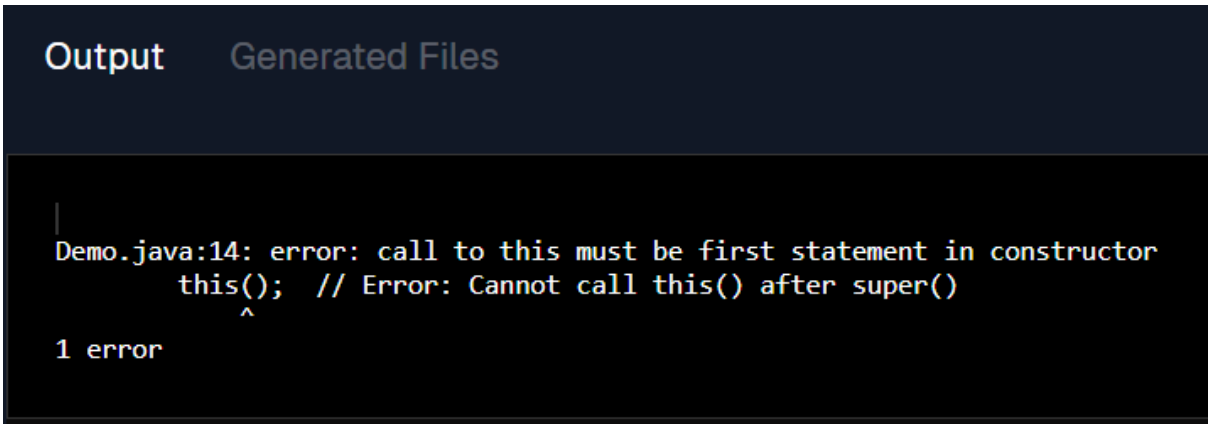
- **Implicit super()**: Every constructor in Java implicitly contains a call to super() unless you explicitly call another constructor using this(). This ensures that the parent class constructor is always invoked before the subclass constructor.
- **this() vs. super()**: Both this() and super() must be the first statement in a constructor. You cannot use both in the same constructor, as they would conflict in terms of order.
- **The Object Class**:
    - In Java, all classes implicitly inherit from the Object class if they don't explicitly extend another class. This makes Object the root of the class hierarchy.
    - When a class like A is extended by class B, A itself extends the Object class. This means that B also indirectly inherits from Object. Therefore, every class in Java, either directly or indirectly, inherits from the Object class.
- **No Multiple Inheritance**: Java does not support multiple inheritance for classes, which avoids complexity and ambiguity, especially with the super keyword. However, Java supports multilevel inheritance, where super is still relevant.

Example of Error with this() and super() Together:

```java
class A {
    A() {
        System.out.println("A is created.");
    }
}

class B extends A {
    B() {
        System.out.println("B is created.");
    }

    B(String name) {
        super(); // Calls the parent class constructor
        this();  // Error: Cannot call this() after super()
        System.out.println("B name is " + name);
    }
}

public class Demo {
    public static void main(String args[]) {
        B obj = new B("Telusko");
    }
}

```

Output:

Compile Error: "call to this must be first statement in constructor"



Commonly Asked Questions

- **Can we use both this() and super() in the same constructor?**
    - No, in Java, either this() or super() must be the first statement in a constructor. You cannot use both together, as it would create a conflict, leading to a compile-time error.
- **What happens if super() is not explicitly mentioned in a constructor?**

- If super() is not explicitly mentioned, Java implicitly calls the default constructor of the superclass.

25-Method Overriding

Method Overriding in Java

Definition: Method overriding in Java occurs when a subclass (or child class) provides a specific implementation of a method that is already defined in its superclass (or parent class). The method in the subclass must have the same name, parameters (or signature), and return type (or a subtype) as the method in the superclass. When this happens, the method in the subclass overrides the method in the superclass.

Key Points:

- **Run-Time Polymorphism:** Method overriding is a key feature that allows Java to achieve run-time polymorphism. The version of the method that gets executed is determined by the object that is used to invoke it, not by the type of reference variable.
- **Inheritance:** Method overriding is closely related to inheritance, as it allows subclasses to modify or extend the behavior of methods inherited from their parent classes.

Example of Method Overriding:

```
class A {
    // Method in the parent class
    public void show() {
        System.out.println("in A show");
    }
}


class B extends A {
    // Overridden method in the child class
    public void show() {
        System.out.println("in B show");
    }
}


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

```
        B obj = new B(); // Creating an object of subclass B
        obj.show(); // This will call the show() method in class B

        // Output:
        // in B show
    }
}
```

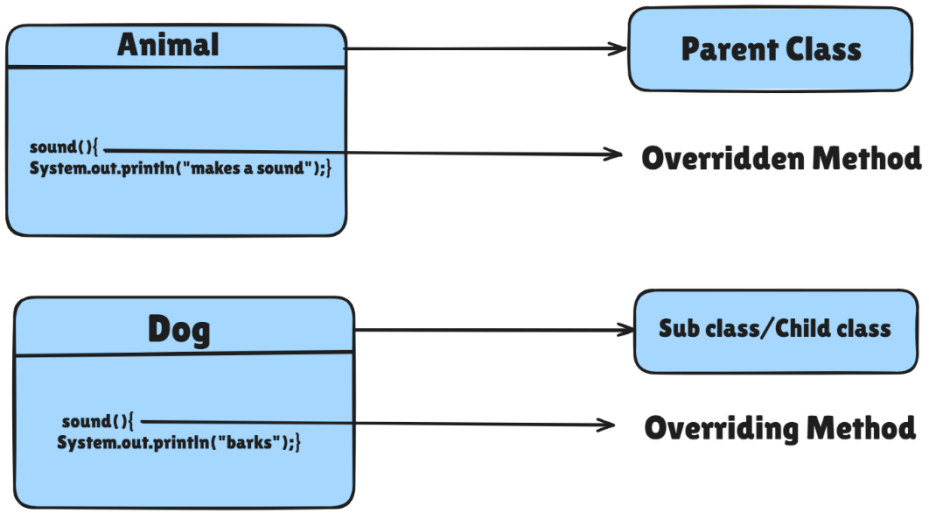
Explanation:

- In this example, class A defines a method show(). Class B, which extends class A, overrides the show() method by providing its implementation.
- When an object of class B is created and its show() method is called, the overridden method in class B is executed, displaying "in B show".
- If class B had not provided its implementation of show(), calling obj.show() would have executed the method from class A, displaying "in A show".

Importance of @Override Annotation:

- When overriding a method, it is a good practice to use the **@Override** annotation above the method. This explicitly tells the compiler that you intend to override a method from the superclass, and the compiler will generate an error if you accidentally do not match the method signature correctly.

