

# Java Unit - 1

**Introduction to Java** : History and Features of Java, Java program structure, Writing simple Java class and main() method, Command-line arguments, Understanding JDK, JRE and JVM

**Data In the Cart** : Using primitive data types, Type conversion, Keywords, Identifiers, Variables, Access modifiers, static keyword, Wrapper class

**Operators** : Working with Bit-wise, arithmetic, logical, and relational operators, Unary, assignment and Ternary operator, Operator precedence

**Conditional Statements** : Using if/else constructs and switch-case statements

## Introduction to Java

### What is Java?

Java is a **high-level, object-oriented, and platform-independent programming language** designed to build robust, secure, and scalable applications. It was developed by **James Gosling** and his team at Sun Microsystems (now part of Oracle) in **1995**. Java is widely used across various domains, including web development, mobile applications, enterprise software, and more.

---

### History of Java

- **Origin and Development:**

Java was developed by **James Gosling** and his team at **Sun Microsystems** in the early 1990s. Initially called **Oak**, it was renamed Java in 1995 due to trademark issues. The language was designed to address the challenges of building software for networked environments and to ensure platform independence.

- **Key Milestones:**

- **1991:** Project started by the Green Team at Sun Microsystems.
  - **1995:** Java 1.0 was officially released, introducing the concept of "Write Once, Run Anywhere" (WORA).
  - **1997:** Java was standardized under the **Java Community Process (JCP)**.
  - **2006:** Java was open-sourced under the **GNU General Public License (GPL)**.
  - **2010:** Oracle Corporation acquired Sun Microsystems and took over Java development.
- 

### Java Language Used For:

1. Web development - Spring, hibernate

2. Mobile App
3. ERP
4. Desktop app - JavaFX, Applet
5. IOT devices
6. Game development - LWJGL
7. Banking Application - Paytm, CRED
8. Networking App - chat application (net package)

## Key Features of Java OR What Makes Java Unique and Why It's a Powerful Language

Java has been a cornerstone of the programming world since its inception in the mid-1990s. Let's dive into the key aspects that make Java unique and powerful:

---

### 1. Platform Independence

- **"Write Once, Run Anywhere (WORA)":**  
Java programs are compiled into an intermediate form called **bytecode**. This bytecode is platform-independent and runs on any device equipped with the **Java Virtual Machine (JVM)**.
    - Example: A program written and compiled on Windows can run seamlessly on Linux or macOS without any changes.
  - This feature is a game-changer, especially for large-scale software systems that need to operate across diverse environments.
- 

### 2. Object-Oriented Programming (OOP)

- Java is built around the principles of **Object-Oriented Programming**, which promotes reusability, scalability, and maintainability.
  - **Key OOP Concepts in Java:**
    1. **Encapsulation:** Wrapping data (fields) and methods in a single unit (class).
    2. **Inheritance:** Reusing existing code to create new functionality.
    3. **Polymorphism:** The ability to process objects differently based on their data type or class.
    4. **Abstraction:** Hiding complex implementation details and exposing only essential features.

- This makes Java suitable for modeling real-world problems and building robust applications.
- 

### 3. Robustness and Reliability

- Java is known for its emphasis on **error-checking** and **runtime exception handling**.
  - Features like **automatic memory management (Garbage Collection)** reduce memory leaks and improve program stability.
  - Strong type-checking during compilation helps catch potential bugs early.
- 

### 4. Secure Programming Environment

- Java was designed with security in mind:
    - The JVM isolates Java programs, preventing unauthorized access to system resources.
    - The Java security model includes features like **bytecode verification**, **sandboxing**, and built-in cryptographic algorithms.
  - These features make Java a preferred choice for applications that prioritize data security, such as banking and e-commerce platforms.
- 

### 5. Multithreading Support

- Java simplifies the creation of programs that can perform multiple tasks simultaneously using **threads**.
    - Example: A Java program can download a file, process data, and update the user interface at the same time.
  - This makes Java ideal for developing high-performance applications like games and multimedia software.
- 

### 6. Rich Standard Library (Java API)

- Java comes with an extensive library of pre-built classes and methods, called the **Java Standard Library** or **Java API**.
  - Examples:

- **java.util**: Collections, date-time handling, and utility classes.
    - **java.io**: Input and output stream handling.
    - **java.net**: Networking functionalities.
  - These libraries reduce development time and ensure developers don't have to "reinvent the wheel."
- 

