1. Into the world of Java!

Introduction to Java

Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It's a general-purpose programming language intended to let application developers "write once, run anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. It's renowned for its robustness, security, and platform independence.

Why you must learn Java

Java is one of the most popular programming languages in the world for several reasons:

- **Versatility:** It's used in a wide range of applications, from mobile apps (Android) and enterprise software to web applications, big data, and scientific applications.
- **Large Community & Resources:** There's a vast community, extensive documentation, and numerous tutorials available, making it easier to learn and get support.
- **Career Opportunities:** Many job opportunities exist for Java developers in various industries.
- **Scalability:** Java applications are known for their scalability and performance, especially in large-scale systems.
- **Object-Oriented:** Its object-oriented nature promotes modular, reusable, and maintainable code.

What is a Programming Language

A programming language is a formal language that comprises a set of instructions used to produce various kinds of output. In essence, it's a way for humans to communicate with computers. Just like humans use languages like English or Spanish, computers understand instructions given in programming languages. These languages have their own grammar (syntax) and vocabulary (keywords).

- **Example:** When you tell a computer to `print("Hello, World!")` in Python, you're using the Python programming language to give it an instruction.

What is an Algorithm

An algorithm is a step-by-step procedure or a set of rules for solving a specific problem or accomplishing a task. Think of it as a recipe – a sequence of clear instructions that, when followed, will always lead to the desired outcome.

- **Example:** An algorithm to make a cup of tea:
    1. Boil water.
    2. Place a tea bag in a cup.
    3. Pour hot water into the cup.
    4. Add sugar/milk (optional).
    5. Stir.

What is Syntax

Syntax refers to the set of rules that define the combinations of symbols that are considered to be correctly structured statements or expressions in a particular programming language. It's like the grammar of a human language. If you don't follow the syntax rules, the computer won't understand your instructions, leading to "syntax errors."

- **Example (Java):** Every statement in Java typically ends with a semicolon (`;`). If you write `System.out.println("Hello World")` without the semicolon, it's a syntax error.

History of Java

Java was originally developed by James Gosling at Sun Microsystems (which was later acquired by Oracle) and released in 1995. Initially named "Oak," it was designed for interactive television. When that project failed, it was re-purposed for the burgeoning World Wide Web. Its core principle was "Write Once, Run Anywhere," making it incredibly appealing for developing web applets that could run on any browser on any operating system.

Magic of Byte Code

This is where Java's "write once, run anywhere" philosophy comes to life. When you compile Java source code (`.java` files), it doesn't directly turn into machine code for a specific operating system. Instead, it gets compiled into an intermediate format called **bytecode** (`.class` files).

- **How it works:** This bytecode is then executed by the Java Virtual Machine (JVM), which acts as an interpreter for the bytecode. Since JVMs are available for various operating systems (Windows, macOS, Linux, etc.), the same bytecode can run on any system that has a compatible JVM installed. This layer of abstraction provides platform independence. Here's a visual representation of this process:

How Java Changed the Internet

In its early days, Java played a crucial role in the development of dynamic and interactive web pages through **Java Applets**. Applets were small Java programs that could be embedded in web pages and run directly within a web browser. Before Java, web pages were largely static. Applets allowed for animations, games, and rich user interfaces directly in the browser, without requiring specific plugins or browser-dependent code. While applets have largely been deprecated due to security concerns and the rise of other web technologies (like JavaScript, HTML5, CSS3), Java's impact on making the internet more dynamic was profound and paved the way for modern web applications.

Java Buzzwords

When Java was first introduced, several "buzzwords" highlighted its key features and advantages:

- **Simple:** Designed to be easy to learn for programmers familiar with C/C++.
- **Object-Oriented:** Based on the object-oriented paradigm.
- **Distributed:** Designed for networked environments.
- **Robust:** Strong memory management and error handling features.
- **Secure:** Built-in security features, especially for applets.
- **Architecture-Neutral:** Platform independent due to bytecode.
- **Portable:** "Write Once, Run Anywhere."

- **High Performance:** Just-In-Time (JIT) compilers improve execution speed.
- **Multithreaded:** Ability to perform multiple tasks concurrently.
- **Dynamic:** Ability to adapt to evolving environments.

What is Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data (attributes or properties) and code (methods or behaviors). The core idea is to model real-world entities as software objects, making programs more modular, reusable, and easier to manage.

- **Key Principles of OOP:**
  - **Encapsulation:** Bundling data and the methods that operate on that data within a single unit (an object). It hides the internal details from the outside world.
  - **Inheritance:** A mechanism where one class (subclass/child) can inherit properties and behaviors from another class (superclass/parent), promoting code reuse.
  - **Polymorphism:** The ability of an object to take on many forms. It allows methods to do different things depending on the object it is acting upon.
  - **Abstraction:** Hiding complex implementation details and showing only the essential features of an object.
- **Example:** Consider a `Car` object.
  - **Attributes:** `color`, `make`, `model`, `speed`.
  - **Methods:** `start()`, `accelerate()`, `brake()`. The `Car` object encapsulates its data (color, speed) and its behavior (start, accelerate).

2. Java Basics

Compiling and Running

The process of getting your Java code to execute involves two main steps:

1. **Compiling:** You use the Java compiler (`javac`) to translate your human-readable Java source code (`.java` file) into platform-independent bytecode (`.class` file).
   - **Command:** `javac MyProgram.java`
   - **Output:** Creates `MyProgram.class`
2. **Running:** You then use the Java Virtual Machine (`java`) to execute the bytecode. The JVM interprets the bytecode and translates it into machine-specific instructions on the fly.
   - **Command:** `java MyProgram`
   - **Output:** The program executes.