```
@Override
public void show() {
    System.out.println("in B show");
}
```



How Method Overriding Works:

- **Dynamic Method Dispatch:** During runtime, the JVM determines which method to execute based on the actual object being referred to, not the reference type. This mechanism is called dynamic method dispatch.
- **Super Keyword:** If you need to call the overridden method from the superclass within the subclass, you can use the super keyword.

```
class B extends A {
    @Override
    public void show() {
        super.show(); // Calls the show() method from class A
        System.out.println("in B show");
    }
}
```

8.2-Packages

A **package** in Java is a mechanism to encapsulate a group of classes, sub-packages, and interfaces. It acts as a container for related classes and interfaces, allowing some to be exposed for external use while others are kept internal. Packages facilitate code reuse, prevent naming conflicts, and organize classes in a logical manner.

Key Uses of Packages

In Java, a **package** serves as a namespace that organizes classes, interfaces, and sub-packages. One of the primary advantages of using packages is their ability to **prevent naming conflicts** or **collisions**. This is especially important in large projects where multiple developers might create classes with the same name, but for different purposes.

Understanding Naming Conflicts

Imagine a scenario where you're developing a software application for a college. Different departments like Computer Science (CSE) and Electrical Engineering (EE) have their own staff management systems. Both departments need an Employee class to manage their respective employees. Without packages, having two Employee classes in the same project would lead to a conflict because the Java compiler wouldn't know which Employee class you are referring to when you try to use it.

How Packages Prevent Naming Conflicts

Java packages provide a **unique namespace** for classes, which allows you to have multiple classes with the same name in different packages. By placing classes in different packages, you can avoid naming conflicts while still organizing your code logically.

For example:

- **Package 1:** college.staff.cse.Employee
- **Package 2:** college.staff.ee.Employee

In this scenario, both departments can have their own Employee class because they are stored in different packages. Here's a detailed breakdown:

- **college.staff.cse.Employee**:
    - This class belongs to the Computer Science Engineering (CSE) department.
    - The package name college.staff.cse acts as a unique identifier or namespace for this Employee class.

    - You can have fields and methods specific to CSE department employees.
- **college.staff.ee.Employee**:
    - This class belongs to the Electrical Engineering (EE) department.
    - The package name college.staff.ee acts as a different namespace, ensuring that this Employee class does not conflict with the one in the CSE package.
    - It can have different fields and methods tailored to the needs of EE department employees.
- **Organizing Classes and Interfaces**: Packages make it easier to locate and use related classes and interfaces**.**

- **Types of Packages**

1. **Built-in Packages**: These packages are part of the Java API and provide essential classes for various functionalities.
    - **java.lang:** Contains fundamental classes, including those for primitive data types and string operations. It is automatically imported.
    - **java.io:** Provides classes for input/output operations.
    - **java.util**: Includes utility classes for data structures like LinkedList and support for date/time operations.
    - **java.net**: Provides classes for networking operations.
2. **User-defined Packages**: These are packages created by the user to organize and manage their own classes.

Accessing Packages from Another Package

There are three ways to access classes from a different package:

1. **Import the entire package**: import package.*;
2. **Import specific class**: import package.classname;
3. **Fully qualified name**: Use the complete path to the class, package.classname.

Example

Suppose we have two classes, Calc and AdvancedCalc, and we want to place them in a package named tools. To use these classes in our main class, we would:

1. Define the package in the class files:

package tools;

2. Import the package and classes in the main class:

import tools.Calc;

import tools.AdvancedCalc;

Java's standard library also uses packages. For example, the Object class is stored in the java.lang package. Even though we don't explicitly import java.lang.Object, it is available by default (java.lang.*).

Significance of Using * in Packages

In Java, when you're working with packages, you often need to import classes from other packages to use them in your code. Instead of importing each class individually, you can use the * symbol to import all the classes in a package at once. This is especially useful when you're working with multiple classes in a single package.

Simple Example

Let's say you have a package called shapes, which contains different classes like Circle, Square, and Rectangle.

```
package shapes;


public class Circle {
    // Circle-related code
}


public class Square {
    // Square-related code
}


public class Rectangle {
    // Rectangle-related code
}
```

Now, if you're writing a program in a different package, and you need to use all these shapes, you can import each class individually:

```
import shapes.Circle;
import shapes.Square;
import shapes.Rectangle;
```

Or, you can simply use the * symbol to import all the classes in the shapes package at once:

```
import shapes.*;
```

This means, instead of writing three separate import statements, you only need one.

Important Note

While import *; can save time, it's also important to use it wisely. If a package has many classes and you only need a few, it's better to import those specific classes. This makes your code clearer and can help avoid conflicts if there are classes with the same name in different packages.

Summary: Using * in imports allows you to bring in all classes from a package with a single statement, making your code cleaner and easier to manage when dealing with multiple classes from the same package. However, it should be used judiciously to maintain clarity and avoid potential naming conflicts.

Publishing and Using Packages

Packages are useful when you have multiple files in a project. You can bundle these packages and share them as libraries over as JAR or WAR file over the internet. For example, database driver classes are provided in specific packages, which can be imported and used in projects to enable database connectivity.

8.3-Access Modifiers

Access modifiers in Java are keywords that define the accessibility or scope of variables, methods, constructors, and classes. They play a crucial role in encapsulating data, ensuring security, and maintaining control over how different parts of your code interact with each other. By using access modifiers, developers can specify which parts of a program can access and modify certain data, thereby protecting sensitive information and preventing unintended interactions.

1. **Private:** It is only accessible within the class in which it is declared.
2. **Default:** The default setting is accessible within the package and is applied when no other modifier is specified.
3. **Protected**: accessible within the package and by subclasses outside the package.
4. **Public:** The public access modifier is accessible from anywhere within the program.

Examples of Each Access Modifier

Let's illustrate each access modifier using simple examples in a scenario where we have a package named **"other"** and a few classes outside of it.

Let's go through the examples for each access modifier and display the expected output or error messages.

Private Access Modifier Example

Class A in other package:

```
package other;


public class A {
    private int marks = 6;


    private void displayMarks() {
        System.out.println("Marks: " + marks);
    }


    public void show() {
        displayMarks();
    }
}
```

Class Demo outside 'other' package:

```
public class Demo {
    public static void main(String[] args) {
        other.A obj = new other.A();


        // System.out.println(obj.marks);
        // Error: marks has private access in A


        // obj.displayMarks();
        // Error: displayMarks() has private access in A


        obj.show(); // This will work
    }}
```

Output:

Marks: 6

Explanation:

The private variable marks and the private method displayMarks are not accessible directly outside the class A. However, within class A, the public method show() allows access to them, resulting in the output being displayed.

Default Access Modifier Example

Class A in other package:

```
package other;


public class A {
    int marks = 6;


    void displayMarks() {
        System.out.println("Marks: " + marks);
    }
}
```

Class B in the same other package:

```
package other;
public class B {
    public static void main(String[] args) {
        A obj = new A();
        System.out.println(obj.marks);
        obj.displayMarks();
    }
}
```

Class Demo outside other package:

```
public class Demo {
    public static void main(String[] args) {
        other.A obj = new other.A();


        // System.out.println(obj.marks);
        // Error: marks is not public in A; cannot be accessed from outside package


        // obj.displayMarks();
        // Error: displayMarks() is not public in A; cannot be accessed from outside package
    }
}
```

Output (for B class in the same package):

Marks: 6

Explanation:

Class A and Class B are both in the same package (`other`), so `marks` and `displayMarks()` in `Class A` can be accessed directly in `Class B` because they have default access.

Class Demo is in a different package. The code shows that when trying to access `marks` and `displayMarks()` from `Class A` in `Class Demo`, it results in a compilation error because default members are not accessible outside their package.

Protected Access Modifier Example

Class A in other package:

```
package other;
public class A {
    protected int marks = 6;
    protected void displayMarks() {
        System.out.println("Marks: " + marks);
    }}
```

Class C in otherPackage (different package but subclass):

```
package otherPackage;
import other.A;
public class C extends A {
    public void showMarks() {
        System.out.println(marks); // Accessible in subclass outside the package
        displayMarks(); // Accessible in subclass outside the package
    }
}
```

Class Demo outside other package:

```
public class Demo {
    public static void main(String[] args) {
        otherPackage.C obj = new otherPackage.C();
        obj.showMarks(); // This will work

        A obj2 = new A();
        // System.out.println(obj2.marks);
        // Error: marks has protected access in A

        // obj2.displayMarks();
        // Error: displayMarks() has protected access in A
    }
}
```

Output:

Marks: 6

Marks: 6

Explanation:

The protected members (marks and displayMarks) are accessible within the same package and by subclasses outside the package. In this case, C extends A and can access protected members. However, Demo cannot access them directly.

Public Access Modifier Example

Class A in other package:

```
package other;
public class A {
    public int marks = 6;
    public void displayMarks() {
        System.out.println("Marks: " + marks);
    }
}
```

Class Demo outside other package:

```
public class Demo {
    public static void main(String[] args) {
        other.A obj = new other.A();
        System.out.println(obj.marks); // Accessible everywhere
        obj.displayMarks(); // Accessible everywhere
    }
}
```

Output:

Marks: 6

Marks: 6

Explanation:

The public members (marks and displayMarks) are accessible from anywhere in the program, regardless of the package.

Key Points

- **Private**: Only accessible within the same class. Not accessible from any other class.
- **Default (Package-Private)**: accessible within the same package. Not accessible from other packages.
- **Protected**: accessible within the same package and by subclasses outside the package.
- **Public**: accessible everywhere, across all packages.

Access Modifiers Table:

Modifier	Same Class	Same Package Subclass	Same Package Non-Subclass	Different Package Subclass	Different Package Non-Subclass
Private	Yes	No	No	No	No
Default	Yes	Yes	Yes	No	No
Protected	Yes	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes	Yes

FAQs on Access Modifiers

1. What are access modifiers in Java?
    - Access modifiers are keywords that control the visibility and accessibility of classes, methods, and variables in Java.
2. Can a private method be overridden?
    - No, private methods cannot be overridden as they are not accessible outside their class.
3. What is the difference between protected and default access?
    - Protected access allows visibility within the package and to subclasses, while default access restricts visibility only within the package.
4. Can we have a private constructor in Java?
    - Yes, a private constructor can be used to prevent the instantiation of a class from outside.

26-Polymorphism

Polymorphism in Java

Definition: Polymorphism in Java is the ability of an object to take on many forms. The term "polymorphism" is derived from two Greek words***: poly***, meaning "**many**," and ***morphism***, meaning "**forms**" or "**behaviors**." Polymorphism allows objects to be treated as instances of their parent class while still allowing them to execute methods in a child-specific way.

Types of Polymorphism:

1. **Compile-time polymorphism (static polymorphism):**
    - In this type, the method behavior is determined at compile-time. This is achieved through **method overloading**.
2. **Runtime Polymorphism (Dynamic Polymorphism):**
    - In this type, the method behavior is determined at runtime. This is achieved through **method overriding**.

Explanation of Polymorphism Types:

- **Compile-time Polymorphism:**
    - Compile-time polymorphism is also known as **static polymorphism** or **early binding.** In this type of polymorphism, the method to be called is determined at the time of compilation based on the method signature.
    - **Method overloading** is a common example of compile-time polymorphism. It allows multiple methods to have the same name but differ by the number or type of parameters.

Example of Method Overloading:

```
public class Calculator {
    // Overloaded method with two parameters
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method with three parameters
    public int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

- **Runtime Polymorphism:**
    - Runtime polymorphism is also known as **Dynamic Polymorphism** or **Late Binding**. In this type, the method to be executed is determined at runtime, depending on the object's actual class.
    - **Method Overriding** is a common example of runtime polymorphism. It allows a subclass to provide a specific implementation of a method that is already defined in its parent class.