## 7. Community Support and Ecosystem

- Java has one of the largest and most active developer communities.
  - A vast ecosystem of frameworks and tools is built around Java:
    - **Spring** and **Hibernate** for backend development.
    - **Apache Spark** for big data.
    - **Android SDK** for mobile app development.
- 

## 8. Performance and Scalability

- Java's Just-In-Time (JIT) Compiler optimizes bytecode execution at runtime, improving performance.
  - Java applications scale efficiently, making them ideal for enterprise-level solutions and cloud computing.
- 

## 9. Versatility Across Domains

- Java is a general-purpose language, which means it can be used in multiple domains:
    - **Web Applications**: Using frameworks like Spring or Struts.
    - **Mobile Development**: Primary language for Android development.
    - **Enterprise Systems**: Banking, ERP, and CRM solutions.
    - **IoT**: Lightweight Java libraries power embedded devices.
- 

## Java Program Structure:

Java programs are organized in a specific structure that makes them easy to read, maintain, and execute. Below is a detailed step-by-step explanation of the components that form the structure of a typical Java program.

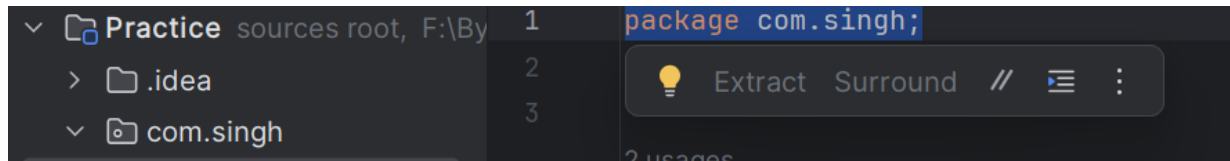
---

## Note:

1. Always ensure that if a class is declared **public**, its name matches the file name.
  2. If no class in the file is **public**, you can name the file anything you like, as long as it compiles correctly.
  3. For clarity and maintainability, it's a good practice to keep the file name the same as the main class name, even if it isn't required.
- 

### 1. Package Declaration (Optional)

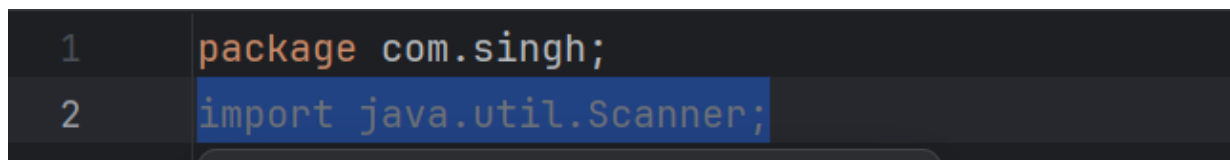
The package declaration specifies the namespace in which the classes are defined. This helps to organize classes and avoid name conflicts.



- **Purpose:** Groups related classes together.
  - **Optional:** Not required if the program doesn't use custom packages.
- 

### 2. Import Statements (Optional)

Import statements allow the program to include predefined or user-defined classes from other packages.



- **Purpose:** Provides access to classes and methods from libraries or other packages.
  - **Optional:** Not required if fully qualified names are used.
- 

### 3. Class Declaration

A Java program must have at least one class definition. The class name should match the filename if it is declared **public**.

```
3 public class BasicDetails {
    1 usage
4     public String name = "Ummmed Singh";
    no usages
5     int contact = 987654321;
6 }
```

- **Purpose:** Defines the blueprint of objects, methods, and variables.
  - **Mandatory:** Every Java program must have at least one class.
- 

#### 4. Main Method

The **main** method is the entry point of any Java program. The Java Virtual Machine (JVM) starts program execution from here.

```
3 public class BasicDetails {
4     public static void main(String[] args) {
5         System.out.println("Ummmed Singh");
6     }
7 }
```

**Purpose:** Acts as the starting point of the program.

- **Structure:**
    - **public:** Makes the method accessible to the JVM.
    - **static:** Allows the method to be executed without creating an instance of the class.
    - **void:** Indicates the method doesn't return a value.
    - **String[] args:** Accepts command-line arguments as an array of strings.
-

