**JAVA Basics to Advanced 2025**

**Variables:**

Variable is a place or a box to store data.

Data can be of different types

* Numbers
* Strings
* True or false
* image

We have to give a name to variable to create/define it.

```

myName = “Hello”;

myAge = 33;

avgMark = 75.8;

```

**Variable Naming Conventions:**

While you name or create a variable it's important to follow certain naming conventions to make your code readable and maintainable:

* Variable names should be descriptive and indicate the purpose of the variable.
* They must start with a letter, underscore (`\_`), or dollar sign (`$`). They cannot start with a digit.
* They can contain letters, digits, underscores, and dollar signs.
* Java is case-sensitive, so `count` and `Count` are different variables.
* By convention, variable names are usually written in `camelCase`, where the first word is in lowercase, and subsequent words start with an uppercase letter (e.g., `firstName`, `totalAmount`).
* Avoid using Java keywords (reserved words like `class`, `public`, `int`, etc.) as variable names.

**Data Types:**

We saw we can store data in variables and it can be of different types.

So, in java the data types are categorized in to two primitive and reference.

**Primitive data types:**

These data types has a fixed size and range. And stores actual values in memory.

* Byte: **byte**
  + Byte has a fixed size of 8 bits and stores whole numbers between -128 to 127.
* Shorst: **short**
  + Short has a fixed size of 16 bits and stores whole numbers between -32,768 to 32,767
* Integer: **int**
  + Integer has a fixed size of 32 bits and stores whole numbers between -2,147,483,648 to 2,147,483,647
* Long: **long**
  + Long has a fixed size of 64 bits and stores large whole numbers between -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
* Float: **float**
  + Float has a fixed size of 32 bits and stores single-precision floating-point numbers approximately $\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{+38}$
* Double: **double**
* Double has a fixed size of 64 bits and stores double-precision floating-point numbers approximately $\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{+308}$
* Boolean: **boolean**
  + Boolean has a fixed size of 1bit and stores `true` or `false` values.
* Character: **char**
  + Character has a fixed size of 16 bits and stores a single Unicode character range between '\u0000' to '\uffff' (0 to 65,535 in decimal)

**Examples:**

```java

byte age = 30;

short year = 2025;

int count = 100000;

long distance = 15000000000L; // 'L' suffix indicates a long literal

float pi = 3.14159f; // 'f' suffix indicates a float literal

double gravity = 9.81;

boolean isJavaFun = true;

char initial = 'J';

```

**Reference data types are:**

These data types do not store the actual value directly in memory. Instead, they store a reference (an address in memory) to the object that holds the actual data. Reference types include:

**Classes**: Blueprints for creating objects (e.g., `String`, `Scanner`, custom classes you define).

**Interfaces**: Contracts that define a set of methods that a class can implement.

**Arrays**: Collections of elements of the same data type.

**Key Points about Reference Types:**

- Their size is not fixed and depends on the size of the object they refer to.

- They are passed by reference. When you assign a reference variable to another, both variables will refer to the same object in memory.

**Examples:**

```java

String name = "Java"; // String is a class

int[] numbers = {1, 2, 3, 4, 5}; // Array of integers

Scanner scanner = new Scanner(System.in); // Scanner is a class

```

**Creation of a variable:**

There are twoparts in creating a variable, one is declaration and other is initialization.

When you are defining a variable you have to add the data type first followed by variable name then equal sign then variable itself ending with a semicolon.

```

int age = 30;

float hai = 22.3f;

double percentage = 82.55

boolean isPresent = true;

String myName = “Hello”

```

You can first declare and then initialize or do it both at the same time is allowed.

Example:

```

int rollNo; // declaration

rollNo = 24; // initialization

int rollNo = 28; both at the same time

```

Primitive data types are also called literals.

**Type conversion and casting**

Converting or changing the data type of a variable is not allowed but casting is.

Bigger size data type cannot be converted to smaller size but smaller size to bigger size can.

