**History of Java**

Java is a high-level, object-oriented programming language developed by James Gosling and his team at Sun Microsystems in the early 1990s. It was officially released in 1995. The goal of Java was to create a platform-independent, secure, and robust programming language.

Java has evolved significantly from a simple object-oriented language to a powerful ecosystem used in web development, enterprise applications, Android development, and cloud computing. With its platform independence and continuous updates, Java remains one of the most widely used programming languages today.

**Java Development Kit (JDK)**

What is JDK?

JDK (Java Development Kit) is a software development kit required to develop Java applications. It provides all the necessary tools, libraries, and components to write, compile, debug, and run Java programs.

JDK is essential for Java development and includes tools for compiling, debugging, and running Java programs.

JDK -> Java Development kit = Collections of tools used for developing and running java programs.

JRE -> Java Runtime Environment = Helps in executing program developed in Java.

Components of JDK

1. Java Compiler (javac)
   * Converts Java source code (.java files) into bytecode (.class files).
   * Example:

javac MyProgram.java

1. Java Runtime Environment (JRE)
   * The JDK includes JRE, which allows Java programs to run.
   * JRE consists of:
     + Java Virtual Machine (JVM) – Executes Java bytecode.
     + Java Standard Libraries – Essential classes like java.lang, java.util, java.io.
2. Java Virtual Machine (JVM)
   * JVM is responsible for running Java bytecode.
   * It provides memory management, garbage collection, and security.
3. Development Tools
   * javac – Java Compiler
   * java – Java application launcher
   * javadoc – Generates documentation from comments
   * jar – Packages compiled classes into JAR files
   * jdb – Java Debugger
   * keytool – Manages security certificates
4. Java APIs (Application Programming Interfaces)
   * JDK provides built-in APIs for developers, including:
     + Core Java (java.lang, java.util, java.io, etc.)
     + Networking APIs (java.net)
     + Database Connectivity (JDBC - java.sql)
     + Concurrency (java.util.concurrent)

**Introduction to Java –**

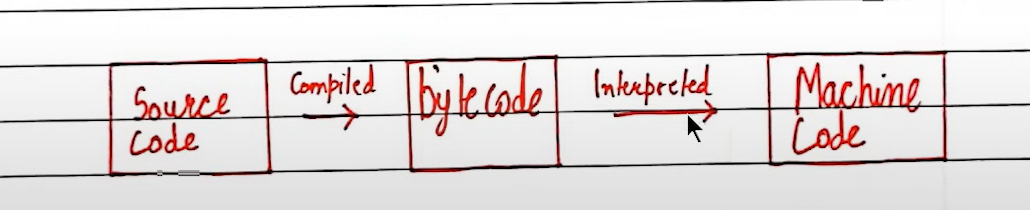
Java is an object oriented programming language developed by sun microsystems of USA in 1991.

It was originally called Oak by James Goslin.

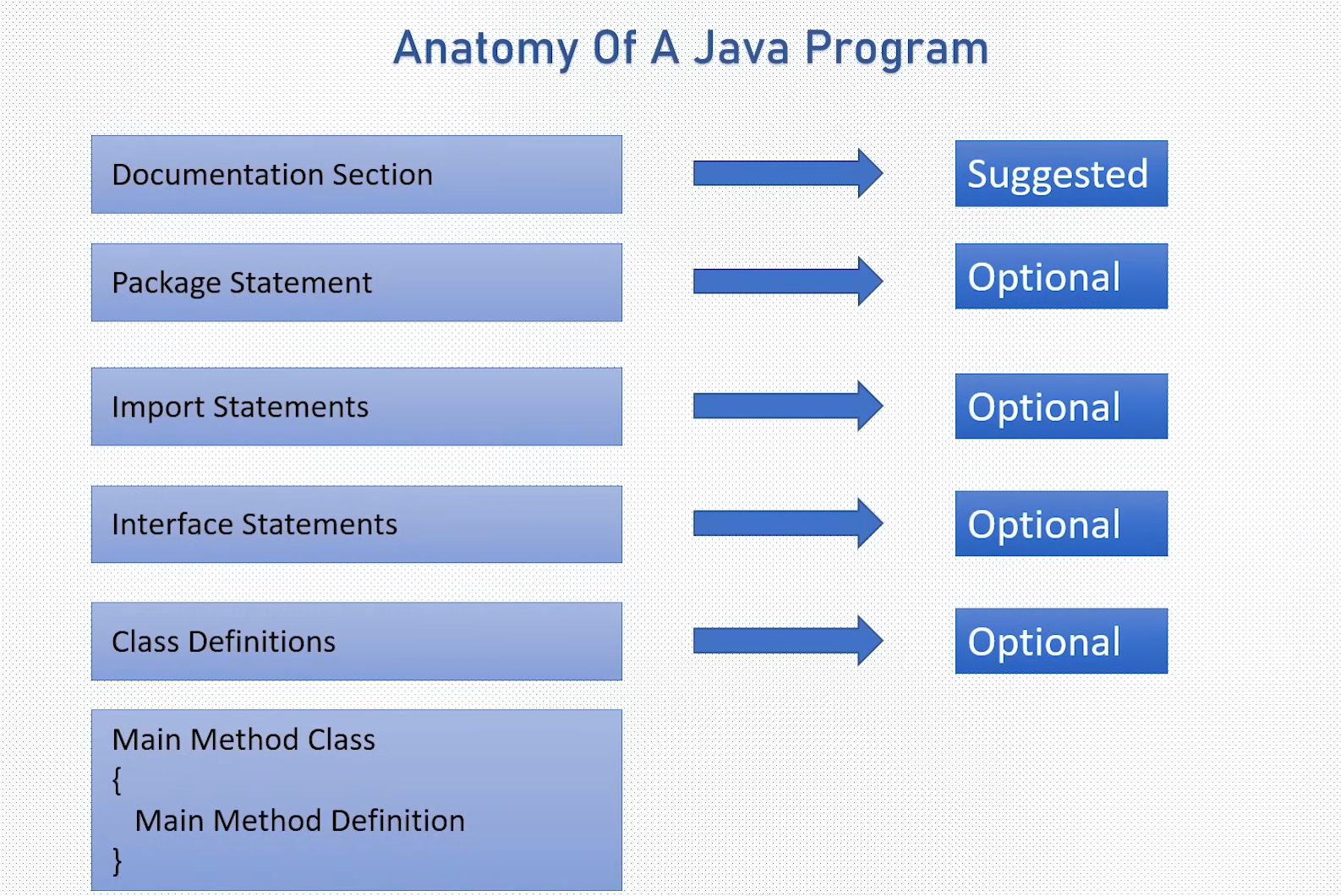
Java is a purely object oriented programming language unlike C++ which can be used both for Procedural (POP) and Object oriented programming(OOP).