Anatomy of a Class

In Java, almost everything happens inside a class. A class is a blueprint or a template for creating objects. It defines the structure and behavior that all objects of that class will have. Here's a basic anatomy:

```
// This is a single-line comment
/*
 * This is a multi-line comment.
 * It provides more detailed explanations.
```

```java
 */

package com.example.myprogram; // Optional: Defines the package where the class belongs

public class MyFirstClass { // Class Declaration: 'public' is an access modifier, 'class' keyword, 'MyFirstClass' is the class name

    // Instance variables (fields/attributes) - represent the state of an object
    String message = "Hello from Java!";
    int number = 10;

    // Constructor - a special method used to initialize objects of the class
    public MyFirstClass() {
        // Constructor logic
    }

    // Methods - represent the behavior of an object
    public void displayMessage() { // 'public' access modifier, 'void' return type (returns nothing), 'displayMessage' is the method name
        System.out.println(message); // Prints the value of the 'message' variable
    }

    // The main method - the entry point of a Java application
    public static void main(String[] args) {
        // Code inside main method is executed when the program runs
        MyFirstClass myObject = new MyFirstClass(); // Creating an object (instance) of MyFirstClass
        myObject.displayMessage(); // Calling a method on the object

        // Another example directly in main
        System.out.println("This is also inside the main method.");
    }
}
```

Let's see this in action:

File Extensions

In Java, you'll encounter a few common file extensions:

- `.java`: This is the **source code** file. It contains the human-readable Java code written by the programmer.
- `.class`: This is the **bytecode** file. It's generated by the Java compiler (`javac`) from a `.java` file and contains platform-independent instructions for the JVM.
- `.jar`: This stands for **Java Archive**. It's a package file format used to aggregate many Java class files, associated metadata, and resources (text, images, etc.) into a single file for distribution. It's essentially a compressed file format.

JDK vs JVM vs JRE

These three terms are crucial to understanding the Java ecosystem:

- **JVM (Java Virtual Machine):**
  - **What it is:** The heart of Java's "write once, run anywhere" capability. It's an abstract machine that provides a runtime environment in which Java bytecode can be executed.
  - **Purpose:** Interprets and executes `.class` files (bytecode). It's responsible for tasks like memory management (garbage collection) and security.
  - **Role:** It only *runs* Java applications.
- **JRE (Java Runtime Environment):**
  - **What it is:** A bundle that contains the JVM, Java core classes (libraries), and supporting files.
  - **Purpose:** It's what you need to *run* a Java application. If you only want to execute Java programs, but not develop them, you just need the JRE.
  - **Role:** Contains everything needed to run a compiled Java program.
- **JDK (Java Development Kit):**
  - **What it is:** The superset of JRE. It contains the JRE, plus development tools like the Java compiler (`javac`), debugger (`jdb`), and other utilities.
  - **Purpose:** It's what you need to *develop, compile, and run* Java applications. If you're a Java developer, you need the JDK.
  - **Role:** Provides the entire environment for Java development and execution.

In simple terms:

- **JVM:** Executes bytecode.
- **JRE:** = JVM + Libraries (to run applications).
- **JDK:** = JRE + Development Tools (to develop and run applications).

Showing Output

In Java, the primary way to display output to the console (the screen) is by using the `System.out.println()` method.

- `System` : A final class from the `java.lang` package (automatically imported).
- `out` : A static member of the `System` class, which is an instance of `PrintStream` .
- `println()` : A method of the `PrintStream` class that prints the argument passed to it and then moves the cursor to the next line.
- `print()` : Similar to `println()` , but it does *not* move the cursor to the next line after printing.

```
public class OutputExample {
    public static void main(String[] args) {
        // Using println() - prints and then moves to the next line
        System.out.println("Hello, Java!");
        System.out.println("This is a new line.");

        // Using print() - prints and stays on the same line
        System.out.print("This is on the same line. ");
        System.out.print("Still on the same line.");
        System.out.println("\nNow we're on a new line after print.");
```

```
        // Printing variables
        String name = "Alice";
        int age = 30;
        System.out.println("My name is " + name + " and I am " + age + " years old.");
    }
}
```

Output:

```
Hello, Java!
This is a new line.
This is on the same line. Still on the same line.
Now we're on a new line after print.
My name is Alice and I am 30 years old.
```

Importance of the `main` method

The `public static void main(String[] args)` method is the **entry point** for any standalone Java application. When you execute a Java program, the JVM specifically looks for this `main` method to start the execution. Let's break down its components:

- `public`: This is an access modifier. It means the `main` method can be accessed from anywhere, which is necessary for the JVM to start execution.
- `static`: This keyword means that the `main` method belongs to the class itself, not to any specific object of the class. This allows the JVM to call `main()` without creating an object of the class first.
- `void`: This is the return type. `void` means the method does not return any value.
- `main`: This is the name of the method. It's a special, reserved name that the JVM recognizes as the starting point.
- `(String[] args)`: This is the parameter list. `args` is an array of `String` objects, which allows you to pass command-line arguments to your Java program when you run it. These arguments are optional.

Why is it important? Without a `main` method (or if it's incorrectly defined), the JVM wouldn't know where to begin executing your program, and your application simply wouldn't run. It's the central hub from which all other parts of your program are typically invoked.

---

3. Data Types, Variables & Input

At the heart of any programming language is the ability to store and manipulate data. In Java, this is primarily done using **variables** and **data types**.

Variables

A **variable** is a named storage location that holds a value. Think of it as a container with a label. The value stored in a variable can change during the execution of a program.

Declaration and Initialization: Before you can use a variable, you must declare it, which means giving it a name and specifying its data type. You can also initialize it (assign an initial value) at the same time.

