**🧠 Quick Overview**  
  
When you run a Java program, it goes like this:

1. Java code (.java) ➝ **Compiled** by javac ➝ Bytecode (.class)
2. Bytecode is run by the **Java Virtual Machine (JVM)**
3. JVM uses both:
   * **Interpreter** to run bytecode line by line
   * **JIT Compiler** to speed up frequently used code

🔄 Interpreter vs JIT: What’s the Difference?

| **Feature** | **Interpreter** | **JIT Compiler** |
| --- | --- | --- |
| ⚙️ Execution | Runs bytecode **line by line** | Converts **hot code** into native machine code |
| 🐢 Speed | Slower, but quick to start | Faster after warm-up |
| 📦 Output | Executes bytecode directly | Produces native code for reuse |
| 🔁 Use Case | Short-lived or rarely used code | Frequently executed code ("hot spots") |

**🔍 How JVM Uses Both Together**

**1. Start with Interpreter**

* When your program starts, the JVM **interprets bytecode** line by line.
* Quick startup, no delay from compilation.

**2. Detect Hot Code (HotSpot JVM)**

* JVM **monitors** which methods or loops are executed frequently.
* These are called **"hot spots"**.

**3. Just-In-Time (JIT) Compilation**

* When a method gets "hot", the JIT compiler:
  + **Compiles** it into native machine code
  + **Caches** it for future calls (runs super fast!)

Now instead of interpreting, the JVM **directly executes the native code**.

**🧬 Real Example Flow**

Let's say this loop runs many times:

for (int i = 0; i < 100000; i++) {

Math.sqrt(i);

}

🔹 JVM starts by interpreting it.  
🔹 After a few thousand iterations, it realizes this code is **hot**.  
🔹 JIT compiles it into fast native machine code.  
🔹 The loop now runs faster using the compiled code.

**🧩 Why This Hybrid Approach Rocks**

* **Interpreter** = Quick startup
* **JIT Compiler** = Long-term performance boost
* JVM adapts **at runtime** based on usage patterns

# **Tokens (Literals, Keywords, Operators, Identifiers)**

**Types of Tokens in Java**

**1. Keywords**

* Reserved words with predefined meanings.
* **Examples**: class, public, static, void, int, if, else, for, while, return, etc.

**2. Identifiers**

* Names given to variables, methods, classes, etc.
* **Rules**: Start with a letter, \_, or $, followed by letters/digits.
* **Examples**: myVariable, calculateSum, Student, MAX\_VALUE, etc.

**3. Literals**

* Fixed values assigned to variables.
* **Types**:
  + **Integer**: 10, -5
  + **Floating-point**: 3.14, -2.5f
  + **Character**: 'A', '\n', '\u0041' (Unicode).
  + **String**: "Hello World".
  + **Boolean**: true, false.
  + **Null**: null.