Example

```

byte b = 127;

int a = 1924;

b = a; // not allowed

a = b; // implicit conversion. This kind of assignment is ok

b = (byte) a; // explicit conversion called casting

boolean c = true;

char d = ‘G’;

int e = 7;

c = d; // not allowed

e = c; // not allowed

float f = 7.4f;

int h = (int) f; // ecplicit casting. Will lose the point value.

int a = 257;

byte b;

b = (byte) a; // do a modulus operation and convert and assign the result to it

// Result = 1

byte a = 10;

byte b = 40;

int result = a\*b; // this is called type promotion

```

**Operators**

Operators are special symbols that perform operations on one or more operands (variables or values). Java provides a rich set of operators to perform various kinds of operations. We can categorize them as follows:

1. **Arithmetic Operators**
2. **Relational operators**
3. **Logical operators**
4. **Assignment operators**
5. **Bitwise operator**
6. **Ternary operator**

**Arithmetic Operators:**

**```**

int a = 5;

int b = 8;

int result = a + b; // addition

int result = a - b; // subtraction

int result = a \* b; // multiplication

int result = a / b; // division

int result = a % b; // modulus

int result = a++; // increment (postfix)

int result = ++a; // increment (prefix)

int result = --b; // decrement (prefix)

int result = b--; // decrement (postfix)

int a = 5;

int b = a++; // b becomes 5, and then a becomes 6

System.out.println("a: " + a + ", b: " + b); // Output: a: 6, b: 5

int c = 10;

int d = ++c; // c becomes 11, and then d becomes 11

System.out.println("c: " + c + ", d: " + d); // Output: c: 11, d: 11

**```**

**Increment and Decrement Operators:**

* Postfix (x++, y--): The value of the variable is used in the current operation before it is incremented or decremented.
* Prefix (++x, --y): The value of the variable is incremented or decremented before it is used in the current operation.

**Relational Operators:**

These operators compare two operands and return a boolean value (true or false).

```Java

int x = 5;

int y = 10;

boolean result = x == y; // equal to. Result: false

boolean result = x != y; // not equal to. Result: true

boolean result = x > y; // greater than. false

boolean result = x < y; // less than. Result: true

boolean result = x <= y; // less than or equal to. Result: true

boolean result = x >= y; // greater than or equal to. Result: true

```

**Logical Operators:**

```Java

boolean x = true;

boolean y = false;

boolean result = x && y; // logical and. result: false

boolean result = !x; // logical not. result: false

boolean result = x || y; // logical or. result: false

```

**Short-circuiting in && and ||:**

* In x && y, if x is false, then y is not evaluated because the result will be false regardless of the value of y.
* In x || y, if x is true, then y is not evaluated because the result will be true regardless of the value of y.

**Assignment Operators:**

* These operators assign a value to a variable. The most common one is the simple assignment operator (=).
* Java also provides shorthand assignment operators that combine an arithmetic operation with assignment.

```Java

//Simple assignment

int x = 10; // equivalent to x = 10

//Add and Assign

x += 5; // equivalent to x = x + 5

//subtract and assign

x -= 5; // equivalent to x = x – 5

//Multiply and assign

x \*= 5; // equivalent to x = x \* 5

//Divide and assign

x /= 5; // equivalent to x = x / 5

//Modulus and assign

x %= 5; // equivalent to x = x % 5

```

**Bitwise Operators:**

These operators perform operations on the individual bits of integer operands (byte, short, int, long).

Example (if a=5 (0101), b=3 (0011))

```Java

a & b // Bitwise AND. result binary 0001. Result (decimal) 1

` // Bitwise OR. result binary `a \. Result (decimal) b`

a ^ b // Bitwise XOR. result binary 0110. Result (decimal) 6

~a // Bitwise NOT. result binary 11111...1010 (depends on bit length). Result (decimal) -6

a << 2 // Left shift. result binary 010100...1010 Result (decimal) 20

a >> 1 // Right shift. result binary 0010...1010 Result (decimal) 2

a >>> 1 // Unsigned right shift. result binary 0010...1010 Result (decimal) 2

```

**Understanding Bitwise Shifts:**

* **Left Shift (<<):** Shifts the bits to the left by the specified number of positions. Each left shift effectively multiplies the number by 2.
* **Right Shift (>>):** Shifts the bits to the right by the specified number of positions. The sign bit (most significant bit) is preserved. Each right shift effectively divides the number by 2 (integer division).
* **Unsigned Right Shift (>>>):** Shifts the bits to the right by the specified number of positions, and the leftmost bits are filled with zeros, regardless of the original sign bit.