Example of Method Overriding:

```
class Animal {
    // Method in the parent class
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    // Overriding the sound method in the subclass
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog(); // Runtime Polymorphism
        myDog.sound(); // Outputs: Dog barks
    }
}
```

Comparison of Method Overloading and Method Overriding:

- **Method Overloading:**
    - Occurs within the same class.

    - Involves methods with the same name but different parameters.
    - It is a compile-time concept.
- **Method Overriding:**
    - Occurs between a superclass and a subclass.
    - Involves methods with the same name, parameters, and return type.
    - It is a runtime concept.

Advantages of Polymorphism:

- **Code Reusability:** Polymorphism allows you to reuse existing code more efficiently.
- **Flexibility:** You can write more flexible and maintainable code.
- **Simplified Interface:** Different types of objects can be accessed through the same interface, simplifying code interactions.

27-Dynamic Method Dispatch

Runtime Polymorphism in Java

Definition:
Runtime polymorphism, also known as Dynamic Method Dispatch, is a process where a call to an overridden method is resolved at runtime rather than compile-time. This mechanism allows Java to determine which method implementation to invoke based on the actual object type, rather than the reference type.

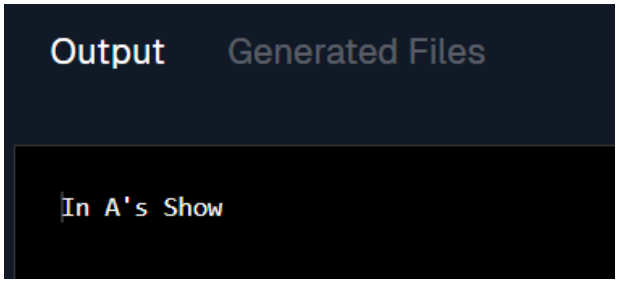
How It Works:
In dynamic method dispatch, an overridden method is called through the reference variable of a superclass. The method that gets executed is determined by the object that the reference variable points to at runtime.

Simplified Example

Here's a simpler way to achieve the same output i.e. same runtime object and type of object reference:

```java
class A {
    public void show() {
        System.out.println("In A's Show");
    }
}

class B extends A {
    public void show() {
        System.out.println("In B's Show");
    }
}

public class Demo {
    public static void main(String[] args) {
        A obj = new A(); // Reference and object of type A
        obj.show(); // Outputs: In A's Show
    }
}

```

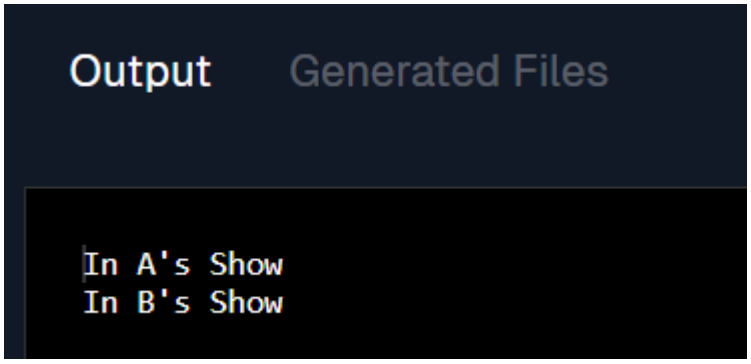
Output:

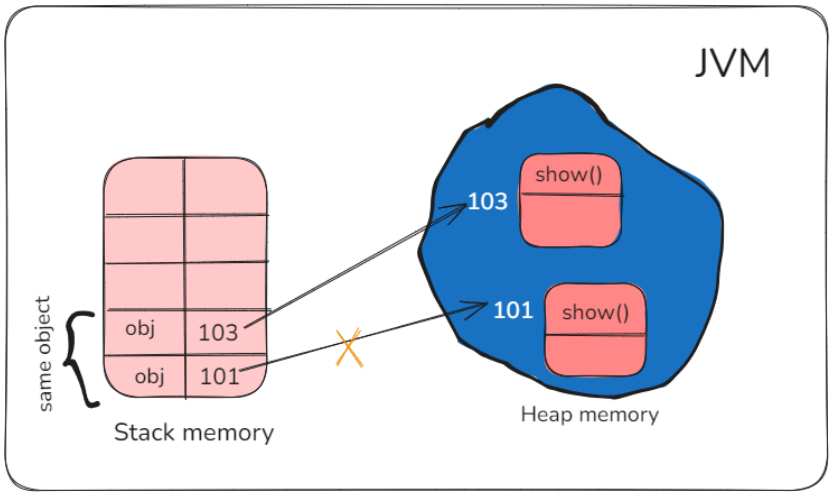


Here's an example to demonstrate dynamic method dispatch:

```java
class A {
    public void show() {
        System.out.println("In A's Show");
    }
}

class B extends A {
    public void show() {
        System.out.println("In B's Show");
    }
}

public class Demo {
    public static void main(String[] args) {
        A obj = new A(); // Reference and object of type A
        obj.show(); // Outputs: In A's Show

        obj = new B(); // Reference of type A, but object of type B
        obj.show(); // Outputs: In B's Show
    }
}
```

Output:





Explanation:

- Initially, obj is a reference to an object of type A. The show() method from class A is invoked.
- Later, same obj is assigned a new object of type B. Now, the show() method of class B is invoked, demonstrating dynamic method dispatch.