**4. Operators**

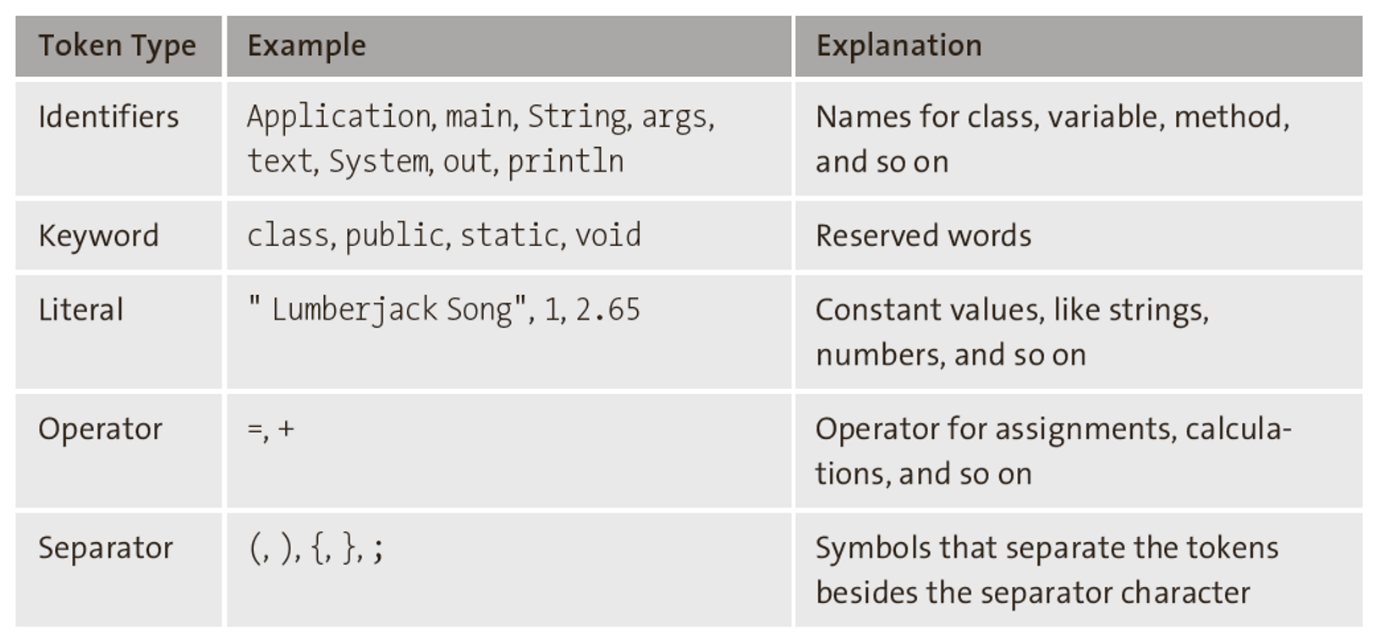
* Symbols that perform operations on operands.
* **Types**:
  + **Arithmetic**: +, -, \*, /, %.
  + **Assignment**: =, +=, -=.
  + **Comparison**: ==, !=, >, <.
  + **Logical**: &&, ||, !.
  + **Bitwise**: &, |, ^, ~.

**5. Separators (Punctuators)**

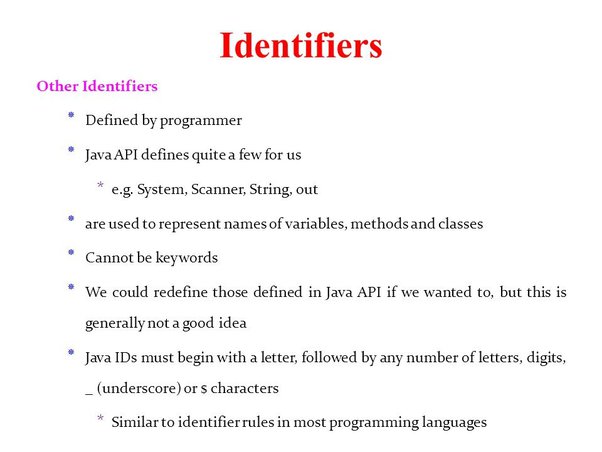
* Symbols that define structure/syntax.
* **Examples**: {}, (), [], ;, ,, ..