How does java works?

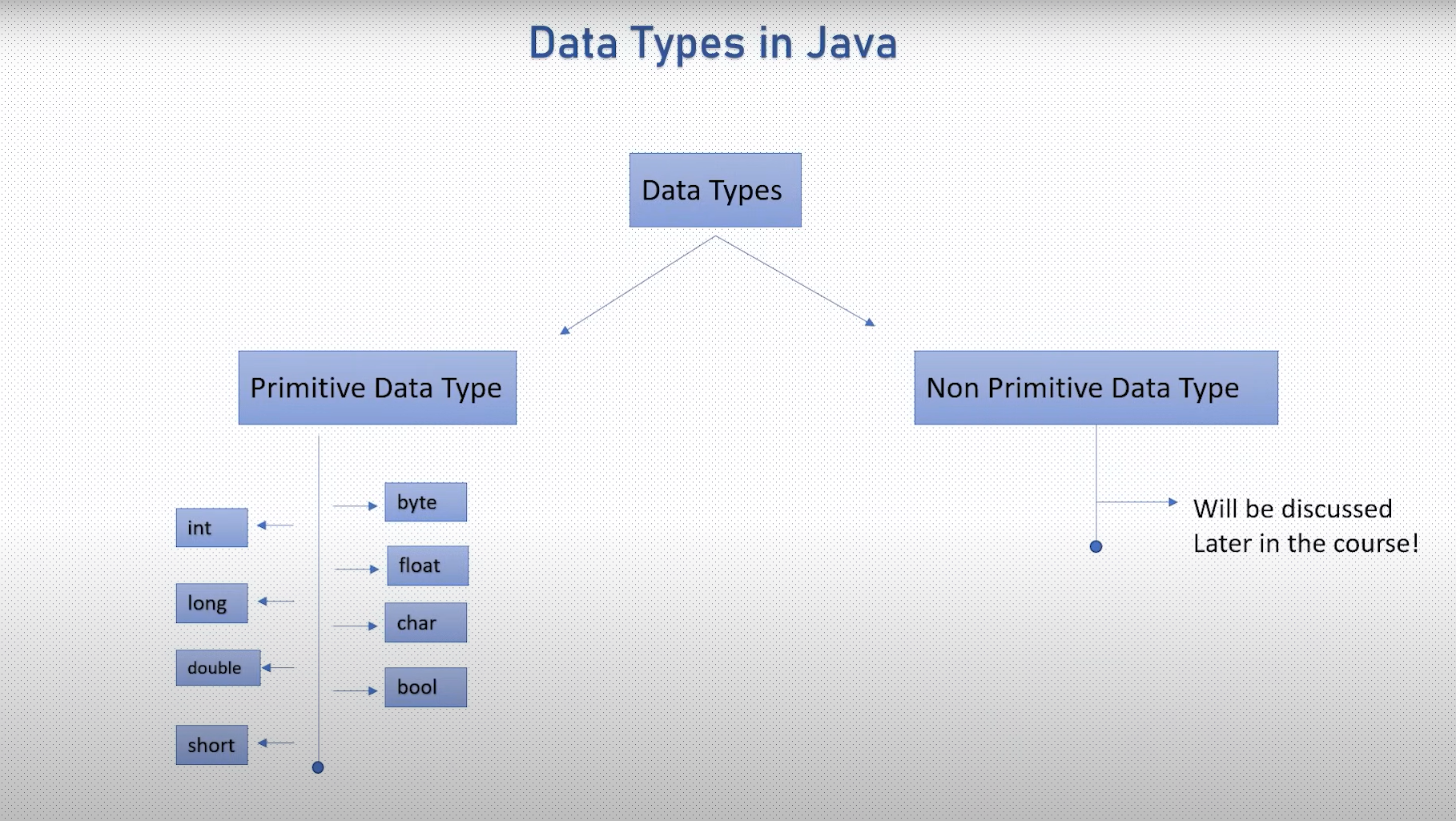
Java is compiled into the bytecode and then it is interpreted to machine code.



**Anatomy of a Java Program -**



**Variables and Data Types in Java**

****

Just like we have some rules that we follow to speak English (the grammar), we have some rules to follow while writing a java program. The set of these rules is called syntax.

**Variables:**

A variable is a container that stores a value. This value can be changed during the execution of the program.

Data Types :

In Java, data types define the kind of values a variable can store. Java has **two categories** of data types:

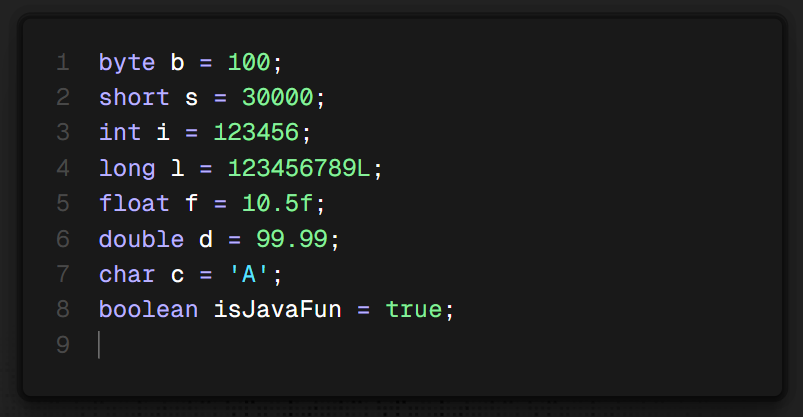
1. **Primitive Data Types** (Built-in types)
2. **Non-Primitive Data Types** (Reference types like Objects, Arrays, and Classes)

**1. Primitive Data Types (8 Types)**

Primitive data types are **basic types** that store simple values directly in memory.