Example that demonstrates how the same reference of type A can invoke methods from classes B, and C based on the object it points to at runtime:

```java
class A {
    public void show() {
        System.out.println("In A's Show");
    }
}

class B extends A {
    public void show() {
        System.out.println("In B's Show");
    }
}

class C extends A {
    public void show() {
        System.out.println("In C's Show");
    }
}

public class Demo {
    public static void main(String[] args) {
        // Reference of type A pointing to an object of type A
        A obj = new A();
        obj.show(); // Outputs: In A's Show

        // Reference of type A pointing to an object of type B
        obj = new B();
        obj.show(); // Outputs: In B's Show

        // Reference of type A pointing to an object of type C
        obj = new C();
        obj.show(); // Outputs: In C's Show
    }
}
```

Explanation:

1. **Reference of Type A Pointing to an Object of Type A:**

```
A obj = new A();

obj.show(); // Output: In A's Show
```

- Here, obj is both a reference of type A and an object of type A. The show() method from class A is invoked, so the output is "In A's Show".

2. **Reference of Type A Pointing to an Object of Type B:**

```
obj = new B();

obj.show(); // Output: In B's Show
```

- The same reference obj is now assigned an object of type B. The show() method from class B is invoked, demonstrating dynamic method dispatch. The output is "In B's Show".

3. **Reference of Type A Pointing to an Object of Type C:**

```
obj = new C();

obj.show(); // Output: In C's Show
```

- Finally, the reference obj is assigned an object of type C. The show() method from class C is invoked, and the output is "In C's Show".

Output:

Key Points:

- **Dynamic Method Dispatch:** The method that gets executed is determined by the actual object type at runtime, even though the reference type remains the same (A in this case).
- **Polymorphism:** This example illustrates the concept of polymorphism in Java, where a single reference type can point to objects of different classes, and the method calls are resolved based on the actual object type.

Advantages of Dynamic Method Dispatch:

- **Supports Method Overriding:** Allows Java to support overriding of methods, which is essential for runtime polymorphism.
- **Flexibility:** A class can define common methods that will be used by all its subclasses, while subclasses can provide specific implementations of those methods.

- **Enhanced Functionality:** Subclasses can add specific methods, enhancing the functionality of the superclass.

28-Final Keyword

Final Keyword in Java

The final keyword in Java is used to restrict the modification of variables, methods, and classes. It plays a crucial role in defining constants and preventing inheritance or method overriding. Let's explore how final can be used in different contexts:

1. Final Variable

A variable declared with the final keyword cannot be modified once it is initialized. This makes the variable effectively a constant.

Explanation:

```
1  public class Demo {
2      public static void main(String[] args) {
3          final int num = 8;
4          num = 9; // This will cause a compile-time error
5          System.out.println(num);  // Output: 8
6      }
7  }
8
```

Output:

```
Demo.java:4: error: cannot assign a value to final variable num
        num = 9; // This will cause a compile-time error
        ^
1 error
```

***Explanation*:** Once num is declared as final, any attempt to change its value will result in a compilation error.

2. Final Class

A class declared as final cannot be extended (inherited). This is useful when you want to prevent other classes from inheriting and modifying its behavior.

Example:

```java
final class Calc {
    public void show() {
        System.out.println("In Calc's show method");
    }

    public void add(int a, int b) {
        System.out.println("Addition is: " + (a + b));
    }
}
class AdvCalc extends Calc {
    // This will cause a compile-time error
   // As class AdvCalc extends Calc { }
}
public class Demo {
    public static void main(String[] args) {
      Calc obj=new Calc();
      obj.show();
      obj.add(5,4);



    }
}

```

Output:

```
Demo.java:10: error: cannot inherit from final Calc
class AdvCalc extends Calc {
                      ^
1 error
```

***Explanation*:** The final keyword prevents the Calc class from being extended by any other class.

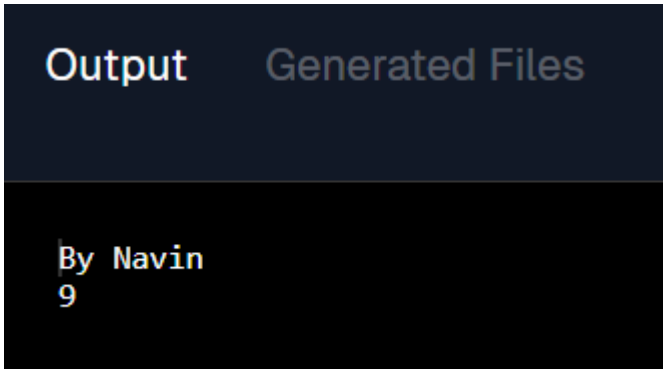
3. Final Method

A method declared with the final keyword cannot be overridden by subclasses. This ensures that the method's implementation remains unchanged in all derived classes.

Example:

```java
class Calc {
    public final void show() {
        System.out.println("By Navin");
    }

    public void add(int a, int b) {
        System.out.println(a + b);
    }
}

class AdvCalc extends Calc {
    // This will cause a compile-time error
    // public void show() { System.out.println("By John"); }
}

public class Demo {
    public static void main(String[] args) {
        AdvCalc obj = new AdvCalc();
        obj.show();  // Output: By Navin
        obj.add(5, 4);  // Output: 9
    }
}
```

Output



Explanation: The show method in Calc is marked as final, so it cannot be overridden by AdvCalc.

Characteristics of final Keyword

- **Final Variables**: Once initialized, their value cannot be changed. This is useful for constants.

- **Final Methods**: These cannot be overridden, ensuring that the implementation stays consistent across all subclasses.
- **Final Classes**: These cannot be extended, which is useful when a class is meant to be used as-is without modification.
- **Initialization**: Final variables must be initialized when declared or within the constructor, ensuring they are assigned exactly once.
- **Performance**: Using final can improve performance as the compiler can optimize the code knowing that the value or method won't change.
- **Security**: The final keyword helps in protecting data and methods from being modified by malicious code.

Conclusion

The final keyword in Java is a powerful tool for creating constants, preventing inheritance, and ensuring methods are not overridden. It provides security, performance, and clarity in the codebase by restricting unwanted modifications.