Syntax: `dataType variableName = value;`

Example:

```java
public class VariableExample {
    public static void main(String[] args) {
        // Declaration of an integer variable
        int age;

        // Initialization of the age variable
        age = 30;

        // Declaration and initialization of a String variable
        String name = "Alice";

        // Changing the value of a variable
        age = 31;

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

Data Types

Data types classify the kind of values a variable can hold. Java is a strongly-typed language, meaning you must declare the type of a variable before using it. Java has two main categories of data types:

1. **Primitive Data Types:** These are fundamental, built-in data types that hold simple values. They are not objects.
    - **Integral Types (for whole numbers):**
        - `byte` : 8-bit signed integer (range -128 to 127)
        - `short` : 16-bit signed integer (range -32,768 to 32,767)
        - `int` : 32-bit signed integer (most commonly used, range approx. -2 billion to 2 billion)
        - `long` : 64-bit signed integer (for very large numbers)
    - **Floating-Point Types (for numbers with decimal points):**
        - `float` : 32-bit single-precision floating-point
        - `double` : 64-bit double-precision floating-point (most commonly used for decimals)
    - **Character Type:**
        - `char` : 16-bit Unicode character (can hold a single letter, digit, or symbol)
    - **Boolean Type:**
        - `boolean` : Represents `true` or `false`
2. **Non-Primitive (Reference) Data Types:** These are more complex data types that refer to objects. They don't store the actual values directly but rather references (memory addresses) to where the objects are stored.

- `String`: Used to store sequences of characters (text). It's an object, not a primitive.
- Arrays: Used to store collections of elements of the same type.
- Classes and Interfaces: User-defined types.

Example of Primitive Data Types:

```
public class PrimitiveDataTypes {
    public static void main(String[] args) {
        byte myByte = 100;
        short myShort = 5000;
        int myInt = 100000;
        long myLong = 150000000000L; // 'L' suffix for long literal

        float myFloat = 3.14f; // 'f' suffix for float literal
        double myDouble = 3.1415926535;

        char myChar = 'A';
        boolean isJavaFun = true;

        System.out.println("byte: " + myByte);
        System.out.println("short: " + myShort);
        System.out.println("int: " + myInt);
        System.out.println("long: " + myLong);
        System.out.println("float: " + myFloat);
        System.out.println("double: " + myDouble);
        System.out.println("char: " + myChar);
        System.out.println("boolean: " + isJavaFun);
    }
}
```

Naming Conventions

Consistent naming conventions make code readable and maintainable. Java has well-established conventions:

- **Variables:** `camelCase` (starts with lowercase, subsequent words start with uppercase).
  - Example: `firstName`, `totalAmount`, `isEnabled`
- **Methods:** `camelCase` (same as variables).
  - Example: `calculateSum()`, `printMessage()`, `getUserInput()`
- **Classes/Interfaces:** `PascalCase` (starts with uppercase, subsequent words start with uppercase).
  - Example: `MyClass`, `BankAccount`, `Shape`
- **Constants:** `UPPER_SNAKE_CASE` (all uppercase, words separated by underscores).
  - Example: `MAX_VALUE`, `PI`, `DEFAULT_TIMEOUT`
- **Packages:** `lowercase.dot.separated` (all lowercase, words separated by dots).
  - Example: `com.example.myapp`, `org.utilities`

Literals

A **literal** is a fixed value that appears directly in your code. It's how you represent specific values of various data types.

- **Integer Literals:**
  - `10`, `0`, `-500` (default `int` type)
  - `100L` (a `long` literal)
- **Floating-Point Literals:**
  - `3.14`, `0.0`, `-1.5` (default `double` type)
  - `2.718f` (a `float` literal)
- **Character Literals:**
  - `'A'`, `'b'`, `'5'`, `'$'` (single characters enclosed in single quotes)
- **String Literals:**
  - `"Hello, Java"`, `"Programming is fun"`, `""` (sequences of characters enclosed in double quotes)
- **Boolean Literals:**
  - `true`, `false`

Example:

```
public class LiteralsExample {
    public static void main(String[] args) {
        int count = 100;           // 100 is an int literal
        double price = 19.99;      // 19.99 is a double literal
        char grade = 'A';          // 'A' is a char literal
        String message = "Welcome!"; // "Welcome!" is a String literal
        boolean isValid = true;    // true is a boolean literal
        long bigNumber = 9876543210L; // 9876543210L is a long literal
        float smallDecimal = 0.5f;   // 0.5f is a float literal

        System.out.println(count);
        System.out.println(price);
        System.out.println(grade);
        System.out.println(message);
        System.out.println(isValid);
        System.out.println(bigNumber);
        System.out.println(smallDecimal);
    }
}
```

Keywords

Keywords are reserved words in Java that have special meaning to the compiler. You cannot use them as identifiers (names for variables, classes, methods, etc.).

Some common Java keywords: `public`, `private`, `protected`, `static`, `void`, `class`, `interface`, `enum`, `if`, `else`, `for`, `while`, `do`, `switch`, `case`, `default`, `break`, `continue`, `return`, `new`, `this`, `super`, `try`, `catch`, `finally`, `throw`, `throws`, `import`, `package`, `abstract`, `final`, `native`, `strictfp`, `synchronized`, `transient`, `volatile`, `assert`, `const` (reserved but unused), `goto` (reserved but unused).

Example (demonstrating proper use, not misusing them):

```
public class KeywordExample { // 'public', 'class' are keywords
    public static void main(String[] args) { // 'public', 'static', 'void' are
keywords
```

```
        int count = 10; // 'int' is a keyword
        if (count > 5) { // 'if' is a keyword
            System.out.println("Count is greater than 5.");
        }
    }
}
```

Escape Sequences

Escape sequences are special combinations of a backslash (`\`) followed by a character that represent characters that are difficult or impossible to type directly, or have a special meaning in string or character literals.

Common Escape Sequences:

- `\n` : Newline (moves cursor to the next line)
- `\t` : Tab (inserts a horizontal tab)
- `\"` : Double quote (allows you to include a double quote within a double-quoted string)
- `\'` : Single quote (allows you to include a single quote within a single-quoted char or double-quoted string)
- `\\` : Backslash (allows you to include a backslash itself)
- `\r` : Carriage return
- `\b` : Backspace

Example:

```
public class EscapeSequenceExample {
    public static void main(String[] args) {
        System.out.println("Hello\nWorld!");        // Newline
        System.out.println("Name:\tAlice");        // Tab
        System.out.println("She said, \"Hello!\""); // Double quote
        System.out.println("It's a beautiful day."); // Single quote (no escape needed in String, but 'It\'s' would also work)
        System.out.println("Path: C:\\Program Files\\Java"); // Backslash
        char singleQuote = '\'';                   // Single quote in char literal
        System.out.println("Single quote char: " + singleQuote);
    }
}
```

User Input

Getting input from the user is a common requirement. In Java, the `Scanner` class (from the `java.util` package) is widely used for this purpose.

Steps:

1. **Import `Scanner`:** `import java.util.Scanner;`
2. **Create a `Scanner` object:** `Scanner scanner = new Scanner(System.in);` (`System.in` represents the standard input stream, usually the keyboard).
3. **Read input:** Use methods like `next()`, `nextLine()`, `nextInt()`, `nextDouble()`, `nextBoolean()` to read different data types.
4. **Close the `Scanner`:** `scanner.close();` (It's good practice to close resources when you're done with them).

Example:

```java
import java.util.Scanner; // Import the Scanner class

public class UserInputExample {
    public static void main(String[] args) {
        // Create a Scanner object to read input from the console
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter your name: ");
        String name = scanner.nextLine(); // Reads a whole line of text

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

        System.out.print("Enter your height in meters (e.g., 1.75): ");
        double height = scanner.nextDouble(); // Reads a double

        System.out.print("Are you a student? (true/false): ");
        boolean isStudent = scanner.nextBoolean(); // Reads a boolean

        // Consume the leftover newline character after nextInt(), nextDouble(), etc.
        // This is important if nextLine() is called after other nextXXX() methods.
        scanner.nextLine();

        System.out.println("\n--- Your Details ---");
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Height: " + height + " meters");
        System.out.println("Student: " + isStudent);

        // Close the scanner to release system resources
        scanner.close();
    }
}
```

Type Conversion and Casting

Sometimes you need to convert a value from one data type to another.

1. **Type Conversion (Widening/Implicit):** This happens automatically when you assign a value of a smaller data type to a larger data type. It's safe because no data is lost.
    - `byte -> short -> int -> long -> float -> double`

    Example:

    ```java
    public class TypeConversionExample {
        public static void main(String[] args) {
            int myInt = 100;
            double myDouble = myInt; // int is automatically converted to double (100.0)

            System.out.println("int: " + myInt);
            System.out.println("double (from int): " + myDouble);
    ```

```
        long myLong = 5000L;
        float myFloat = myLong; // long is automatically converted to float
(5000.0f)
        System.out.println("float (from long): " + myFloat);
    }
}
```

2. **Type Casting (Narrowing/Explicit):** This requires you to explicitly tell the compiler to convert a value from a larger data type to a smaller one. This can lead to **loss of data** or precision if the value is too large for the target type.

   Syntax: `(targetDataType) value`

   Example:

```
public class TypeCastingExample {
    public static void main(String[] args) {
        double myDouble = 9.78;
        int myInt = (int) myDouble; // Explicitly cast double to int. Decimal
part is truncated.

        System.out.println("double: " + myDouble);
        System.out.println("int (from double): " + myInt); // Output: 9

        int largeInt = 130;
        byte myByte = (byte) largeInt; // Explicitly cast int to byte. Data
loss occurs here.
                                       // 130 is outside byte's range (-128 to
127).
                                       // Result will be -126 due to overflow.

        System.out.println("int: " + largeInt);
        System.out.println("byte (from int): " + myByte); // Output: -126

        char myChar = 'A';
        int charToInt = (int) myChar; // Cast char to its ASCII/Unicode integer
value
        System.out.println("char: " + myChar + " -> int: " + charToInt); //
Output: 65

        int intToChar = 66;
        char backToChar = (char) intToChar; // Cast int to char
        System.out.println("int: " + intToChar + " -> char: " + backToChar); //
Output: B
    }
}
```

---

4. Operators, If-else, Number System

Assignment Operator

The **assignment operator** (`=`) is used to assign a value to a variable.

Syntax: `variable = value;`

Example:

```
public class AssignmentOperatorExample {
    public static void main(String[] args) {
        int x = 10; // Assigns the value 10 to variable x
        String message = "Hello"; // Assigns the string "Hello" to message

        int y;
        y = x; // Assigns the current value of x (which is 10) to y

        System.out.println("x: " + x);
        System.out.println("y: " + y);
        System.out.println("message: " + message);
    }
}
```

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

- `+`: Addition
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division
- `%`: Modulo (remainder of a division)

Example:

```
public class ArithmeticOperatorsExample {
    public static void main(String[] args) {
        int a = 20;
        int b = 5;

        System.out.println("a + b = " + (a + b)); // Addition: 25
        System.out.println("a - b = " + (a - b)); // Subtraction: 15
        System.out.println("a * b = " + (a * b)); // Multiplication: 100
        System.out.println("a / b = " + (a / b)); // Division: 4 (integer division)
        System.out.println("a % b = " + (a % b)); // Modulo: 0

        int c = 21;
        int d = 5;
        System.out.println("c / d = " + (c / d)); // Integer division: 4 (truncates
decimal)
        System.out.println("c % d = " + (c % d)); // Modulo: 1 (21 divided by 5 is 4
with remainder 1)

        double x = 10.0;
        double y = 3.0;
        System.out.println("x / y = " + (x / y)); // Floating-point division: 3.333...
```

```
    }
}
```

Order of Operation

Java follows the standard mathematical order of operations (PEMDAS/BODMAS):

1. **P**arentheses `()`
2. **E**xponents (not a direct operator in Java, usually via `Math.pow()`)
3. **M**ultiplication `*`, **D**ivision `/`, **M**odulo `%` (from left to right)
4. **A**ddition `+`, **S**ubtraction `-` (from left to right)

Example:

```
public class OrderOfOperationExample {
    public static void main(String[] args) {
        int result1 = 10 + 5 * 2;
        // Expected: 10 + (5 * 2) = 10 + 10 = 20
        System.out.println("Result 1: " + result1);

        int result2 = (10 + 5) * 2;
        // Expected: (15) * 2 = 30
        System.out.println("Result 2: " + result2);

        int result3 = 20 / 4 + 2 % 3;
        // Expected: (20 / 4) + (2 % 3) = 5 + 2 = 7
        System.out.println("Result 3: " + result3);

        int result4 = 10 - 2 * 3 + 1;
        // Expected: 10 - (2 * 3) + 1 = 10 - 6 + 1 = 4 + 1 = 5
        System.out.println("Result 4: " + result4);
    }
}
```

Shorthand Operators

Shorthand (or compound) assignment operators combine an arithmetic operation with an assignment. They are a concise way to modify a variable's value.

- `+=`: Add and assign (`a += b` is equivalent to `a = a + b`)
- `-=`: Subtract and assign (`a -= b` is equivalent to `a = a - b`)
- `*=`: Multiply and assign (`a *= b` is equivalent to `a = a * b`)
- `/=`: Divide and assign (`a /= b` is equivalent to `a = a / b`)
- `%=`: Modulo and assign (`a %= b` is equivalent to `a = a % b`)

Example:

```
public class ShorthandOperatorsExample {
    public static void main(String[] args) {
        int x = 10;
        x += 5; // x = x + 5; // x is now 15
        System.out.println("x after += 5: " + x);

        int y = 20;
        y -= 7; // y = y - 7; // y is now 13
```

```
        System.out.println("y after -= 7: " + y);

        int z = 3;
        z *= 4; // z = z * 4; // z is now 12
        System.out.println("z after *= 4: " + z);

        int w = 25;
        w /= 5; // w = w / 5; // w is now 5
        System.out.println("w after /= 5: " + w);

        int q = 17;
        q %= 3; // q = q % 3; // q is now 2 (remainder of 17 / 3)
        System.out.println("q after %= 3: " + q);
    }
}
```

Unary Operators

Unary operators work on a single operand.

- `+`: Unary plus (indicates positive value, rarely used as numbers are positive by default)
- `-`: Unary minus (negates a value)
- `++`: Increment (increases value by 1)
  - **Prefix:** `++x` (increments then uses value)
  - **Postfix:** `x++` (uses value then increments)
- `--`: Decrement (decreases value by 1)
  - **Prefix:** `--x` (decrements then uses value)
  - **Postfix:** `x--` (uses value then decrements)
- `!`: Logical NOT (inverts a boolean value)

Example:

```
public class UnaryOperatorsExample {
    public static void main(String[] args) {
        int a = 5;
        System.out.println("+a: " + (+a)); // Unary plus (redundant but valid)
        System.out.println("-a: " + (-a)); // Unary minus: -5

        int b = 10;
        System.out.println("b: " + b); // 10
        System.out.println("++b (prefix): " + (++b)); // b becomes 11, then prints 11
        System.out.println("b after prefix: " + b); // 11

        int c = 10;
        System.out.println("c: " + c); // 10
        System.out.println("c++ (postfix): " + (c++)); // prints 10, then c becomes 11
        System.out.println("c after postfix: " + c); // 11

        int d = 15;
        System.out.println("--d (prefix): " + (--d)); // d becomes 14, then prints 14
        System.out.println("d after prefix: " + d); // 14
```

```
        int e = 15;
        System.out.println("e-- (postfix): " + (e--)); // prints 15, then e becomes 14
        System.out.println("e after postfix: " + e); // 14

        boolean isJavaFun = true;
        System.out.println("isJavaFun: " + isJavaFun);
        System.out.println("!isJavaFun: " + (!isJavaFun)); // Logical NOT: false
    }
}
```

If-else

The `if-else` statement is used for conditional execution. It allows your program to make decisions based on whether a condition is `true` or `false`.

Syntax:

```
if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}
```

You can also have `else if` for multiple conditions.

Example:

```
public class IfElseExample {
    public static void main(String[] args) {
        int temperature = 25;

        if (temperature > 30) {
            System.out.println("It's a hot day!");
        } else if (temperature > 20) { // This block runs if temperature is NOT > 30,
but IS > 20
            System.out.println("It's a pleasant day.");
        } else if (temperature > 10) { // This block runs if temperature is NOT > 20,
but IS > 10
            System.out.println("It's a cool day.");
        }
        else { // This block runs if none of the above conditions are true
            System.out.println("It's cold!");
        }

        System.out.println("\nAnother example:");
        int score = 75;
        String grade;

        if (score >= 90) {
            grade = "A";
        } else if (score >= 80) {
            grade = "B";
        } else if (score >= 70) {
            grade = "C";
```

```
        } else if (score >= 60) {
            grade = "D";
        } else {
            grade = "F";
        }
        System.out.println("Score: " + score + ", Grade: " + grade);
    }
}
```

Relational Operators

Relational (or comparison) operators are used to compare two values. They always return a `boolean` result (`true` or `false`).

- `==`: Equal to
- `!=`: Not equal to
- `>`: Greater than
- `<`: Less than
- `>=`: Greater than or equal to
- `<=`: Less than or equal to

Example:

```
public class RelationalOperatorsExample {
    public static void main(String[] args) {
        int x = 10;
        int y = 5;
        int z = 10;

        System.out.println("x == y: " + (x == y));   // false
        System.out.println("x != y: " + (x != y));   // true
        System.out.println("x == z: " + (x == z));   // true
        System.out.println("x > y: " + (x > y));     // true
        System.out.println("x < y: " + (x < y));     // false
        System.out.println("x >= z: " + (x >= z));   // true
        System.out.println("y <= x: " + (y <= x));   // true
    }
}
```

Logical Operators

Logical operators combine boolean expressions to form more complex conditions.

- `&&`: Logical AND (returns `true` if *both* operands are `true`)
- `||`: Logical OR (returns `true` if *at least one* operand is `true`)
- `!`: Logical NOT (unary, inverts the boolean value)

Example:

```
public class LogicalOperatorsExample {
    public static void main(String[] args) {
        boolean isSunny = true;
        boolean isWarm = false;
        int age = 25;
        int minAge = 18;
```

```
        int maxAge = 30;

        // Logical AND (&&)
        // Both conditions must be true
        if (isSunny && isWarm) {
            System.out.println("It's a perfect beach day!");
        } else {
            System.out.println("Maybe not ideal for the beach."); // This will print
        }