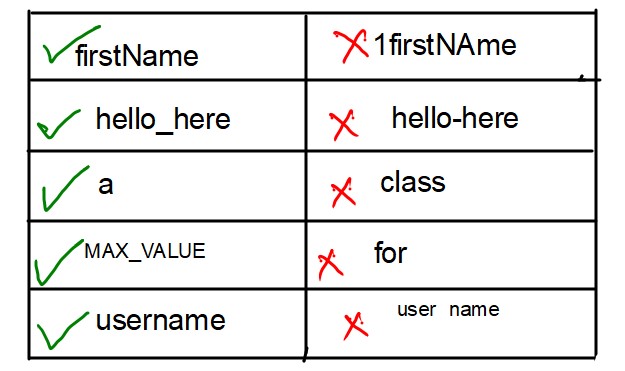
**6. Comments**

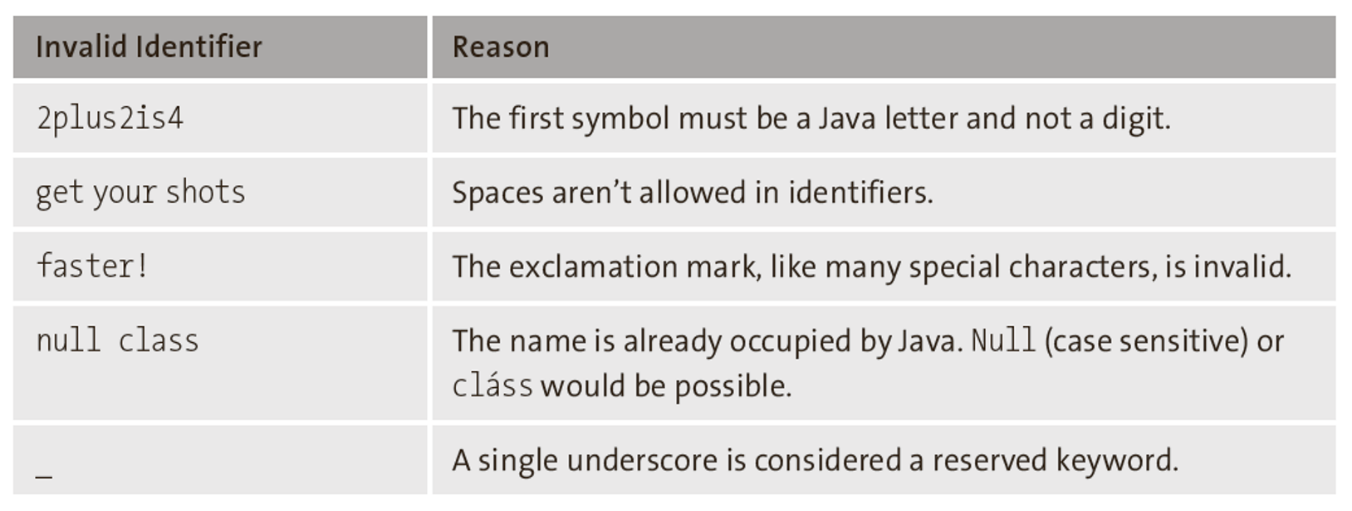
* Text ignored by the compiler (not technically tokens, but part of code).
* **Types**:
  + Single-line: // This is a comment.
  + Multi-line: /\* ... \*/.
  + Documentation: /\*\* ... \*/.



