```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
       }
}
```

1. public class Main {

- **class**: In Java, every program must be inside a class.
- **Main**: This is the **name** of the class. It starts with a capital letter (as per naming convention).
- **public**: This means the class is accessible from anywhere in the program.

2. public static void main(String[] args) {

This is the **main method**, where your Java program **starts running**.

- **public**: This method can be called from outside the class.
- **static**: You don't need to create an object to call this method.
- **void**: The method doesn't return anything.
- **main**: The Java Virtual Machine (JVM) looks for this exact name to run your program.
- **String[] args**: This is used to receive input from the command line (optional for basic programs).

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

- This line **prints text** to the screen.
- **System.out**: Refers to the standard output (your screen).
- **println()**: Prints the text and moves to a new line.
- "Hello, World!": This is the message being printed.

Compile the program `javac Main.java`

Run the program `java Main`

A .class file in Java contains platform-independent bytecode generated after compiling a .java file.It can run on any OS with a JVM, without needing the original source code.

Int – 4bytes, long – 8bytes, short – 2bytes, float – 4bytes, double –

8bytes double num = 5.6; float num = 5.6f;

1byte – 8bits

Total bits = **8** , One bit is used for **sign** (positive or negative), so: Remaining **7 bits** are for the value. So the range becomes:

Using those 8 bits:

- You can store **256 total values**
- If it's **unsigned** → values from **0 to 255**
- If it's **signed** → values from **-128 to 127** (that's -2^7 to $2^7 - 1$)

- **Positive values**: The range is `0 to 127` (which is 127 numbers).
- **Negative values**: The range is `-128 to -1` (which is 128 numbers).

- **MSB (Most Significant Bit)**: The leftmost bit is **1**, which represents **128**.

- **LSB (Least Significant Bit)**: The rightmost bit is **0**, which represents **0**.

```
128  64  32  16  8  4  2  1
-------------------------

1   1   0   1   0  1  1  0  (binary number)
```

Unicode is a **universal character encoding standard** used to represent characters from all languages uses \uXXXX format, Java uses **Unicode internally** to represent characters, making it capable of handling **global languages**.

char ch = '\u0041'; // Unicode for 'A'

System.out.println(ch); // Output: A

A **literal** is a fixed value assigned to a variable — like a number, character, or string directly written in the code.

int num = 100; // 100 is an integer

literal float pi = 3.14f; // 3.14f is a

float literal char grade = 'A'; // 'A'

is a character literal

String msg = "Hello"; // "Hello" is a string literal

Type Conversion in Java

- **Implicit (Widening)** – Auto conversion from smaller to larger type.

Example: `int a = 10; double b = a;`

- **Explicit (Narrowing)** – Manual conversion from larger to smaller type.

Example: `double x = 9.8; int y = (int) x;`

Type promotion is the process where Java automatically converts smaller data types to larger data types

byte a = 10;

byte b = 20;

int c = a + b; // byte + byte → promoted to int

/ : for to get quotient. % : for to get remainder. = : assignment operator. (<,>,==,!=,<=,>=)

== : comparison operator.

(T==F) Int a = 7;

Int b = ++a; //First increment & then fetch which prints 8.

Int b = a++; //Fetch the value & then increment which prints 7.

Conditional Statement

If – Else, If – Else If – Else

Ternary Operator is a concise way of writing if else condition in one line.

condition ? value-if true : value-if false;

Ex :

```
int

n=5;

int r=0;

r=n%2==0 ? 10 : 20 ;
System.out.println(r);
```

Switch Case

```
Int n=3;

Switch(n){

Case 1:

S.o.pln("Monday");

Case 2:

S.o.pln("Tuesday");

Case 3:

S.o.pln("Wednesday");

}
```

- break prevents fall-through (execution of next cases).
- default runs when **no case matches**.

Loops :

For Loop : Used when you know **how many times** to repeat.

```
public class ForLoopExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 3; i++) {
            System.out.println("For Loop: " + i);              }}}
```

While Loop : Checks the condition first, then runs the loop.

```
public class WhileLoopExample {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 3) {
            System.out.println("While Loop: " + i);
            i++; }}}
```

Do-While Loop : Runs once even if the condition is false initially.

```
public class DoWhileFalseCondition {
    public static void main(String[] args) {
        int i = 10;
        do {
            System.out.println("This will run once even though condition is false.");
        } while (i < 5);  // false condition
    }}
```

In Java, **variables** are classified into **three main types** based on their scope and usage:

1. Local Variables

- Declared inside methods, constructors, or blocks.
- Scope is limited to the method/block where they are defined.
- Must be initialized before use.