        // Check if age is between minAge and maxAge (inclusive)
        if (age >= minAge && age <= maxAge) {
            System.out.println("Age is within the allowed range."); // This will print
        }

        // Logical OR (||)
        // At least one condition must be true
        if (isSunny || isWarm) {
            System.out.println("It's either sunny or warm (or both)."); // This will
print
        }

        boolean hasPermission = false;
        boolean isAdmin = true;

        if (hasPermission || isAdmin) {
            System.out.println("Access granted!"); // This will print
        }

        // Logical NOT (!)
        boolean hasLicense = false;
        if (!hasLicense) { // if NOT hasLicense (i.e., if hasLicense is false)
            System.out.println("Please get your license."); // This will print
        }
    }
}
```

Operator Precedence

Operator precedence determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before operators with lower precedence. When operators have the same precedence, associativity (usually left-to-right) determines the order.

Simplified Precedence (highest to lowest, common operators):

1. Parentheses `()`
2. Postfix `++`, `--`
3. Prefix `++`, `--`, Unary `+`, `-`, Logical NOT `!`
4. Multiplicative `*`, `/`, `%`
5. Additive `+`, `-`
6. Relational `<`, `>`, `<=`, `>=`
7. Equality `==`, `!=`

8. Logical AND `&&`
9. Logical OR `||`
10. Assignment `=`, `+=`, `-=`, etc.

It's often best practice to use parentheses `()` to explicitly define the order of operations, even if default precedence would produce the same result, as it improves readability.

Example:

```
public class OperatorPrecedenceExample {
    public static void main(String[] args) {
        int a = 5, b = 2, c = 3;

        // Example 1: Arithmetic and Assignment
        int result1 = a + b * c; // Equivalent to a + (b * c)
        System.out.println("Result 1 (a + b * c): " + result1); // 5 + 2 * 3 = 5 + 6 = 11

        // Example 2: With parentheses
        int result2 = (a + b) * c; // Forces addition first
        System.out.println("Result 2 ((a + b) * c): " + result2); // (5 + 2) * 3 = 7 * 3 = 21

        // Example 3: Relational and Logical
        boolean condition1 = a > b && b < c; // (a > b) && (b < c)
                                             // (5 > 2) && (2 < 3)
                                             // true && true = true
        System.out.println("Condition 1 (a > b && b < c): " + condition1);

        boolean condition2 = a == b || b != c; // (a == b) || (b != c)
                                               // (5 == 2) || (2 != 3)
                                               // false || true = true
        System.out.println("Condition 2 (a == b || b != c): " + condition2);

        // Example 4: Mixed operators
        int x = 10;
        boolean complexCondition = x++ > 10 || --x < 10 && x != 10;
        // Step-by-step:
        // x++ > 10  -> 10 > 10 (false). x becomes 11.
        // false || (--x < 10 && x != 10)
        // Since it's an OR, and the left side is false, the right side is evaluated.
        // --x < 10  -> x becomes 10 (from 11). 10 < 10 (false).
        // false && x != 10 -> false && 10 != 10 -> false && false = false
        // false || false = false
        System.out.println("Complex Condition: " + complexCondition + ", x: " + x); // false, x: 10
    }
}
```

Intro to Number System

Computers fundamentally operate using binary digits (bits), which are 0s and 1s. While we commonly use the decimal (base-10) system, programmers often encounter other number

systems:

- **Decimal (Base-10):** Uses digits 0-9. Each position represents a power of 10.
  - Example: `123 = 1*10^2 + 2*10^1 + 3*10^0`
- **Binary (Base-2):** Uses digits 0 and 1. Each position represents a power of 2.
  - Example: `1011_2 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 8 + 0 + 2 + 1 = 11_10`
- **Octal (Base-8):** Uses digits 0-7. Each position represents a power of 8.
  - In Java, an octal literal starts with a `0` (e.g., `010` is 8 in decimal).
- **Hexadecimal (Base-16):** Uses digits 0-9 and letters A-F (A=10, B=11, ..., F=15). Each position represents a power of 16.
  - In Java, a hexadecimal literal starts with `0x` or `0X` (e.g., `0xFF` is 255 in decimal).

Java can represent integers in decimal, binary, octal, and hexadecimal formats directly in code:

Example:

```
public class NumberSystemExample {
    public static void main(String[] args) {
        int decimalNum = 255; // Standard decimal literal
        int binaryNum = 0b11111111; // Binary literal (starts with 0b)
        int octalNum = 0377; // Octal literal (starts with 0)
        int hexNum = 0xFF; // Hexadecimal literal (starts with 0x)

        System.out.println("Decimal: " + decimalNum); // 255
        System.out.println("Binary (0b11111111): " + binaryNum); // 255
        System.out.println("Octal (0377): " + octalNum); // 255
        System.out.println("Hexadecimal (0xFF): " + hexNum); // 255

        System.out.println("\nBinary representation of 10: " +
Integer.toBinaryString(10)); // Utility method
        System.out.println("Octal representation of 10: " +
Integer.toOctalString(10));
        System.out.println("Hex representation of 10: " + Integer.toHexString(10));
    }
}
```

Intro to Bitwise Operators

Bitwise operators perform operations on individual bits of integer data types (byte, short, int, long). They are often used in low-level programming, encryption, and optimizing certain algorithms.

- `&` : Bitwise AND
- `|` : Bitwise OR
- `^` : Bitwise XOR (Exclusive OR)
- `~` : Bitwise NOT (Unary Complement)
- `<<` : Left Shift
- `>>` : Right Shift (Signed)
- `>>>` : Right Shift (Unsigned)

How they work (example with `&`): `5 & 3` Binary of 5: `0101` Binary of 3: `0011`

Result: `0001` (which is 1 in decimal) (Only if both bits are 1, the result is 1)

Example:

```
public class BitwiseOperatorsExample {
    public static void main(String[] args) {
        int a = 5;  // Binary: 0101
        int b = 3;  // Binary: 0011