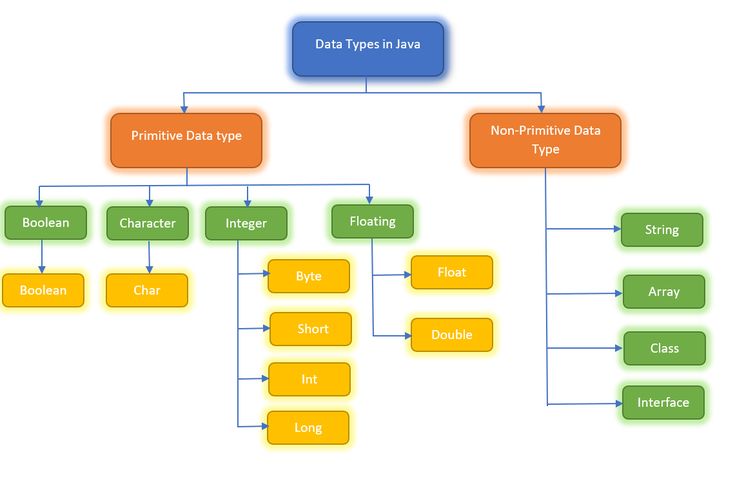


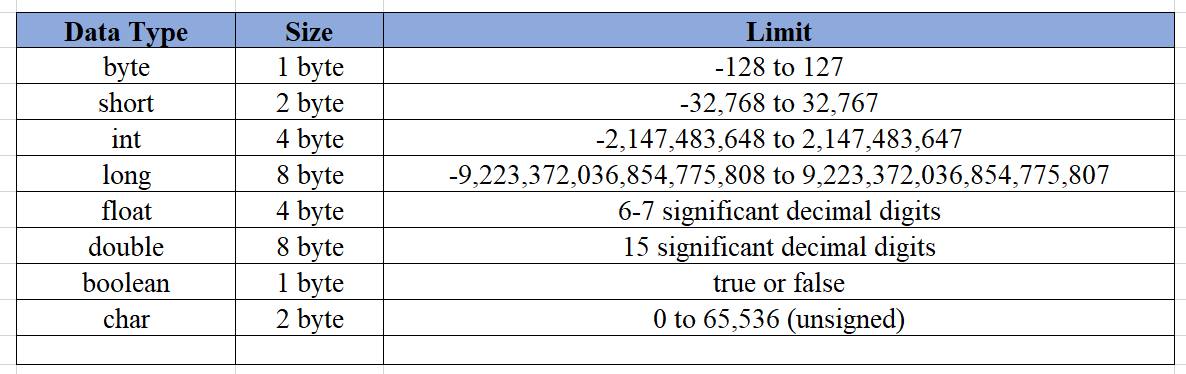






## **Datatypes**



  
**1. Primitive Data Types**

These are the basic building blocks in Java. Java has 8 primitive data types:

* **byte**: An 8-bit integer used for saving memory (e.g., byte age = 25;).
* **short**: A 16-bit integer, often for smaller range values (e.g., short distance = 300;).
* **int**: A 32-bit integer for whole numbers (e.g., int salary = 50000;).
* **long**: A 64-bit integer used for large values (e.g., long population = 7000000000L;).
* **float**: A 32-bit floating point for decimal values (e.g., float price = 19.99f;).
* **double**: A 64-bit floating point for precise decimal values (e.g., double pi = 3.14159265359;).
* **char**: A single 16-bit Unicode character (e.g., char grade = 'A';).
* **boolean**: Represents true or false values (e.g., boolean isJavaFun = true;).

**2. Non-Primitive (Reference) Data Types**

These are more complex types and include classes, interfaces, arrays, and strings:

* **String**: Represents a sequence of characters (e.g., String name = "Java";).
* **Arrays**: Store multiple values of the same type (e.g., int[] numbers = {1, 2, 3, 4};).
* **Classes**: Define objects and methods (e.g., creating a custom class like class Car {}).
* **Interfaces**: Used to specify what a class must do without defining how (e.g., interface Vehicle {}).

| **Data Type** | **When to Use** | **Why to Use** |
| --- | --- | --- |
| **float** | When precision isn't critical (e.g., animations, graphics). | Fast and memory-efficient. |
| **double** | When precision is moderately important (e.g., most scientific computations). | Balances precision and performance. |
| **BigDecimal** | When high precision is mandatory (e.g., currency, banking apps). | Eliminates rounding errors entirely. |

**Special Mention: Var**

var is a **reserved type** in Java that allows the compiler to infer the type of a variable **at compile time**. It was introduced in **Java 10** as part of the **local-variable type inference** feature.

**Is var a Data Type?**

🚫 **No**, var is **not a data type** like int, double, or String.  
✅ Instead, it is a **placeholder** that tells the compiler to infer the type **from the assigned value**.  
  
**Key Point:** var **reduces boilerplate code** but doesn’t make Java dynamically typed. The inferred type **cannot change** after initialization.

| **Rule** | **Description** | **Example** |
| --- | --- | --- |
| **Must be initialized** | var cannot be declared without an initial value. | ❌ var x; // Compilation error |
| **Cannot be null without explicit type** | The compiler cannot infer the type of null. | ❌ var data = null; // Error |
| **Cannot be used as a method parameter** | Method parameters must have explicit types. | ❌ void myMethod(var x) { } // Error |
| **Cannot be used in instance or class variables** | var is only allowed in local variables. | ❌ class Test { var x = 5; } // Error |
| **Works only inside methods, loops, and blocks** | It cannot be used for class-level fields. | ✅ void example() { var count = 10; } |