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Size** | **Default Value** | **Description** |
| byte | 1 byte | 0 | Stores small integers (-128 to 127) |
| short | 2 bytes | 0 | Stores larger integers (-32,768 to 32,767) |
| int | 4 bytes | 0 | Stores whole numbers (-2³¹ to 2³¹-1) |
| long | 8 bytes | 0L | Stores very large whole numbers (-2⁶³ to 2⁶³-1) |
| float | 4 bytes | 0.0f | Stores decimal numbers (Single precision) |
| double | 8 bytes | 0.0d | Stores decimal numbers (Double precision) |
| char | 2 bytes | '\u0000' | Stores a single character (Unicode) |
| boolean | 1 bit | false | Stores true or false |

**Examples**

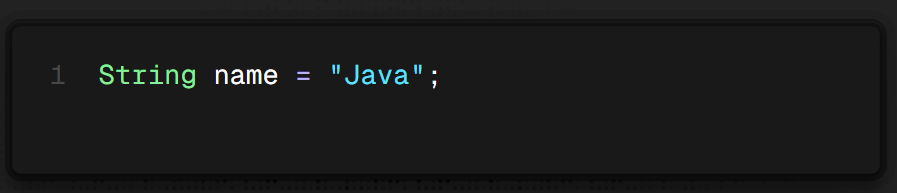


**2. Non-Primitive Data Types**

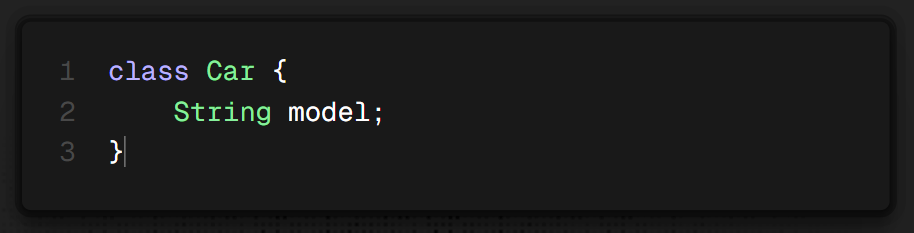
Non-primitive data types store references (memory addresses) rather than actual values.

**Types of Non-Primitive Data Types:**

1. **String** (Stores sequences of characters)



1. **Array** (Stores multiple values of the same type) 
2. **Class** (User-defined data type)



1. **Interface** (Blueprint for classes)



**Key Differences: Primitive vs. Non-Primitive**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Primitive Data Type** | **Non-Primitive Data Type** |
| Storage | Stores actual value | Stores reference (address) |
| Speed | Faster | Slower |
| Default Value | Has default values | Defaults to null |
| Examples | int, char, boolean | String, Array, Class |

**Conclusion**

* **Primitive types** store simple values and are memory-efficient.
* **Non-primitive types** store complex structures like objects and arrays.
* Java is **strongly(statically) typed**, meaning you must declare a variable’s type before using it.

**How to calculate the range of values a variable can hold:**

For example, if you have a data type with a size of 1 byte (which is 8 bits), you can determine its range using the formula:

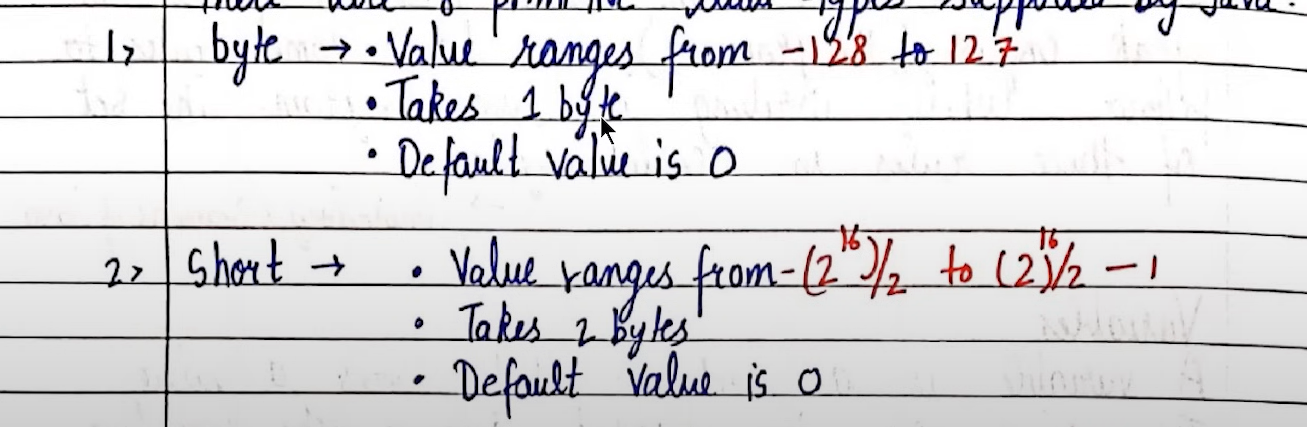
**(-2^(8-1)) to (2^(8-1) - 1)**

So if the data type takes n bytes of space then the formula for range of values can be :

**(-2^(8n-1)) to (2^(8n-1) - 1)**

The exponent is reduced by 1 (i.e., 8-1) because one bit is used for the sign in signed data types. The positive range is further reduced by 1 because zero is also included.