**Ternary Operator (Conditional Operator):**

This is a shorthand for a simple if-else statement. Its syntax is:

```Java

// condition ? value\_if\_true : value\_if\_false;

int num1 = 10; int num2 = 5; int max = (num1 > num2) ? num1 : num2; // If num1 > num2, max = num1, else max = num2 System.out.println("The maximum is: " + max); // Output: The maximum is: 10

```

**Operator Precedence:**

Operators have a defined precedence that determines the order in which they are evaluated in an expression. It's often a good practice to use parentheses to explicitly control the order of operations and make your code clearer. Here's a general order of precedence (from highest to lowest):

1. Postfix increment/decrement (x++, x--)
2. Prefix increment/decrement (++x, --x), Unary plus/minus (+x, -x), Logical NOT (!), Bitwise NOT (~)
3. Multiplication (\*), Division (/), Modulus (%)
4. Addition (+), Subtraction (-)
5. Shift operators (<<, >>, >>>)
6. Relational operators (<, >, <=, >=), instanceof
7. Equality operators (==, !=)
8. Bitwise AND (&)
9. Bitwise XOR (^)
10. Bitwise OR (|)
11. Logical AND (&&)
12. Logical OR (||)
13. Ternary operator (?:)
14. Assignment operators (=, +=, -=, \*=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=)

Understanding operators and their precedence is crucial for writing correct and predictable Java code.

**Control Flow**

Java provides several control flow statements that allow you to control the sequence of execution in your program. These can be broadly categorized into:

**Conditional Statements:** These statements allow you to execute different blocks of code based on whether a certain condition is true or false.

* **if statement:** Executes a block of code if a specified condition is true.

```Java

int age = 20;

if (age >= 18) { System.out.println("You are eligible to vote."); }

```

* **if-else statement:** Executes one block of code if a condition is true and another block if the condition is false.

```Java

int temperature = 15;

if (temperature > 20) {

System.out.println("It's warm.");

} else {

System.out.println("It's cool.");

}

```

* **if-else if-else statement:** Allows you to check multiple conditions in sequence. The first condition that evaluates to true will have its corresponding block of code executed. The final else block (if present) is executed if none of the preceding conditions are true.

```Java

int score = 75;

if (score >= 90) { System.out.println("Grade: A"); } else if (score >= 80) { System.out.println("Grade: B"); } else if (score >= 70) { System.out.println("Grade: C"); } else if (score >= 60) { System.out.println("Grade: D"); } else { System.out.println("Grade: F"); }

```

* **switch statement:** Provides a way to execute different blocks of code based on the value of a single variable (or expression). The variable must be of a primitive type (byte, short, int, char), an enum, or a String (since Java 7).

```Java

int dayOfWeek = 3;

String dayName;

switch (dayOfWeek) {

case 1:

dayName = "Monday";

break; // Important to exit the switch block

case 2:

dayName = "Tuesday";

break;

case 3:

dayName = "Wednesday";

break;

case 4:

dayName = "Thursday";

break;

case 5:

dayName = "Friday";

break;

case 6:

dayName = "Saturday";

break;

case 7:

dayName = "Sunday";

break;

default:

dayName = "Invalid day";

break;

}

System.out.println("Day " + dayOfWeek + " is " + dayName);

```

**Key points about switch statements:**

* + The case labels must be constant expressions of the same type as the switch variable.
  + The break statement is crucial to exit the switch block after a matching case is found. If break is omitted, execution will "fall through" to the next case. This can sometimes be intentional but is often a source of bugs.
  + The default case is optional and is executed if none of the case values match the switch variable's value. It's good practice to include a default case to handle unexpected values.