# **Operators in java**

**What are Operators in Java?**

Operators in Java are **symbols** used to perform operations on variables and values. They are fundamental for writing logic in Java program

| **Operator Type** | **Purpose** |
| --- | --- |
| **Arithmetic Operators** | Perform mathematical operations (+, -, \*, /, %) |
| **Relational (Comparison) Operators** | Compare values (==, !=, >, <, >=, <=) |
| **Logical Operators** | Perform logical operations (&&, |
| **Bitwise Operators** | Work on bits (&, |
| **Assignment Operators** | Assign values (=, +=, -=, \*=, /=) |
| **Unary Operators** | Operate on a single variable (++ and --) |
| **Ternary Operator** | Shortens if-else statements (? :) |

**1. Arithmetic Operators 🔢**

📌 **Used for mathematical calculations in applications like banking, e-commerce, and games.**

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| + | Addition | int sum = a + b; |
| - | Subtraction | int diff = a - b; |
| \* | Multiplication | int product = a \* b; |
| / | Division | int quotient = a / b; |
| % | Modulus (Remainder) | int remainder = a % b; |

**2. Relational (Comparison) Operators 🔍**

📌 **Used in decision-making in applications like authentication, access control, and sorting algorithms.**

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| == | Equal to | if (a == b) |
| != | Not equal to | if (a != b) |
| > | Greater than | if (a > b) |
| < | Less than | if (a < b) |
| >= | Greater than or equal to | if (a >= b) |
| <= | Less than or equal to | if (a <= b) |

**3. Logical Operators ⚡**

📌 **Used in authentication, business logic validation, and conditional checks.**

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| && | Logical AND | if (a > 0 && b > 0) |
| || | Logical OR | if (a > 0 or b > 0) |
| ! | Logical NOT | if (!isLoggedIn) |

**4. Bitwise Operators 🧮**

📌 **Used in low-level programming, encryption, and graphics processing.**

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| & | Bitwise AND | a & b |
| | | | | Bitwise OR |
| ^ | Bitwise XOR | a ^ b |
| ~ | Bitwise Complement | ~a |
| << | Left Shift | a << 2 |
| >> | Right Shift | a >> 2 |

**5. Assignment Operators ✍️**

📌 **Used to assign values in variables in all applications.**

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| = | Assignment | a = 10; |
| += | Add and Assign | a += 5; (Same as a = a + 5) |
| -= | Subtract and Assign | a -= 3; |
| \*= | Multiply and Assign | a \*= 2; |
| /= | Divide and Assign | a /= 4; |