Example :



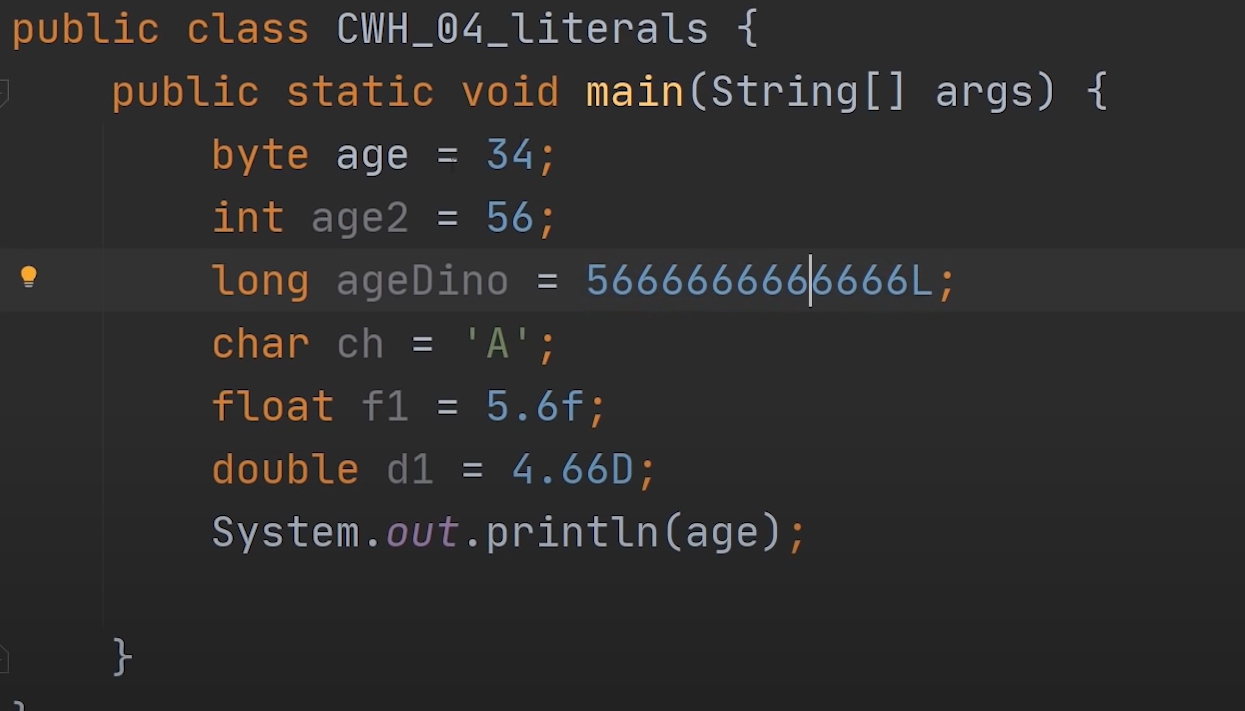
Literals :

A constant value which can be assigned to the variable is called as a literal.

It is just the right hand side value while declaring a variable.

So basically you need to tell the compiler that what type of value are you storing in the variable( used for data types like float, double, long, etc. ) by adding a letter at the end of the value.

Example :



**Difference Between Literals and Data Types in Java**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Literals** | **Data Types** |
| **Definition** | A fixed constant value used in the program. | A classification that specifies the type of data a variable can store. |
| **Purpose** | Represents actual values assigned to variables. | Defines the kind of data that can be stored in a variable. |
| **Examples** | 10, 3.14, 'A', "Hello", true, null | int, float, char, String, boolean, double |
| **Usage** | Used as a value inside the code. | Used to declare variables that can hold different types of values. |
| **Types** | Integer, Floating-Point, Character, String, Boolean, Null | Primitive (int, double, char, etc.) and Non-Primitive (String, Array, Class, etc.) |

**Operator Precedence and Associativity in Java**

In Java, **operator precedence** determines the **order** in which operators are evaluated in an expression.  
If multiple operators have the **same precedence**, then **associativity** decides whether they are evaluated **left to right** or **right to left**.

**Operator Precedence Table in Java**

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator** | **Type** | **Associativity** |
| **1 (Highest)** | () [] . | Parentheses, Array Access, Member Access | **Left to Right** |
| **2** | ++ -- + - ! ~ (type) | Unary, Cast, Logical NOT | **Right to Left** |
| **3** | \* / % | Multiplication, Division, Modulus | **Left to Right** |
| **4** | + - | Addition, Subtraction | **Left to Right** |
| **5** | << >> >>> | Bitwise Shift Operators | **Left to Right** |
| **6** | < <= > >= instanceof | Relational Operators | **Left to Right** |
| **7** | == != | Equality Operators | **Left to Right** |
| **8** | & | Bitwise AND | **Left to Right** |
| **9** | ^ | Bitwise XOR | **Left to Right** |
| **10** | ` | ` | Bitwise OR |
| **11** | && | Logical AND | **Left to Right** |
| **12** | ` |  | ` |
| **13** | ?: | Ternary Conditional | **Right to Left** |
| **14** | = += -= \*= /= %= &= ` | = ^= <<= >>= >>>=` | Assignment Operators |
| **15 (Lowest)** | , | Comma Operator | **Left to Right** |