**Looping Statements:** These statements allow you to execute a block of code repeatedly as long as a certain condition is true.

**for loop:** Used when you know in advance how many times you want to execute a block of code. It consists of three parts:

* **Initialization:** Executed only once at the beginning of the loop. Typically used to initialize a loop counter.
* **Condition:** Evaluated before each iteration. The loop continues as long as the condition is true.
* **Increment/Decrement (Update):** Executed after each iteration. Typically used to update the loop counter.

```Java

for (int i = 0; i < 5; i++) {

System.out.println("Iteration: " + i);

}

```

You can also have more complex for loops with multiple initializations or updates, but it's generally better to keep them simple for readability.

**Enhanced for loop (for-each loop):** This is a simplified way to iterate over collections (like arrays and lists).

```Java

int[] numbers = {1, 2, 3, 4, 5};

for (int number : numbers) {

System.out.println("Number: " + number);

}

```

**while loop:** Executes a block of code as long as a specified condition is true. The condition is checked *before* each iteration.

```Java

int count = 0;

while (count < 3) {

System.out.println("Count is: " + count);

count++;

}

```

**do-while loop:** Similar to the while loop, but the condition is checked *after* each iteration. This guarantees that the loop body will execute at least once, even if the condition is initially false.

```Java

int i = 0;

do {

System.out.println("Value of i: " + i);

i++;

} while (i < 2);

```

**Jump Statements:** These statements allow you to alter the normal flow of execution within loops or switch statements.

**break statement:**

* + When used inside a loop (for, while, do-while), it immediately terminates the loop and transfers control to the statement immediately following the loop.
  + When used inside a switch statement, it terminates the switch block.

```Java

for (int i = 0; i < 10; i++) {

if (i == 5) {

break; // Exit the loop when i is 5

}

System.out.println("Looping: " + i);

}

System.out.println("Loop finished.");

```

**continue statement:** When used inside a loop, it skips the rest of the current iteration and proceeds to the next iteration of the loop.

```Java

for (int i = 0; i < 5; i++) {

if (i == 2) {

continue; // Skip the rest of this iteration when i is 2

}

System.out.println("Processing: " + i);

}

```

Understanding and using these control flow statements effectively is essential for creating programs that can make decisions and perform repetitive tasks.

**Arrays**

An **array** in Java is a fixed-size, ordered collection of elements of the same data type. Arrays provide a way to store and access multiple values using a single variable name and an index (position).

**Key Characteristics of Arrays:**

* **Fixed Size:** Once an array is created, its size cannot be changed.
* **Homogeneous:** All elements in an array must be of the same data type (primitive or reference).
* **Ordered:** Elements in an array are stored in a specific order, and each element has an index starting from 0.
* **Contiguous Memory Allocation:** Typically, array elements are stored in contiguous memory locations, which allows for efficient access to elements.

**Declaration and Initialization of Arrays:**

There are several ways to declare and initialize arrays in Java:

**1. Declaration followed by Initialization:**

```Java

// Declare an array of integers named 'numbers' that can hold 5 elements

int[] numbers;

// Allocate memory for 5 integers

numbers = new int[5];

// Initialize the elements of the array using their index (0-based)

numbers[0] = 10;

numbers[1] = 20;

numbers[2] = 30;

numbers[3] = 40;

numbers[4] = 50;

```

You can also combine the declaration and allocation in one step:

Java

int[] scores = new int[10]; // Declare and allocate memory for 10 integers

**2. Declaration and Initialization with Initial Values:**

You can directly initialize an array with values during declaration using curly braces {}:

```Java

int[] ages = {25, 30, 22, 28, 27}; // An array of 5 integers

String[] names = {"Alice", "Bob", "Charlie"}; // An array of 3 Strings

```

In this case, the size of the array is automatically determined by the number of elements provided within the curly braces.