2. Instance Variables

- Declared inside a class but **outside any method**.
- Belong to each object (instance) of the class.
- Each object has its own copy.

3. Static Variables (Class Variables)

- Declared with the `static` keyword inside a class but outside methods.
- Belong to the class, not to individual objects.
- Shared among all instances of the class.

```
public class VariableTypes {

    static int staticVar = 1;        // Static variable

    int instanceVar = 2;             // Instance variable



    void show() {

        int localVar = 3;            // Local variable

        System.out.println("Local: " + localVar);

        System.out.println("Instance: " + instanceVar);

        System.out.println("Static: " + staticVar);

    }

    public static void main(String[] args) {

        new VariableTypes().show();

    }

}
```

Output :

```
Local: 3          Instance: 2       Static: 1
```

1. String

- **Immutable**: Once created, it **cannot be changed**.
- Every modification creates a **new object** in memory.
- Safe for **multi-threaded** use but **less efficient** for many modifications.

```
String str = "Hello";
str = str + " World";  // Creates a new object
System.out.println(str);  // Output: Hello World
```

2. StringBuffer

- **Mutable**: Can be changed after creation.
- **Thread-safe**: Methods are synchronized.
- Slightly **slower**, but safe for **multi-threaded** environments.

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb);  // Output: Hello World
```

3. StringBuilder

- Also **mutable**, like `StringBuffer`.
- **Not thread-safe**, but **faster**.
- Best choice for **single-threaded** applications needing lots of changes.

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" Java");
System.out.println(sb);  // Output: Hello Java
```

Feature	String	StringBuffer	StringBuilder
Mutability	Immutable	Mutable	Mutable
Thread-Safety	Thread-safe	Thread-safe (Synchronized)	Not thread-safe
Performance	Slower	Slower	Faster
Use-Case	Fixed values	Multi-threaded tasks	Single-threaded tasks

Thread: A thread is a lightweight unit of a process that runs part of your program independently.

Thread-Safe: Ensures correct behavior when multiple threads access shared data. Only one person can book a seat at a time. Once it's booked, it's locked for others. No double bookings.

Not Thread-Safe: May lead to errors or unexpected behavior when accessed by multiple threads at once. Many people try to book the same seat at once.Result: Two people get the same seat — conflict!

Static Variable in Java

A **static variable** is a variable that is shared by **all instances** of a class. It is declared using the `static` keyword.

- **Belongs to the class**: A static variable is associated with the class, rather than with any instance (object) of the class.
- **Single copy**: There is only **one copy** of the static variable, no matter how many objects are created from the class.
- **Access**: Static variables can be accessed directly by the class name or through an object.

Key Points:

- It is initialized when the class is loaded.
- It can be accessed by both static and non-static methods, but non-static methods need an instance to access it.

Example:

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

    // Constructor increments the static variable
    Counter() {
        count++;
    }

    // Static method to display count
    static void displayCount() {
        System.out.println("Count: " + count);
    }
}

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

        // Access static variable through class name
        Counter.displayCount();  // Output: Count: 2
    }
}
```

Explanation:

- The `count` variable is static, meaning it is shared by all objects of the `Counter` class.
- When two objects (`c1` and `c2`) are created, the static `count` variable is incremented each time.
- The output will show that the static variable `count` is **2** (since both objects increment it).

Instance Variable in Java

An **instance variable** is a variable that is defined within a class but outside of any method, constructor, or block. Each object (instance) of the class has its own copy of the instance variable.

Key Points:

- **Unique to each instance**: Each object of the class has a separate copy of the instance variable.
- **Defined without the `static` keyword**: Instance variables do not use the `static` keyword.
- **Accessed via object**: You access instance variables through an instance (object) of the class.

Example:

```
java
CopyEdit
class Person {
    // Instance variables
    String name;
    int age;

    // Constructor to initialize instance variables
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method to display the instance variables
    void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

public class InstanceVariableExample {
    public static void main(String[] args) {
        // Creating objects (instances) of Person
        Person person1 = new Person("Alice", 25);
        Person person2 = new Person("Bob", 30);

        // Accessing instance variables through objects
        person1.display();
        person2.display();
    }
}
```

Output:

```
Name: Alice
Age: 25
Name: Bob
Age: 30
```

Explanation:

- **Instance variables** `name` and `age` are specific to each object of the `Person` class.
- `person1` has its own copy of `name` and `age`, and so does `person2`.
- Each object can hold different values for the instance variables.

Local Variable in Java

A **local variable** is a variable that is declared inside a method, constructor, or block and can only be used within that method, constructor, or block.

Key Points:

- **Scope**: The scope of a local variable is limited to the block of code in which it is declared (e.g., a method or a loop).
- **Lifetime**: The lifetime of a local variable is only during the execution of the method, constructor, or block where it is defined.
- **No default value**: Local variables must be explicitly initialized before they are used.

Example:

```
java
CopyEdit
class LocalVariableExample {
    // Method with local variables
    void display() {
        // Local variables
        int x = 10;
        String name = "John";

        // Using local variables
        System.out.println("x: " + x);
        System.out.println("Name: " + name);
    }

    public static void main(String[] args) {
        // Creating an instance of LocalVariableExample
        LocalVariableExample obj = new LocalVariableExample();

        // Calling the method
        obj.display();
    }
}
```

Output:

```
x: 10
Name: John
```

Explanation:

- **Local variables** `x` and `name` are declared inside the `display()` method.
- They are accessible only within the `display()` method.
- Once the method execution finishes, the local variables go out of scope and are destroyed.

In Java, you can take input from the user using the `Scanner` class. Here's how to do it:

Steps to take input:

1. **Import** the `Scanner` class.
2. **Create** a `Scanner` object to read input.
3. **Use** methods like `nextLine()`, `nextInt()`, `nextDouble()` etc., to get different types of input.

Example for taking input:

```
java
CopyEdit
import java.util.Scanner;

public class InputExample {
    public static void main(String[] args) {
        // Create a Scanner object
        Scanner scanner = new Scanner(System.in);

        // Taking string input
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();

        // Taking integer input
        System.out.print("Enter your age: ");
        int age = scanner.nextInt();

        // Taking double input
        System.out.print("Enter your height: ");
        double height = scanner.nextDouble();

        // Display the inputs
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Height: " + height);

        // Close the scanner
        scanner.close();
    }
}
```

Input and Output Example:

```
Enter your name: John
Enter your age: 25
Enter your height: 5.9

Name: John
Age: 25
Height: 5.9
```

Feature	Local Variable	Instance Variable	Static Variable
Scope	Limited to the method/block.	Accessible via object instances.	Shared across all instances of the class.
Lifetime	Exists only during method/block execution.	Tied to the object lifetime.	Exists as long as the class is loaded.
Initialization	Must be initialized before use.	Initialized in the constructor or at declaration.	Initialized once when the class is loaded.
Access	Only within the method/block.	Accessed through objects.	Accessed via class name or objects.

When to Use Variables:

1. **Local Variables**:
   - Use for temporary data within methods.
2. **Instance Variables**:
   - Use for data specific to an object.
3. **Static Variables**:
   - Use for data shared across all instances of a class.
   - **Static variable** `count` is shared by all instances (`obj1`, `obj2`, and `obj3`).
   - The **same value** of `count` is modified each time an object is created, and it reflects across all objects.

Class Loader in Java

A **Class Loader** in Java loads classes into the JVM at runtime. It ensures that the classes needed by a program are available when required.

Types of Class Loaders:

1. **Bootstrap Class Loader**: Loads core Java libraries.
2. **Extension Class Loader**: Loads classes from the `ext` directory.
3. **System/Application Class Loader**: Loads classes from the classpath.

Access Modifiers in Java

Access modifiers control the visibility and accessibility of classes, methods, and variables in Java. There are four main types:

1. **Public**:
   - **Visibility**: The class, method, or variable is accessible from **any other class**.
   - **Usage**: Best used when you want to make something globally accessible.
   - **Example**:

```
public class MyClass { }
```

2. **Private**:
    - **Visibility**: The method or variable is accessible **only within the same class**.
    - **Usage**: Used for **encapsulation** to hide details from outside classes.
    - **Example**:

      ```
      private int age;
      ```

3. **Protected**:
    - **Visibility**: The method or variable is accessible within the **same package** or by **subclasses** (even if they are in different packages).
    - **Usage**: Typically used in inheritance scenarios.
    - **Example**:

      ```
      protected void display() { }
      ```

4. **Default (No Modifier)**:
    - **Visibility**: The method or variable is accessible **only within the same package**.
    - **Usage**: Used when you want access control within the same package, but not from other packages.
    - **Example**:

      ```
      void display() { }  // Default access
      ```

Example:

```
java
CopyEdit
public class AccessModifiersExample {
    public int publicVar;    // Accessible everywhere
    private int privateVar;  // Accessible only in this class
    protected int protectedVar;  // Accessible within same
package or subclasses
    int defaultVar;          // Accessible within the same
package
}
```

- **Public**: Accessible everywhere.
- **Private**: Accessible only within the class.
- **Protected**: Accessible within the package and subclasses.
- **Default**: Accessible within the package.

1. `final`

- **Keyword** used to **restrict** changes.
- It can be applied to **variables**, **methods**, and **classes**.

Where Used	Meaning
final **variable**	Value cannot be changed (constant).
final **method**	Method cannot be overridden in subclasses.
final **class**	Class cannot be inherited/extended.

```
final int a = 10;
// a = 20;  // ❌ Error: cannot assign a value to final variable
final class Animal {
    final void sound() {
        System.out.println("Animal makes sound");   }}
// class Dog extends Animal {}  // ❌ Error: cannot inherit from final
class
```

---

2. `finally`

- **Block** used in **exception handling**.
- Code inside `finally` **always executes** whether exception occurs or not.
- Good for **closing resources** like files, database connections.

```
public class Example {
    public static void main(String[] args) {
        try {
            int a = 5/0;  // Exception here
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e);
        } finally {
            System.out.println("Finally block always
executes!");   }}}
```

---

3. `finalize()`

- **Method** in Object class.
- Used to perform **cleanup operations** before the object is **garbage collected**.
- You can **override** `finalize()` in your class if needed.

⚡ But **Note**:
`finalize()` is **deprecated** (from Java 9 onwards) because Java now handles cleanup automatically.

```
public class Example {
    protected void finalize() { //Syntax
        System.out.println("Finalize method called!");  }

    public static void main(String[] args) {
        Example obj = new Example();
        obj = null;
        System.gc();  // Requesting garbage collector to run    }}
```

1. toString() : Returns a string representation of an object.

```
class A {
    public String toString() {
        return "Object A";  }}
public class Main {
    public static void main(String[] args) {
        A obj = new A();
        System.out.println(obj);  // Object A   }}
```

---

2. hashCode() : Returns the hash value (integer) of an object.

```
class A {}
public class Main {
    public static void main(String[] args) {
        A obj = new A();
        System.out.println(obj.hashCode());  // e.g., 12345678  }}
```

---

3. equals(Object obj) : Checks if two objects are **logically equal**.

```
class A {
    int id = 1;
    public boolean equals(Object o) {
        A a = (A)o;
        return this.id == a.id;     }}
public class Main {
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new A();
        System.out.println(a1.equals(a2));  // true }}
```

---

4. getClass() : Returns the **runtime class** of an object.

```
class A {}
public class Main {
    public static void main(String[] args) {
        A obj = new A();
        System.out.println(obj.getClass());  // class A }}
```

Creates a **copy** of an object (shallow copy).

```
class A implements Cloneable {
    int x = 10;
    public Object clone() throws CloneNotSupportedException {
        return super.clone();       }}
public class Main {
    public static void main(String[] args) throws
CloneNotSupportedException {
        A obj1 = new A();
        A obj2 = (A)obj1.clone();
        System.out.println(obj2.x);  // 10      }}
```

6. wait(), notify(), notifyAll() : Used for **thread communication** inside a synchronized block.

```
class A {
    synchronized void test() throws InterruptedException {
        wait();
        System.out.println("Resumed");  }
    synchronized void resume() {
        notify();       }}
```

7. toHexString(int i) : Converts an integer to a **hexadecimal string**.

```
public class Main {
    public static void main(String[] args) {
        System.out.println(Integer.toHexString(255));  // ff    }}
```

Method	Short Use
`toString()`	Object ➔ String
`hashCode()`	Object ➔ Hashcode
`equals()`	Compare objects
`getClass()`	Find object class
`clone()`	Copy object
`wait()`	Pause thread
`notify()`	Resume one thread
`notifyAll()`	Resume all threads
`toHexString()`	Int ➔ Hex String