## 5. Comments (Optional)

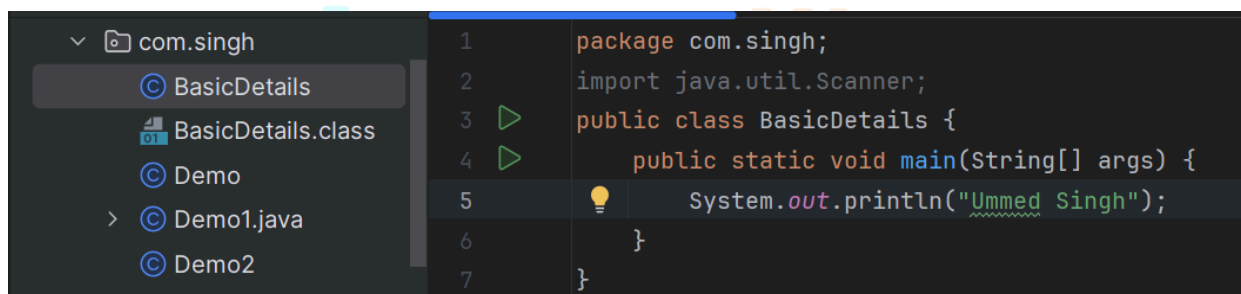
Comments enhance readability and provide information about the code. They are ignored by the compiler.

```
5      System.out.println("Ummed Singh");
6
7      //      Single line comment
8      /*      Multiline comment      */
9      }
```

- **Purpose:** Improves code maintainability.

---

## Writing simple java class and main() method:

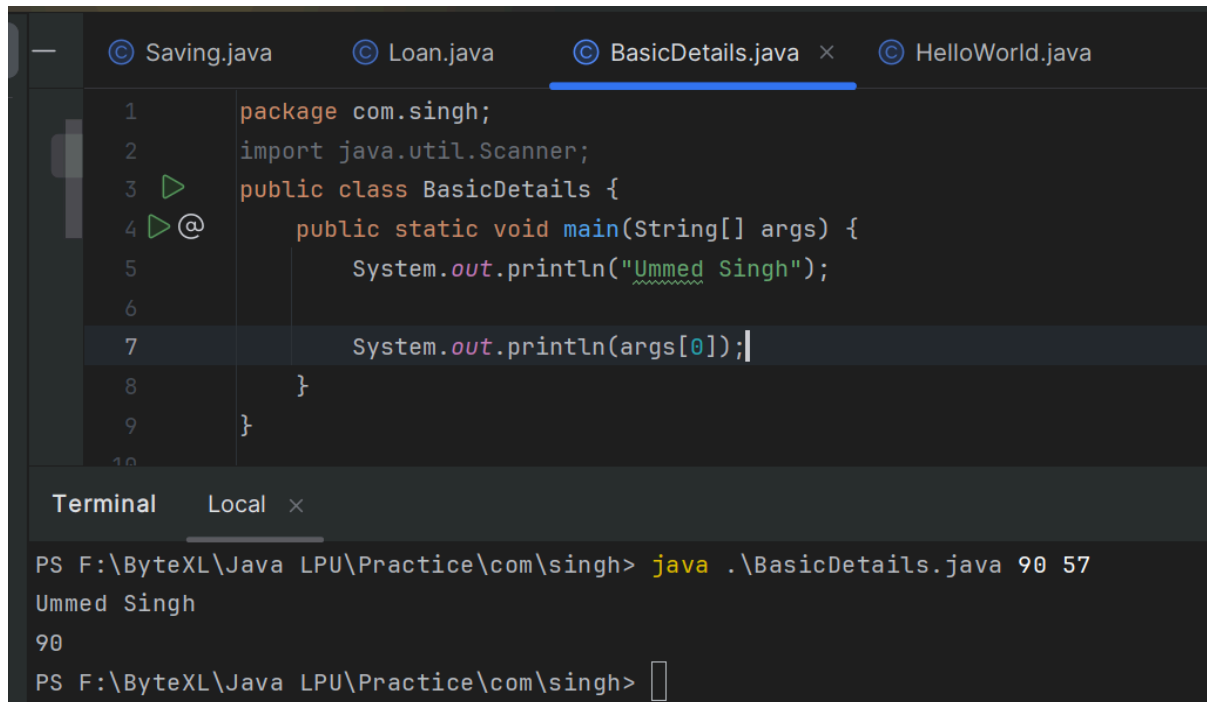


The screenshot shows an IDE with a project named 'com.singh'. The file explorer on the left lists 'BasicDetails', 'BasicDetails.class', 'Demo', 'Demo1.java', and 'Demo2'. The main editor displays the code for 'BasicDetails.java':

```
1      package com.singh;
2
3      import java.util.Scanner;
4
5      public class BasicDetails {
6
7          public static void main(String[] args) {
8
9              System.out.println("Ummed Singh");
10
11          }
12      }
```