**Accessing Array Elements:**

You can access individual elements of an array using their index, which starts from 0 for the first element and goes up to array.length - 1 for the last element.

```Java

int[] values = {100, 200, 300};

System.out.println("First element: " + values[0]); // Output: First element: 100

System.out.println("Second element: " + values[1]); // Output: Second element: 200

System.out.println("Third element: " + values[2]); // Output: Third element: 300

```

**Array Length:**

You can get the number of elements in an array using the length property (note that it's a property, not a method, so you don't use parentheses).

```Java

int[] data = {1, 2, 3, 4, 5, 6};

System.out.println("Length of the array: " + data.length); // Output: Length of the array: 6

```

**Iterating Through Arrays:**

You can use loops to process all the elements of an array. The for loop and the enhanced for loop are commonly used for this purpose.

**Using a traditional for loop:**

```Java

int[] numbers = {10, 20, 30, 40, 50};

for (int i = 0; i < numbers.length; i++) {

System.out.println("Element at index " + i + ": " + numbers[i]);

}

```

**Using an enhanced for loop (for-each loop):**

This provides a more concise way to iterate through the elements of an array without explicitly managing indices.

```Java

int[] scores = {85, 92, 78, 95, 88};

for (int score : scores) {

System.out.println("Score: " + score);

}

```

**Multi-dimensional Arrays:**

Java also supports multi-dimensional arrays, which are arrays of arrays. The most common type is a two-dimensional array, which can be thought of as a table with rows and columns.

**Declaration and Initialization of a 2D Array:**

```Java

// Declare a 2D array with 3 rows and 4 columns

int[][] matrix = new int[3][4];

// Initialize elements

matrix[0][0] = 1;

matrix[0][1] = 2;

matrix[1][2] = 7;

// ... and so on

// Declare and initialize with values

int[][] grid = {

{1, 2, 3}, // Row 0

{4, 5, 6}, // Row 1

{7, 8, 9} // Row 2

};

```

**Accessing Elements in a 2D Array:**

You need two indices to access an element in a 2D array: the row index and the column index.

```Java

System.out.println("Element at row 1, column 2: " + grid[1][2]); // Output: 6

```

**Iterating Through a 2D Array:**

You typically use nested loops to iterate through the rows and columns of a 2D array.

```Java

int[][] table = {

{10, 20},

{30, 40},

{50, 60}

};

for (int i = 0; i < table.length; i++) { // Iterate through rows

for (int j = 0; j < table[i].length; j++) { // Iterate through columns in the current row

System.out.print(table[i][j] + " ");

}

System.out.println(); // Move to the next line after each row

}

```

Java also allows for arrays with more than two dimensions, but they are less commonly used in basic programming.

Arrays are a fundamental data structure in Java and are used extensively for storing and manipulating collections of data. However, it's important to remember their fixed size limitation. For more dynamic collections, Java provides the **Collections Framework**, which we will explore later.

**Object Oriented Programing Concepts.**

Object-Oriented Programming (OOP) Principles in Java. OOP is a programming paradigm centered around the concept of "objects," which are instances of "classes." Java is fundamentally an object-oriented language, and understanding these principles is essential for writing well-structured, maintainable, and scalable code.

Every object has property and behaviour.

To create an object we need a class, the blue print.

JVM creates object in java.

Class code get converted to byte code and it goes to JVM to create an object.

```Java

class Calculator

{

// variables inside a class at the are called Instance variable

int ref = 4;

// Methods

public void inAdd()

{

System.out.println(“Inside add”);

}

public int add(int n1, int n2)

{

int result = n1 + n2;

Return result;

}

}

public class Demo

{

public static void main(String[] args)

{

// Stage 1

int num1 = 3;

int num2 = 4;

int result = num1 + num2; // action or behaviour

System.out.println(result);

// Stage 2

// To use the above created class we need to create an object in our

current class.

Calculator calc = new Calculator();

// Calculator is the class name, calc is the reference variable.

// Created an object of type Calculator using Calculator class

(Blueprint).

// now calling the method inside Calculator class using newly created object.

calc.inAdd();

calc add( num1, num2 );

}

}

```