**6. Unary Operators 🎯**

📌 **Used in loops, counters, and optimizations.**

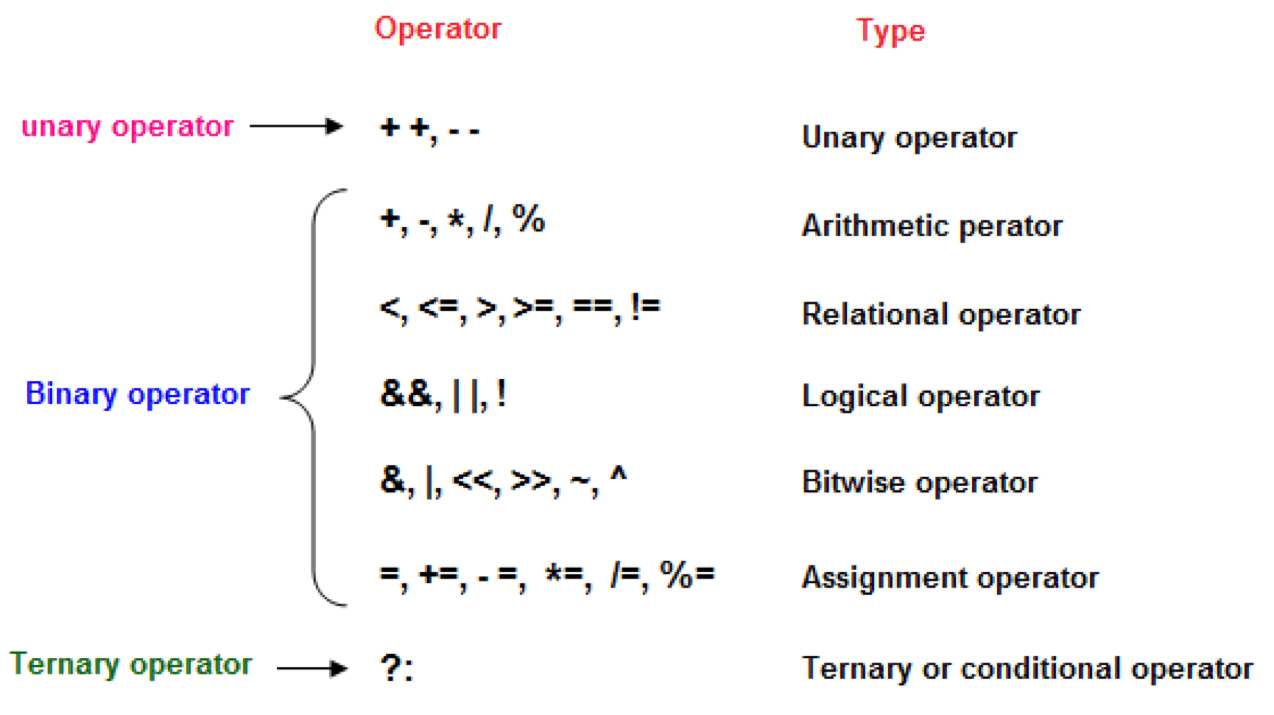
| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| ++ | Increment | a++ (Post) or ++a (Pre) |
| -- | Decrement | a-- or --a |

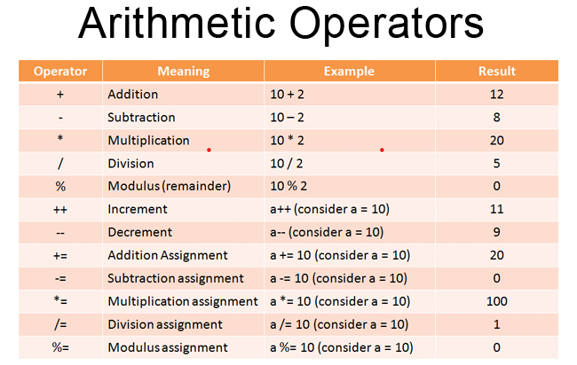
**7. Ternary Operator (? :) 🏷️**

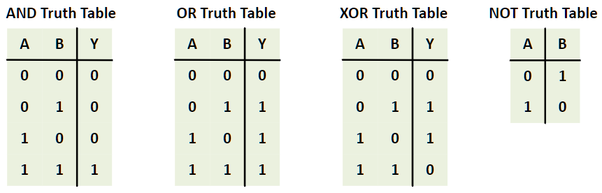
📌 **Used for simple if-else statements.**

**Real-World Usage**:

* String status = (marks >= 40) ? "Pass" : "Fail";
* discount = (isMember) ? 10 : 5;





**Truth table for reference**  


**Operator Precedence in Java**

Operator precedence determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before those with lower precedence. If operators have the **same precedence**, they are evaluated based on their **associativity** (left-to-right or right-to-left).

**Associativity** determines the order in which operators of the **same precedence** are evaluated when they appear in an expression. It can be either:

1. **Left-to-Right** (Most operators)
2. **Right-to-Left** (Some operators like assignment, unary, ternary)

**Java Operator Precedence Table (Highest to Lowest)**

| **Category** | **Operators** | **Associativity** |
| --- | --- | --- |
| **Postfix** | expr++, expr-- | Left-to-right |
| **Unary** | ++expr, --expr, +expr, -expr, ~, ! | Right-to-left |
| **Multiplicative** | \*, /, % | Left-to-right |
| **Additive** | +, - | Left-to-right |
| **Shift** | <<, >>, >>> | Left-to-right |
| **Relational** | <, >, <=, >=, instanceof | Left-to-right |
| **Equality** | ==, != | Left-to-right |
| **Bitwise AND** | & | Left-to-right |
| **Bitwise XOR** | ^ | Left-to-right |
| **Bitwise OR** | | | Left-to-right |
| **Logical AND** | && | Left-to-right |
| **Logical OR** | || | Left-to-right |
| **Ternary** | ? : | Right-to-left |
| **Assignment** | =, +=, -=, \*=, /=, %=, etc. | Right-to-left |