---

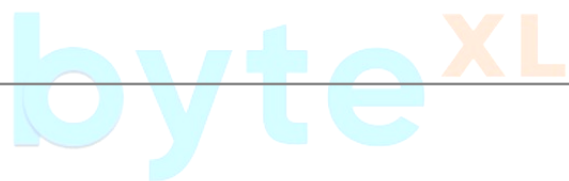
## Command line arguments:



The screenshot shows an IDE with four tabs: Saving.java, Loan.java, BasicDetails.java (active), and HelloWorld.java. The code in BasicDetails.java is as follows:

```
1 package com.singh;
2 import java.util.Scanner;
3 public class BasicDetails {
4     @ public static void main(String[] args) {
5         System.out.println("Ummmed Singh");
6
7         System.out.println(args[0]);
8     }
9 }
```

Below the code editor is a terminal window titled 'Terminal' with a 'Local' tab. It shows the command `java .\BasicDetails.java 90 57` being executed, resulting in the output `Ummmed Singh` and `90`.

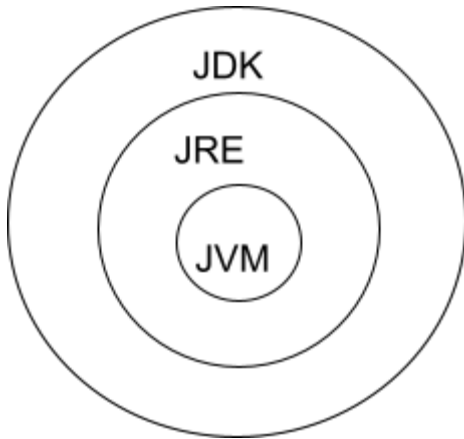


## Understanding JDK, JRE, JVM:

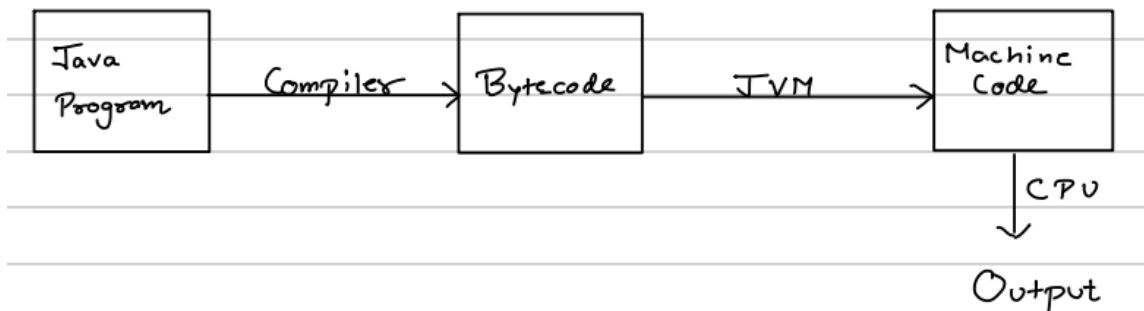
Java is a platform-independent programming language that relies on an ecosystem of tools and components. Three critical components in this ecosystem are **JDK**, **JRE**, and **JVM**.

### 1. JVM





- 
- The Java Virtual Machine (JVM) is a crucial part of the Java ecosystem that allows Java programs to run on any device or operating system. Understanding the JVM is essential for grasping how Java code is executed, optimized, and managed.
- It's just an abstract machine that doesn't exist physically
- It is responsible for converting compiled Java code into machine language that the host system's CPU can understand and execute.



- 
- JVM is platform dependent
- So we need to install a JVM based on the platform, i.e., macOS, Linux, or Windows. The input for JVM is bytecode, and the output is machine code. Now, since bytecode can be run by any JVM, it makes a Java program platform-independent.
- JVM has JIT (Just In Time) compilers, which take bytecode and convert it into machine code.
- It handles everything: loading code, executing it, managing memory, and cleaning up.