        System.out.println("a & b (AND): " + (a & b)); // 0001 = 1
        System.out.println("a | b (OR): " + (a | b));   // 0111 = 7
        System.out.println("a ^ b (XOR): " + (a ^ b)); // 0110 = 6
        System.out.println("~a (NOT): " + (~a));       // -6 (explaining two's
complement is complex here, but it inverts all bits)

        int num = 8; // Binary: 0000 1000
        System.out.println("num << 1 (Left Shift): " + (num << 1)); // 0001 0000 = 16
(multiplies by 2)
        System.out.println("num >> 1 (Right Shift): " + (num >> 1)); // 0000 0100 = 4
(divides by 2)

        int negativeNum = -8; // Binary (two's complement): 1111 1111 1111 1111 1111
1111 1111 1000
        System.out.println("negativeNum >> 1 (Signed Right Shift): " + (negativeNum >>
1)); // -4 (preserves sign bit)
        System.out.println("negativeNum >>> 1 (Unsigned Right Shift): " + (negativeNum
>>> 1)); // Large positive number (fills with 0s)
    }
}
```

Excellent! Let's delve into loops, methods, and arrays, which are essential for building more complex and organized Java programs.

5. While Loop, Methods & Arrays

Comments

Comments are non-executable lines of text within your code. They are used to explain what the code does, why it does it, or to temporarily disable code. The compiler ignores them.

There are three types of comments in Java:

1. **Single-line comments:** Start with `//` and go to the end of the line.
2. **Multi-line comments (Block comments):** Start with `/*` and end with `*/`. Can span multiple lines.
3. **Documentation comments (Javadoc comments):** Start with `/**` and end with `*/`. Used to generate API documentation.

Example:

```
public class CommentsExample {
    public static void main(String[] args) {
        // This is a single-line comment. It explains the next line.
        int x = 10; // Declaring and initializing a variable
```

```
        /*
         * This is a multi-line comment.
         * It can span across several lines
         * to provide more detailed explanations.
         */
        int y = 20;

        /**
         * This is a Javadoc comment.
         * It's used to document classes, methods, and fields.
         * @param args Command line arguments (not used in this example)
         */
        System.out.println("x + y = " + (x + y));
    }
}
```

While Loop

A **`while` loop** repeatedly executes a block of code as long as a given boolean condition remains `true`. It's an entry-controlled loop, meaning the condition is checked *before* each iteration.

Syntax:

```
while (condition) {
    // Code to be executed repeatedly
    // Make sure something inside the loop changes the condition to eventually become false,
    // otherwise, you'll have an infinite loop!
}
```

Example:

```
public class WhileLoopExample {
    public static void main(String[] args) {
        // Example 1: Counting from 1 to 5
        int count = 1; // Initialization
        System.out.println("Counting up:");
        while (count <= 5) { // Condition
            System.out.println("Count: " + count);
            count++; // Update (important to avoid infinite loop)
        }

        // Example 2: Summing numbers until a limit
        int sum = 0;
        int number = 1;
        System.out.println("\nSumming numbers:");
        while (sum < 10) {
            sum += number;
            System.out.println("Adding " + number + ", current sum: " + sum);
            number++;
        }
```

```
        System.out.println("Final sum: " + sum);
    }
}
```

Methods

A **method** is a block of code that performs a specific task. Methods help organize code, make it reusable, and improve readability.

Syntax:

```
accessModifier static returnType methodName(parameterType parameterName, ...) {
    // Method body
    // Code to perform the task
    return value; // If returnType is not void
}
```

- **`accessModifier`**: (e.g., `public`, `private`) Controls where the method can be accessed.
- **`static`**: (Optional) Means the method belongs to the class itself, not to any specific object of the class.
- **`returnType`**: The data type of the value the method will send back. Use `void` if the method doesn't return anything.
- **`methodName`**: A meaningful name for the method (camelCase convention).
- **`parameters`**: (Optional) A comma-separated list of inputs the method accepts.

Example:

```
public class MethodExample {

    // A method that doesn't return any value (void) and takes no arguments
    public static void sayHello() {
        System.out.println("Hello from a method!");
    }

    // A method that takes two integer arguments and returns their sum
    public static int add(int num1, int num2) {
        int sum = num1 + num2;
        return sum; // Returns an integer value
    }

    // A method that takes a String argument and prints a greeting
    public static void greetUser(String name) {
        System.out.println("Hello, " + name + "!");
    }

    public static void main(String[] args) {
        // Calling the sayHello method
        sayHello();

        // Calling the add method and storing its returned value
        int result = add(5, 3); // 5 and 3 are arguments
        System.out.println("Sum of 5 and 3: " + result);
```

```
        // Calling the add method directly in a print statement
        System.out.println("Sum of 10 and 20: " + add(10, 20));

        // Calling the greetUser method
        greetUser("Alice");
        greetUser("Bob");
    }
}
```

Return statement

The **`return` statement** is used to exit a method and, if the method has a non-`void` return type, to send a value back to the caller.

- In a `void` method, `return;` can be used to exit early.
- In a method with a specific `returnType`, `return value;` must be used, where `value` matches the `returnType`.

Example (as seen in `add` method above, and another here):

```
public class ReturnStatementExample {

    public static int multiply(int x, int y) {
        int product = x * y;
        return product; // Returns the calculated product
    }

    public static boolean isEven(int number) {
        if (number % 2 == 0) {
            return true; // Return true if even
        }
        return false; // Return false if not even (this is reached only if the if condition is false)
    }

    public static void checkEligibility(int age) {
        if (age < 18) {
            System.out.println("You are not eligible yet.");
            return; // Exits the method early for ineligible people
        }
        System.out.println("You are eligible!");
    }

    public static void main(String[] args) {
        int p = multiply(4, 6);
        System.out.println("Product: " + p); // Output: 24

        System.out.println("Is 7 even? " + isEven(7)); // Output: false
        System.out.println("Is 10 even? " + isEven(10)); // Output: true

        checkEligibility(16); // Output: You are not eligible yet.
        checkEligibility(20); // Output: You are eligible!
    }
}
```

Arguments

Arguments are the actual values that are passed into a method when it is called. These values are received by the method's **parameters**. The number, type, and order of arguments must match the parameters defined in the method's signature.

Example (revisiting `MethodExample` to highlight arguments):