**Example 1: Arithmetic Operators**

int result = 5 + 3 \* 2;

* **Evaluation:**  
  3 \* 2 (Multiplication first, higher precedence than +) → 6  
  5 + 6 → 11
* **Output:** 11

**Example 2: Postfix vs. Unary**

int x = 5;

int y = x++ + ++x;

* **Evaluation:**  
  x++ (post-increment, returns 5, then x becomes 6)  
  ++x (pre-increment, x becomes 7, returns 7)  
  5 + 7 = 12
* **Output:** y = 12, x = 7

**Example 3: Logical Operators**

boolean a = true, b = false, c = true;

boolean res = a || (b && c);

* **Evaluation:**  
  && has higher precedence than ||  
  b && c → false && true → false  
  a || false → true || false → true
* **Output:** res = true

**Example 4: Ternary Operator (Right-to-left Associativity)**

int x = 10, y = 5;

int z = x > y ? x : y > 0 ? y : 0;

* **Evaluation:**  
  Rightmost ? : is evaluated first due to **right associativity**:  
  y > 0 ? y : 0 → 5 > 0 → 5  
  Now: x > y ? x : 5 → 10 > 5 → 10
* **Output:** z = 10

**Example 5: Assignment Operators (Right-to-left Associativity)**

int a = 5;

a += a \*= 2;

* **Evaluation:**  
  a \*= 2 → a = 5 \* 2 → a = 10  
  Now: a += 10 → a = 10 + 10 → 20
* **Output:** a = 20

**Why Does Associativity Matter?**

* **Left-to-right** is intuitive (like reading math).
* **Right-to-left** is used for assignments (a = b = 5) and unary operations (!~x).
* **Mistakes happen if ignored!**

**Common Pitfall**

int x = 5;

int y = x++ + x++ + x++;  
  
5 + 6 + 7

* **Left-to-right evaluation**:  
  (x++ = 5, x=6) + (x++ = 6, x=7) + (x++ = 7, x=8) → 5 + 6 + 7 = 18
* **Final x = 8, y = 18**

**Key Takeaways**

1. **Parentheses () override precedence** (always evaluated first).
2. **Postfix (x++) has higher precedence than prefix (++x)**.
3. **\*, /, % have higher precedence than +, -**.
4. **Logical AND (&&) has higher precedence than Logical OR (||)**.
5. **Ternary (? :) and Assignment (=) operators have right-to-left associativity**.

**Rule of Thumb**

**When in doubt, use parentheses () to make the order explicit!**  
Example:

a=1, b=2,c=1

**✅ Guess the Output – Java Operator Exercises**

**1] Pre-increment vs Post-increment**

int a = 5;

int b = a++;

int c = ++a;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

**2]** **Operator Precedence**

int result = 10 + 20 \* 2 / 4;

System.out.println("Result = " + result);

**3] Compound Assignment**

int x = 10;

x += 5 \* 2;

System.out.println("x = " + x);

**4] Relational + Logical Operators**

int age = 20;

boolean result = (age >= 18 && age < 60);

System.out.println(result);

**5] Integer Division**

int a = 7;

int b = 2;

System.out.println(a / b);

System.out.println(a % b);

**6] Ternary Operator**

int marks = 75;

String grade = (marks >= 90) ? "A" : (marks >= 60 ? "B" : "C");

System.out.println("Grade: " + grade);

**7] Logical NOT Operator**

boolean isRaining = false;

System.out.println("Stay inside? " + !isRaining);

**8] Chained Assignment**

int a, b, c;

a = b = c = 10 + 5;