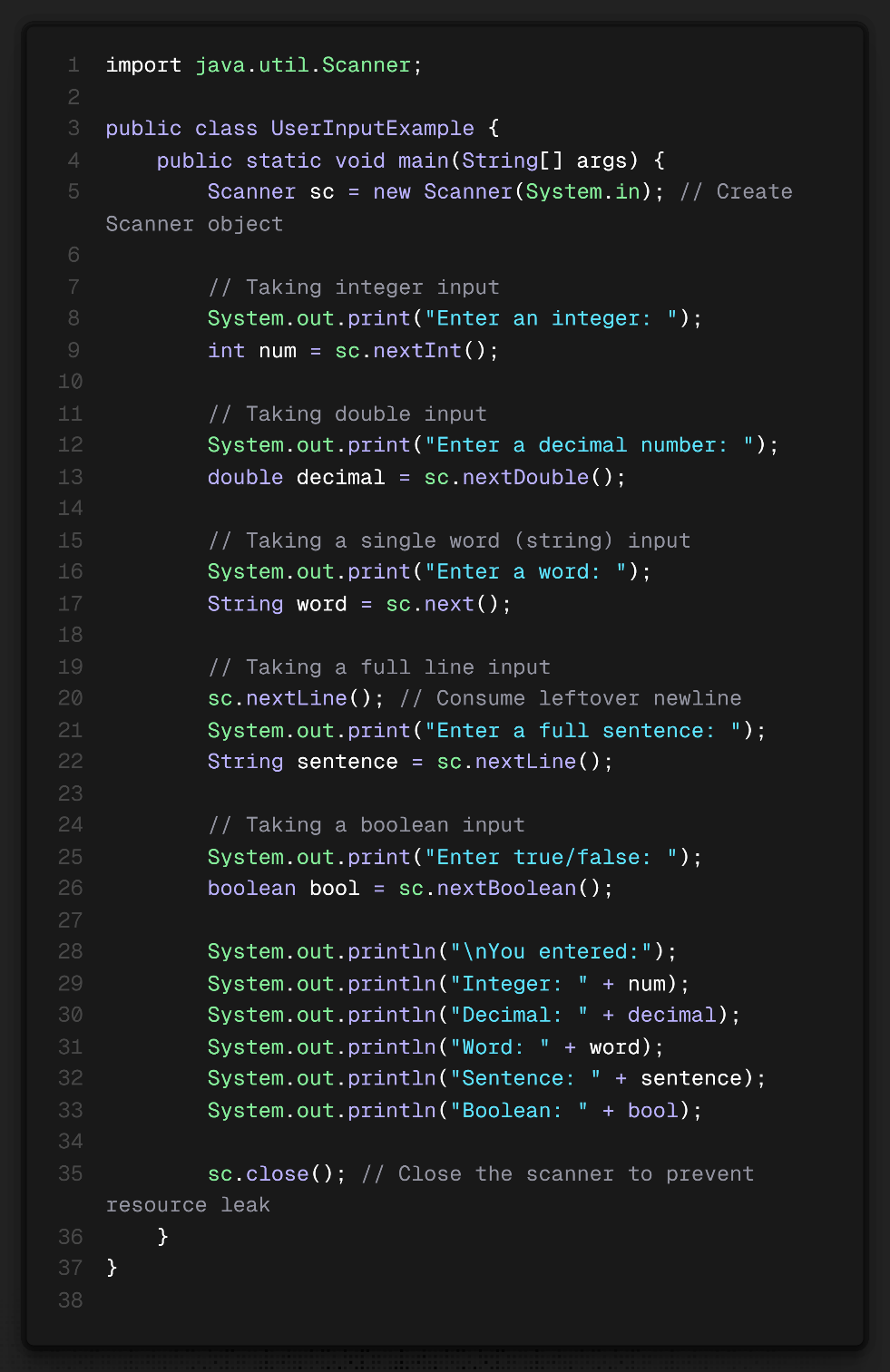
**Taking Input In Java**

In Java you take user input using Scanner class

**1. Using Scanner Class**

The Scanner class (from java.util package) allows us to take input from the user.

**Example: Taking Different Types of Input**



**Scanner Methods**

| **Method** | **Description** |
| --- | --- |
| nextInt() | Reads an integer |
| nextDouble() | Reads a double (decimal) number |
| nextFloat() | Reads a float |
| nextBoolean() | Reads a boolean (true or false) |
| next() | Reads a single word (stops at space) |
| nextLine() | Reads an entire line (useful for sentences) |

**Strings In Java**

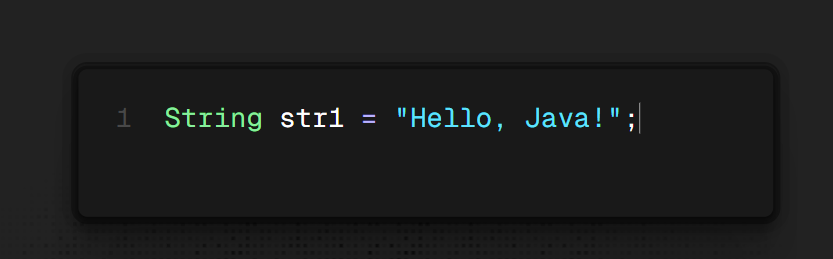
**Strings in Java**

A **String** in Java is a sequence of characters. Java provides **String class** to work with text, which is **immutable** (cannot be changed after creation). Strings in Java are widely used for handling textual data.

**1. Creating Strings in Java**

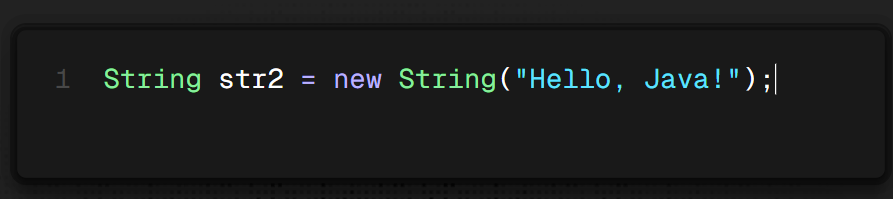
There are **two ways** to create a String in Java:

**1️ Using String Literals (Recommended)**

****

* Stored in **String Pool** (inside Heap Memory).
* If the same string already exists, Java does **not** create a new object (saves memory).