FAQs about the final Keyword in Java

1. **What is a final method in Java?**

   A final method is a method that cannot be overridden by subclasses. It ensures that the method's behavior remains unchanged across the inheritance hierarchy.

2. **Is a final method inherited?**

   Yes, a final method is inherited by the subclass, but it cannot be overridden or modified.

3. **Can a final class be instantiated?**

   Yes, a final class can be instantiated, just like any other class. However, it cannot be extended.

4. **What is the difference between final and static keywords in Java?**

   - Final Keyword: Used to declare variables, methods, or classes as unmodifiable.
   - Static Keyword: Used to declare class members (variables and methods) that belong to the class itself, not to instances of the class.

5. **Can we declare a String variable as final?**

   Yes, you can declare a String variable as final, which means that once it is initialized, it cannot be changed.

29-Object Class equals toString hashcode

Introduction to the Object Class

In Java, the Object class is the root class of the class hierarchy. Every class in Java is directly or indirectly derived from the Object class, meaning that it serves as the parent class for all other classes. If a class does not explicitly extend another class, it implicitly extends the Object class, making all methods of the Object class available to every Java class.

The Object class provides several key methods that are essential for basic object manipulation, including **toString()**, **equals()**, and **hashCode()**. Understanding these methods is crucial for working with objects in Java, especially when dealing with collections, object comparison, and debugging.

The toString() Method

The toString() method in Java is used to obtain a string representation of an object. By default, when you print an object, the toString() method is implicitly called, returning a string that includes the class name followed by the "@" symbol and the object's hashcode in hexadecimal form.

Syntax of toString() Method:

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}

```

Default Behavior of toString() Method: By default, the toString() method returns the name of the class, followed by "@" and the hexadecimal representation of the object's hash code. This behavior is useful when you need a basic string representation of an object, but in most cases, it's more practical to override the toString() method to provide a more meaningful output.

Example:

```
class Laptop {
    String model;
    int price;
}

public class Demo {
    public static void main(String[] args) {
        Laptop obj = new Laptop();
        obj.model = "Lenovo Yoga";
        obj.price = 1000;
        System.out.println(obj);
    }
}
```

Output:

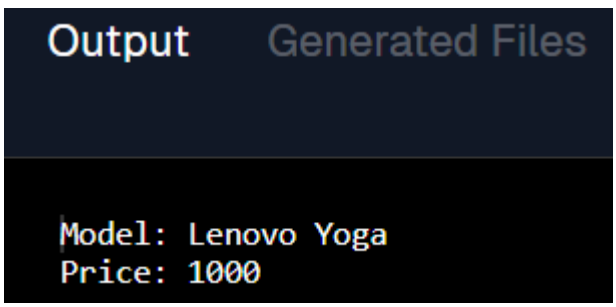
In the above example, when we try to print the object obj, it returns a string that represents the class name and hash code. This occurs because the toString() method of the Object class is invoked by default.

Overriding the toString() Method: To make the output more meaningful, you can override the toString() method in your class.

Example:

```
1  class Laptop {
2      String model;
3      int price;
4
5      public String toString() {
6          return "Model: " + model + "\nPrice: " + price;
7      }
8  }
9
10 public class Demo {
11     public static void main(String[] args) {
12         Laptop obj = new Laptop();
13         obj.model = "Lenovo Yoga";
14         obj.price = 1000;
15         System.out.println(obj);
16     }
17 }
```

Output:



Here, the toString() method is overridden to provide a more meaningful representation of the Laptop object.

The hashCode() Method

The hashCode() method returns a hash code value (an integer) for the object. This hash code is primarily used in hashing-based collections and algorithms such as HashMap, HashSet, and Hashtable.

Syntax of hashCode() Method:

```
public int hashCode() {
    // Returns the hash code value for the object.
}
```

When you override the equals() method in a class, it is also a good practice to override the hashCode() method to ensure that objects that are considered equal have the same hash code.

The equals() Method

The equals() method is used to compare two objects for equality. The default implementation of equals() provided by the Object class checks whether two references point to the same object in memory.

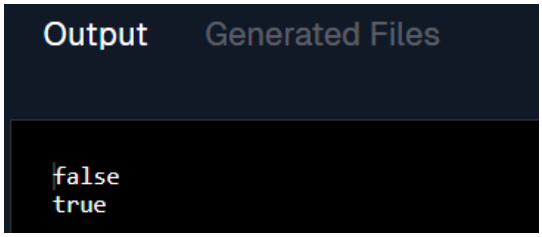
Syntax of equals() Method:

```
public boolean equals(Object obj) {
    // Compares this object with the specified object for equality.
}
```

Example:

```
class laptop{
String model;
int price;

public String toString(){
 return "model : "+model+ "price :"+ price;
}
public boolean equals(laptop that)
{
    return  (this.model.equals(that.model) && this.price == (that.price));
}
}
public class Demo{
public static void main(String[] args){
 laptop obj=new laptop();
laptop obj1=new laptop();

obj.model="Lenovo yoga";
obj.price=1000;
obj1.model="Lenovo yoga";
obj1.price=1000;