System.out.println(a + " " + b + " " + c);

**9] Char Arithmetic**

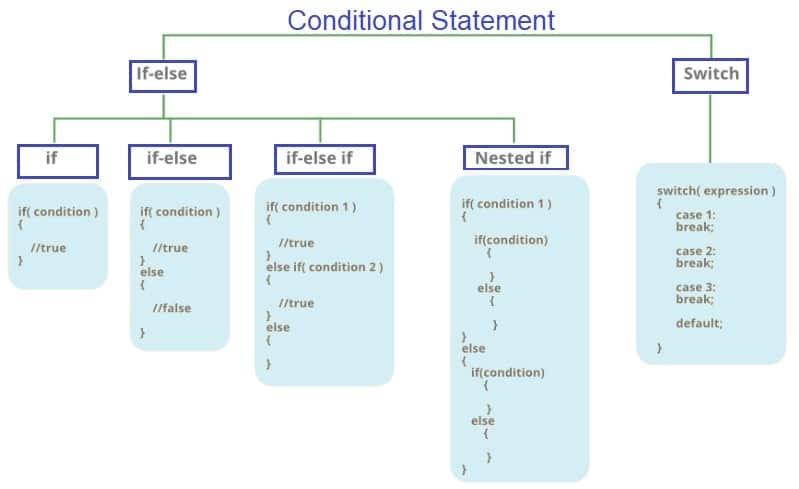
char ch = 'A';

ch = (char)(ch + 3);

System.out.println(ch);

## **Introduction to Conditional statements in java**

Conditional statements allow Java programs to **make decisions** based on certain conditions. They are essential for **controlling program flow** and executing different code blocks based on different scenarios.



**🚀 Summary Table**

| **Statement** | **Use Case** |
| --- | --- |
| **if** | Execute a block **only if** a condition is **true** |
| **if-else** | Execute one block if **true**, another if **false** |
| **if-else-if** | Check **multiple** conditions sequentially |
| **nested if** | Check **sub-conditions** within a condition |
| **switch-case** | Replace multiple if-else checks for **fixed values** |

## **Assignments for this week:**

**1. Basic Arithmetic Calculator**📌 **Concepts:** Arithmetic operators, input handling  
🔹 **Task:** Write a Java program that takes two numbers and an operator (+, -, \*, /, %) as input and performs the corresponding arithmetic operation.  
  
**Input:**  
Enter first number: 10

Enter second number: 5

Enter operator (+, -, \*, /, %): \*   
  
(Inputs 10, 5 & \* are being entered by user. Program should take input from user for first, second number and operator)  
  
**Output:**

Result (10 \* 5) = 50

**2. Temperature Converter**

📌 **Concepts:** Data types, arithmetic operations  
🔹 **Task:** Convert temperature from **Celsius to Fahrenheit** and vice versa.  
🔹 **Formula:**

* Fahrenheit = (Celsius × 9/5) + 32
* Celsius = (Fahrenheit - 32) × 5/9

**Input:**Enter temperature: 100

Choose conversion (C -> F or F -> C): C  
  
(Inputs 100 & C are being entered by user)  
  
**Output:**  
Fahrenheit: 212°F

**3. Check Leap Year**

📌 **Concepts:** Logical & relational operators  
🔹 **Task:** Write a program to check if a year is **leap year** or not.  
🔹 **Leap Year Conditions:**

* **Divisible by 4**
* **Not divisible by 100**, unless also divisible by 400

**Input:**Enter year: 2024  
(Input 2024 is being entered by user)

**Output:**  
2024 is a leap year.   
  
**4. Student Grade Calculator**

📌 **Concepts:** Arithmetic & relational operators, conditional statements  
🔹 **Task:** Accept marks for **5 subjects**, calculate percentage, and assign a grade:

* >= 90 → A
* 80-89 → B
* 70-79 → C
* < 70 → Fail

**Input:**Enter marks in 5 subjects: 85 90 78 88 92  
(Input 85, 90, 78, 88, 92 separated by spaces is entered by user)  
**Output:**  
Grade : B