**2️ Using new Keyword (Not Recommended)**

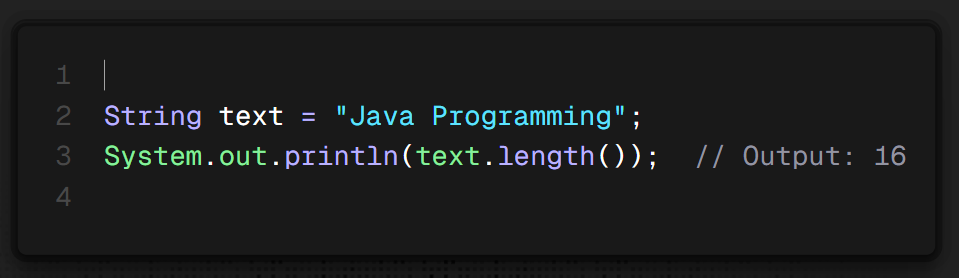
****

* Creates a **new object** in Heap Memory, even if the same string exists in the String Pool.

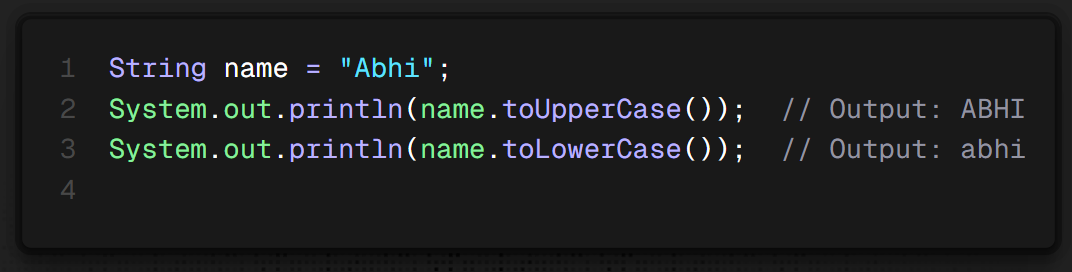
**2. String Methods in Java**

The String class provides many useful methods.

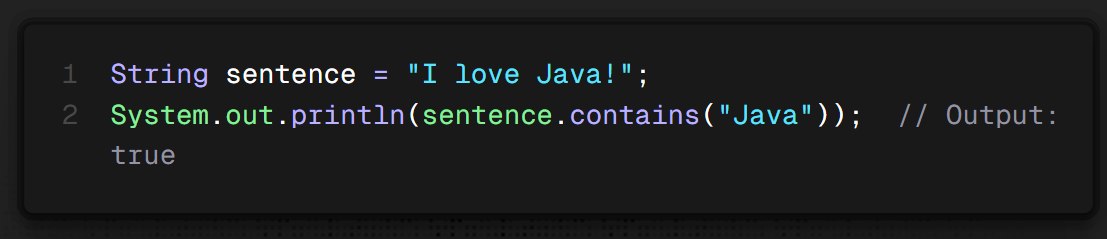
**🔹 String Length**

****

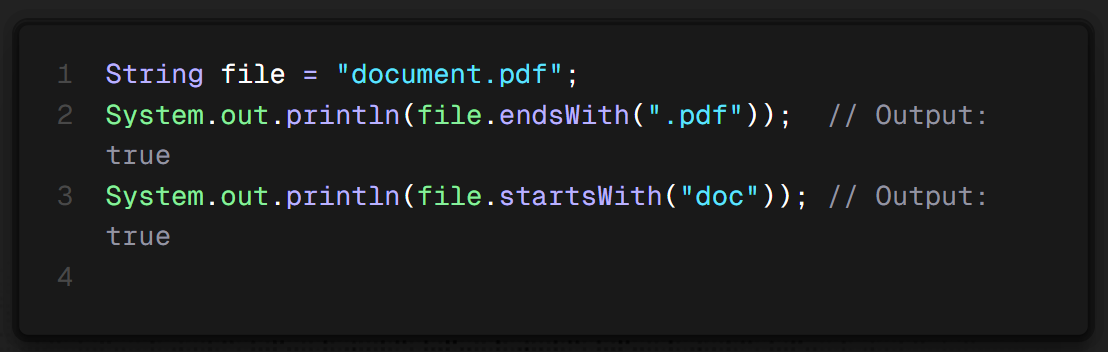
**🔹 Convert to Uppercase / Lowercase**

****

**🔹 Checking if String Contains a Substring**

****

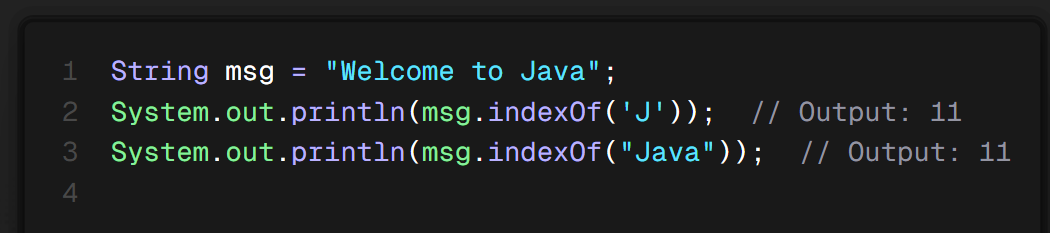
**🔹 Checking if String Starts/Ends with a Specific Substring**



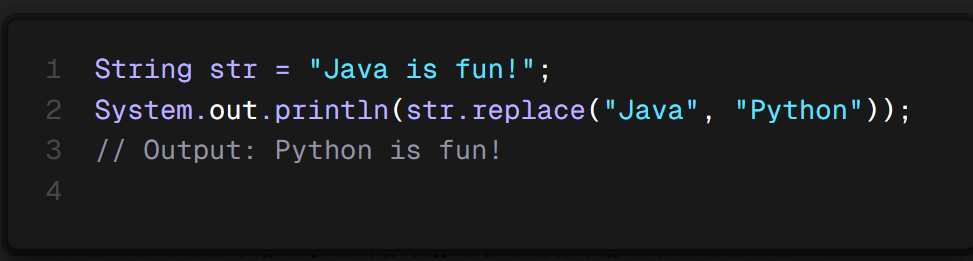
**🔹 Getting Character at a Specific Index**