```
public class ArgumentsExample {

    // num1 and num2 are parameters
    public static int subtract(int num1, int num2) {
        return num1 - num2;
    }

    // message is a parameter
    public static void displayMessage(String message) {
        System.out.println("The message is: " + message);
    }

    public static void main(String[] args) {
        int value1 = 15;
        int value2 = 7;

        // value1 and value2 are arguments passed to subtract()
        int difference = subtract(value1, value2);
        System.out.println("Difference: " + difference); // Output: 8

        // "Hello world!" is an argument passed to displayMessage()
        displayMessage("Hello world!");

        // 100 is an argument
        displayMessage("Current count: " + 100);
    }
}
```

Arrays

An **array** is an object that stores multiple values of the same data type in a contiguous memory location. It's a fixed-size data structure. Each element in an array is accessed by an `index`, which starts from `0`.

Declaration: `dataType[] arrayName;` or `dataType arrayName[];` (first is preferred)

Initialization (creating the array): `arrayName = new dataType[size];`

Declaration and Initialization: `dataType[] arrayName = new dataType[size];`

Initialization with values: `dataType[] arrayName = {value1, value2, value3, ...};`

Accessing elements: `arrayName[index]`

Properties:

- `arrayName.length` : Returns the number of elements in the array.

Example:

```java
public class ArraysExample {
    public static void main(String[] args) {
        // Declare an array of integers
        int[] numbers;

        // Initialize the array to hold 5 integers
        numbers = new int[5]; // Elements are initialized to default values (0 for
int)

        // Assign values to array elements using their index
        numbers[0] = 10;
        numbers[1] = 20;
        numbers[2] = 30;
        numbers[3] = 40;
        numbers[4] = 50;
        // numbers[5] = 60; // This would cause an ArrayIndexOutOfBoundsException!

        // Access and print elements
        System.out.println("Element at index 0: " + numbers[0]); // Output: 10
        System.out.println("Element at index 2: " + numbers[2]); // Output: 30

        // Get the length of the array
        System.out.println("Array length: " + numbers.length); // Output: 5

        // Loop through the array using a for loop
        System.out.println("\nAll elements:");
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("numbers[" + i + "]: " + numbers[i]);
        }

        // Declare and initialize an array with values directly
        String[] fruits = {"Apple", "Banana", "Cherry"};
        System.out.println("\nFruits:");
        for (String fruit : fruits) { // Enhanced for-loop (for-each)
            System.out.println(fruit);
        }

        // Another array example for doubles
        double[] prices = new double[3];
        prices[0] = 1.99;
        prices[1] = 2.49;
        prices[2] = 0.99;

        System.out.println("\nPrices:");
        for (int i = 0; i < prices.length; i++) {
            System.out.println("Price of item " + (i + 1) + ": $" + prices[i]);
        }
    }
}
```

2D Arrays

A **2D array** (or multi-dimensional array) is an array of arrays. It's often visualized as a grid or a table with rows and columns.

Declaration: `dataType[][] arrayName;`

Initialization: `arrayName = new dataType[numRows][numColumns];`

Declaration and Initialization: `dataType[][] arrayName = new dataType[numRows][numColumns];`

Initialization with values: `dataType[][] arrayName = {{val1, val2}, {val3, val4}, ...};`

Accessing elements: `arrayName[rowIndex][columnIndex]`

Example:

```
public class TwoDArraysExample {
    public static void main(String[] args) {
        // Declare and initialize a 2D array (3 rows, 4 columns)
        int[][] matrix = new int[3][4];

        // Assign values to elements
        matrix[0][0] = 10;
        matrix[0][1] = 20;
        matrix[0][2] = 30;
        matrix[0][3] = 40;

        matrix[1][0] = 50;
        matrix[1][1] = 60;
        matrix[1][2] = 70;
        matrix[1][3] = 80;

        matrix[2][0] = 90;
        matrix[2][1] = 100;
        matrix[2][2] = 110;
        matrix[2][3] = 120;

        System.out.println("Matrix elements (using nested loops):");
        // Loop through rows
        for (int i = 0; i < matrix.length; i++) {
            // Loop through columns of the current row
            for (int j = 0; j < matrix[i].length; j++) {
                System.out.print(matrix[i][j] + "\t"); // Print element and a tab
            }
            System.out.println(); // Move to the next line after each row
        }

        // Declare and initialize a 2D array with values directly
        String[][] studentGrades = {
            {"Alice", "A", "B"},
            {"Bob", "B", "C"},
            {"Charlie", "A", "A"}
```

```
        };

        System.out.println("\nStudent Grades:");
        for (int i = 0; i < studentGrades.length; i++) {
            System.out.print("Student: " + studentGrades[i][0] + ", Grades: ");
            for (int j = 1; j < studentGrades[i].length; j++) {
                System.out.print(studentGrades[i][j] + (j == studentGrades[i].length -
1 ? "" : ", "));
            }
            System.out.println();
        }

        // Jagged array (rows can have different number of columns)
        int[][] jaggedArray = new int[3][];
        jaggedArray[0] = new int[2]; // Row 0 has 2 columns
        jaggedArray[1] = new int[4]; // Row 1 has 4 columns
        jaggedArray[2] = new int[1]; // Row 2 has 1 column

        jaggedArray[0][0] = 1;
        jaggedArray[0][1] = 2;
        jaggedArray[1][0] = 3;
        jaggedArray[1][1] = 4;
        jaggedArray[1][2] = 5;
        jaggedArray[1][3] = 6;
        jaggedArray[2][0] = 7;

        System.out.println("\nJagged Array:");
        for (int i = 0; i < jaggedArray.length; i++) {
            for (int j = 0; j < jaggedArray[i].length; j++) {
                System.out.print(jaggedArray[i][j] + "\t");
            }
            System.out.println();
        }
    }
}
```