System.out.println(obj == obj1);
System.out.println(obj.equals(obj1));
}
}
```

Output:



In this example, the equals() method is overridden to compare the content of two Laptop objects rather than their memory references.

Advantages of Overriding toString(), equals(), and hashCode()

- **Improved Debugging:** Overriding toString() allows you to get a more meaningful output when printing objects, making it easier to debug.
- **Accurate Object Comparison:** Overriding equals() ensures that object comparisons are based on actual content rather than memory references.
- **Proper Hashing:** Overriding hashCode() ensures that objects that are considered equal have the same hash code, which is critical for the correct functioning of hash-based collections.

Conclusion

The Object class in Java provides fundamental methods such as toString(), equals(), and hashCode(), which are crucial for object manipulation. By understanding and correctly overriding these methods, you can enhance the functionality and readability of your Java programs.

FAQs

1. **Can we override the toString() method in every class?**
    - Yes, you can override the toString() method in any class to provide a custom string representation of your objects.
2. **Why is it important to override the hashCode() method when equals() is overridden?**
    - It's important because objects that are considered equal by the equals() method must have the same hash code to maintain the contract between equals() and hashCode().
3. **What happens if we do not override the equals() method?**
    - If you do not override the equals() method, the default implementation from the Object class will be used, which compares memory references rather than content.
4. **Is it mandatory to override toString() in every class?**

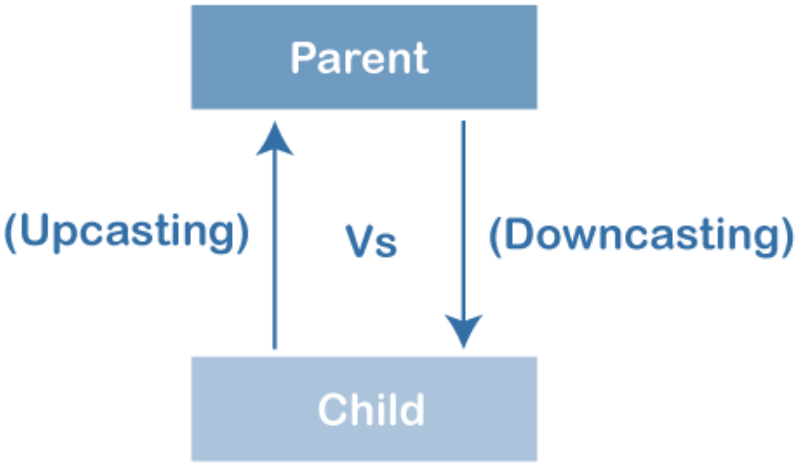
- No, it's not mandatory, but it's often useful for debugging and logging purposes.

5. **Can two different objects have the same hash code?**
    - Yes, different objects can have the same hash code due to hash code collisions, but good hash code implementations minimize the chances of collisions.

30-Upcasting and Downcasting

Typecasting is the process of converting one data type to another. In Java, this concept extends beyond basic data types to include objects, leading to what is known as **object typecasting**. Object typecasting involves converting a reference of one type into another, specifically between parent and child classes. This can be categorized into two types:

- **Upcasting** (Child to Parent)
- **Downcasting** (Parent to Child)

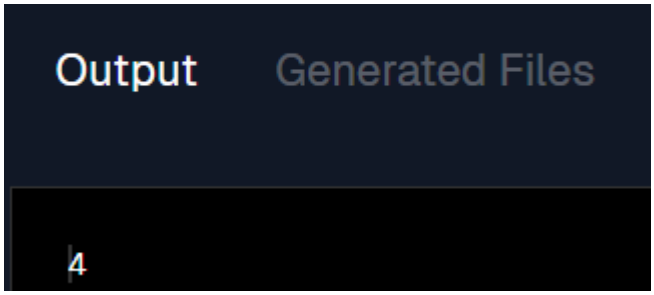


Typecasting ensures that variables are processed correctly by functions. While upcasting can be done implicitly or explicitly, downcasting requires explicit casting and can be risky if not handled properly.

Example of Typecasting

```java
public class Demo {
    public static void main(String[] args) {
        double d = 4.5;
        int i = (int) d;
        System.out.println(i); // Output: 4
    }
}
```

In this example, we are typecasting a double to an int. The compiler warns that data might be lost due to the conversion, but we accept it. The output is 4, demonstrating how typecasting works with primitive types.



Object-Oriented Typecasting (OOP)

Upcasting and **Downcasting** are essential concepts in Java's object-oriented programming (OOP). They allow us to cast objects from one type to another within an inheritance hierarchy.

Upcasting

Upcasting is the process of converting a child class reference to a parent class reference. It's a safe operation that doesn't require explicit casting because every child object is an instance of the parent class. Upcasting is also known as **generalization** or **widening**.

- **Benefits of Upcasting**:
    - Simplifies code by allowing a uniform interface to handle different objects.
    - Enables polymorphism, where a single method can be used on objects of different types.

Example:

```
1  class A {
2      public void show1() {
3          System.out.println("in A's show1 method");
4      }
5  }
6
7  class B extends A {
8      public void show2() {
9          System.out.println("in B's show2 method");
10     }
11 }
12
13 public class Demo {
14     public static void main(String[] args) {
15         A obj = new B(); // Upcasting
16         obj.show1(); // Output: in A's show1 method
17     }
18 }
```

In this example, B is upcasted to A. The show1() method of class A is accessible, but the show2() method of B is not, since the reference type is A.

```
Output    Generated Files

in A's show1 method
```

Downcasting

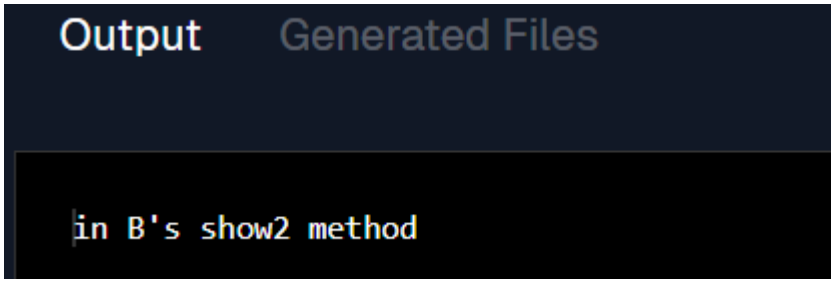
Downcasting is the reverse of upcasting, where we cast a parent class reference back to a child class reference. This operation is more restrictive and can lead to runtime exceptions if not handled correctly. In Java, downcasting must be done explicitly using the cast operator.

- **Risks of Downcasting**:
    - Can throw ClassCastException if the object being cast is not actually an instance of the target class.
    - Should be used cautiously, often with checks using the instanceof operator.

Example:

```java
class A {
    public void show1() {
        System.out.println("in A's show1 method");
    }
}

class B extends A {
    public void show2() {
        System.out.println("in B's show2 method");
    }
}

public class Demo {
    public static void main(String[] args) {
        A obj = new B(); // Upcasting
        B obj1 = (B) obj; // Downcasting
        obj1.show2(); // Output: in B's show2 method
    }
}
```

Here, obj is first upcasted to A, then downcasted back to B. The method show2() from class B is accessible after downcasting.



Conclusion

Whether you are performing upcasting or downcasting, these operations are fundamental to understanding polymorphism and inheritance in Java. Upcasting is generally safer and more commonly used, while downcasting requires careful handling to avoid runtime errors.

FAQs on Upcasting and Downcasting

1. **What is the difference between upcasting and downcasting?**
    - Upcasting refers to converting a child class reference to a parent class reference, while downcasting is converting a parent class reference back to a child class reference.
2. **Is upcasting always safe in Java?**
    - Yes, upcasting is safe and can be done implicitly because every child class object is an instance of its parent class.
3. **Why is downcasting not safe?**
    - Downcasting can lead to ***ClassCastException*** at runtime if the object being cast is not an instance of the target class. It requires explicit casting and often involves runtime checks using the instanceof operator.
4. **Can we access child class methods after upcasting?**
    - No, after upcasting, only methods and variables defined in the parent class are accessible. To access child class-specific methods, downcasting is required.

.

➢ 31-Wrapper Classes

➢ **Introduction**

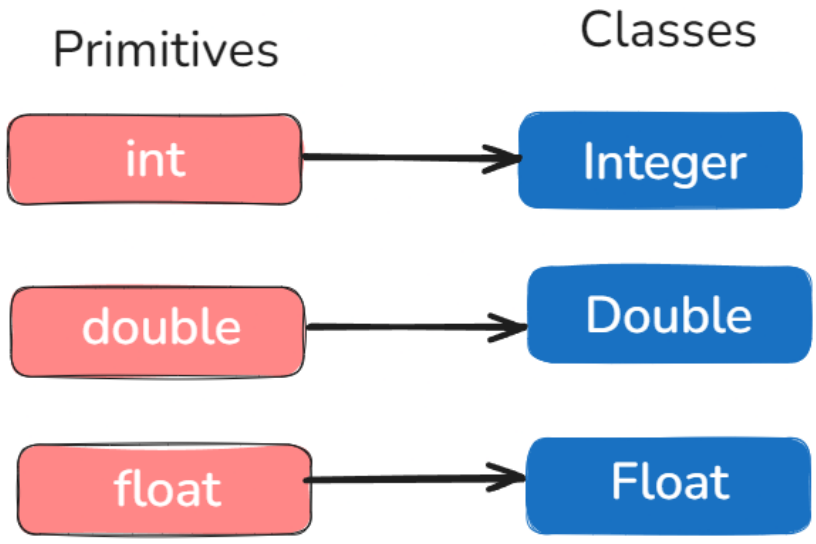
When discussing Java and Object-Oriented Programming (OOP), it's important to note that Java is not a purely object-oriented language. This is because it uses primitive data types such as int for integers, double or float for decimal values, and char for characters. These primitive data types are not objects, which contrasts with the OOP principle that everything should be an object.

➢ **Why Java is Not Purely Object-Oriented**

In a purely object-oriented language, all entities, including data types, must be represented as objects. However, in Java, primitive data types do not follow this principle, which is why Java is considered a **partially object-oriented language**.

➢ **Wrapper Classes in Java**

To bridge the gap between primitive types and the object-oriented nature of Java, the language provides **Wrapper Classes**. A Wrapper class in Java is a class whose object wraps or contains a primitive data type. When we create an object of a wrapper class, it encapsulates the primitive data type inside it. This allows the primitive data to be used in contexts that require objects.



➢ **Primitive Data Types and Their Corresponding Wrapper Classes**

For every primitive data type in Java, there is a corresponding wrapper class:

Primitive Data Type	Wrapper Class
char	Character
byte	Byte

Primitive Data Type	Wrapper Class
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

- **Examples and Scenarios**

Scenario 1: Primitive to Wrapper Type

In this scenario, we convert a primitive type to its corresponding wrapper type.

```
public class Demo {
    public static void main(String[] args) {
        int num = 7;
        Integer num1 = new Integer(8);  // Deprecated
        System.out.println(num1);  // Output: 8
    }
}
```

Note: *The use of new Integer(8) is deprecated in recent versions of Java. Instead, you should rely on autoboxing, where the conversion from a primitive type to a wrapper class is done automatically.*

- **Scenario 2: Autoboxing and Unboxing**

Autoboxing: The automatic conversion of a primitive type to the object of its corresponding wrapper class.

```
int num = 7;
Integer num1 = num;  // Autoboxing
```

Unboxing: The reverse process, where the object of a wrapper class is automatically converted back to its corresponding primitive type.

```
int num2 = num1;  // Unboxing
System.out.println(num2);  // Output: 7
```

➢ **Scenario 3: String to int Conversion**

This scenario demonstrates how to convert a String to an int using a wrapper class method.

➢ **Advantages of Wrapper Classes**

```
public class Demo {
    public static void main(String[] args) {
        String str = "12";
        int num3 = Integer.parseInt(str);
        System.out.println(num3 * 2);  // Output: 24
    }
}
```

Collections Framework: Collections in Java can only hold objects, not primitive types. Wrapper classes enable the use of primitive data in collections like ArrayList, HashSet, etc.

Method Operations: Wrapper classes provide useful methods like compareTo(), equals(), and toString().

Cloning: Only objects can be cloned in Java, not primitives.

Null Values: Wrapper objects can be null, while primitives cannot.

Serialization: Serialization, the process of converting an object into a byte stream, only works with objects.

➢ **FAQs on Wrapper Classes**

Which are the wrapper classes in Java?

Wrapper classes in Java are classes whose objects encapsulate primitive data types.

Why use the wrapper class in Java?

Wrapper classes are used to convert primitive data types into objects, which is necessary when you want to modify arguments passed into a method or store primitives in collections.

What are the 8 wrapper classes in Java?

- The 8 wrapper classes in Java are Boolean, Byte, Short, Character, Integer, Long, Float, and Double.