****

**🔹 Finding Index of a Character or Word**



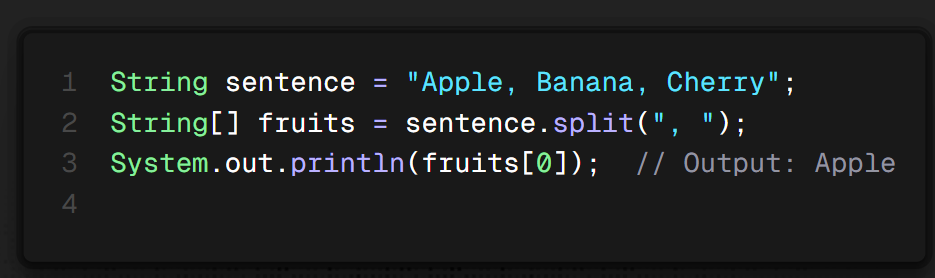
**🔹 Replacing Characters in a String**



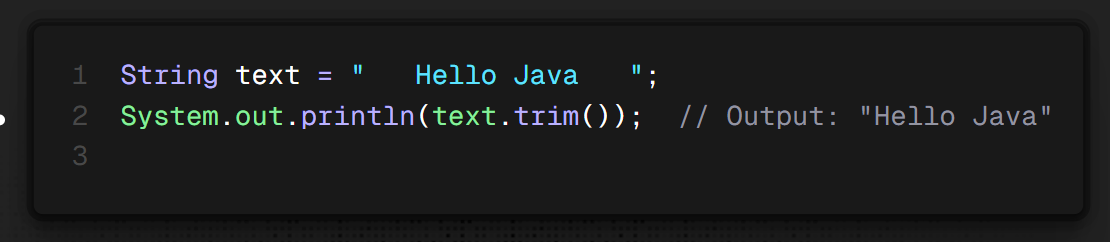
**🔹 Extracting a Substring**



**🔹 Splitting a String**



**🔹 Trimming Spaces**

****

1. **String Immutability in Java**

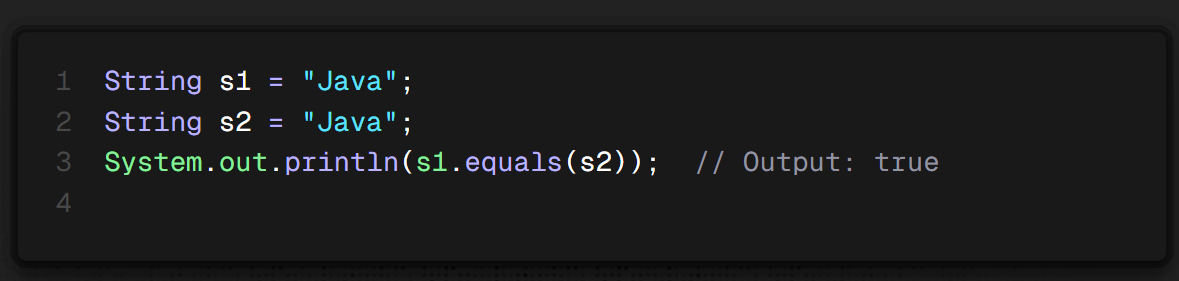


* Strings in Java are **immutable** → Any modification creates a new string, the original remains unchanged.

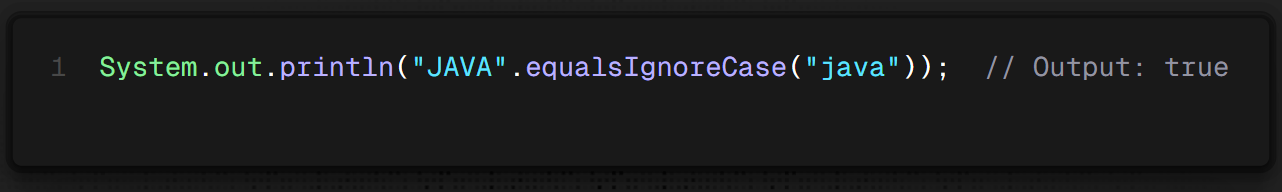
✅ **Solution: Use a StringBuilder** if you need mutable strings.

**4. String Comparison in Java**

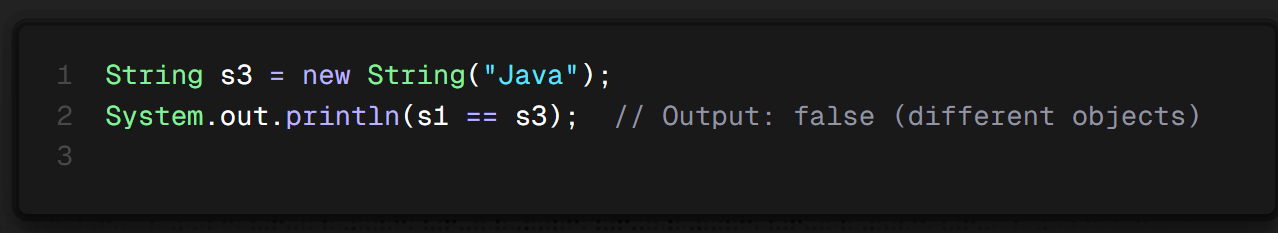
**🔹 Using equals() (Content Comparison)**



🔹 **equalsIgnoreCase()** (Ignores Case)

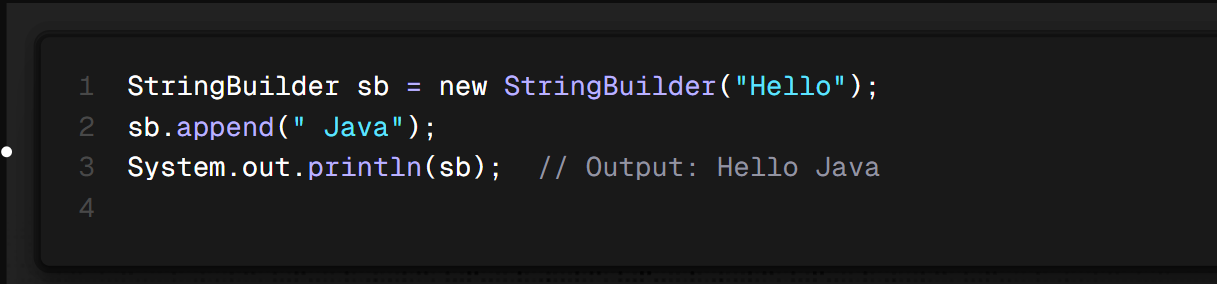


**🔹 Using == (Reference Comparison)**



**5. StringBuilder & StringBuffer (Mutable Strings)**

**🔹 StringBuilder (Fast, Not Thread-Safe)**

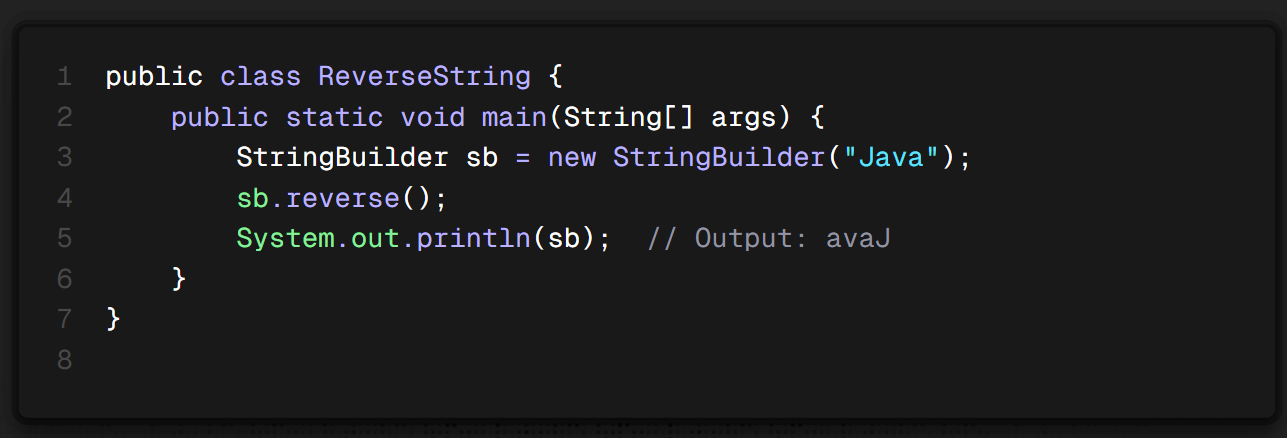


**🔹 StringBuffer (Thread-Safe, Slower)**

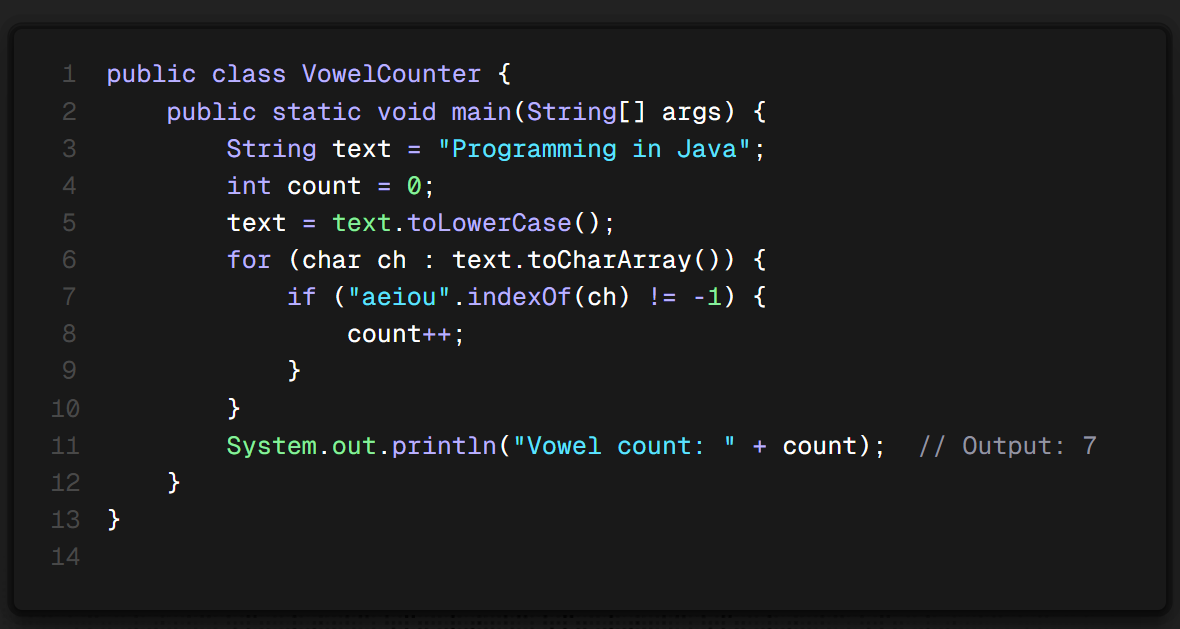


✅ **Use StringBuilder when working with dynamic strings (modifications).**  
✅ **Use String when working with fixed text data.**

**6. Example: Reversing a String**



**7. Example: Counting Vowels in a String**



**🚀 Summary**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **String** | **StringBuilder** | **StringBuffer** |
| **Mutability** | Immutable | Mutable | Mutable |
| **Performance** | Slow (New Object Each Time) | Fast | Slower than StringBuilder |
| **Thread Safety** | Yes | No | Yes |
| **Use Case** | When data doesn't change frequently | When data changes frequently | When thread safety is needed |