## Why is JVM Important?

- **Runs Everywhere:** It lets Java run on any device or OS.
- **Manages Memory:** It automatically takes care of memory, so you don't have to worry about freeing up space manually.
- **Optimizes Performance:** Uses techniques to make your program run faster.

## 2. JRE (Java Runtime Environment)

The JRE is a software package that provides the necessary environment to run Java programs.

- **Purpose:** Provides libraries and tools to execute Java applications.
- **Components:**
  - **JVM:**
    - The core of JRE, which runs Java programs.
  - **Core Libraries:**
    - Predefined classes and methods (e.g., `java.util`, `java.lang`).
  - **Supporting Files:**
    - Configuration files and resources required for execution.
- **What It Does Not Include:**
  - Development tools like the compiler (`javac`) or debugger.

**Key Point:** If you only want to run Java programs and not develop them, you only need the JRE.

---

### 3. JDK (Java Development Kit)

The JDK is a complete development environment for building Java applications.

- **Purpose:** Provides tools for developing, debugging, and monitoring Java programs.
- **Components:**
  1. **JRE:**
    - The JDK includes the JRE for running Java programs.
  2. **Development Tools:**
    - **Compiler (`javac`):** Converts Java source code into bytecode.
    - **Debugger (`jdb`):** Helps debug Java programs.
    - **JavaDoc:** Generates documentation from comments in the source code.
    - **JavaFX:** Tools for building graphical applications.
  3. **Other Utilities:**
    - `jar` (Java Archive tool), `jconsole` (monitoring tool), etc.

**Key Point:** If you want to write, compile, and debug Java programs, you need the JDK.

---

### 4. Relationship Between JDK, JRE, and JVM

- **JVM** is the foundation that runs Java programs.
- **JRE** includes the JVM and additional libraries to provide a runtime environment.
- **JDK** is a superset of JRE that also includes development tools.

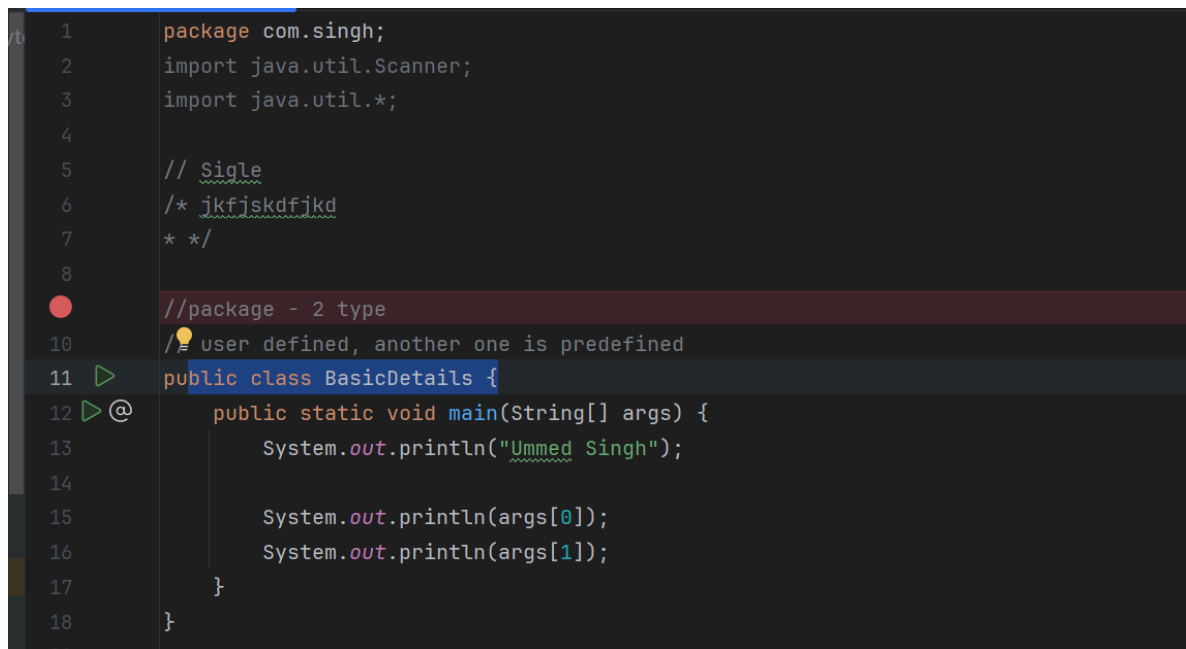
JDK = JRE + Development Tools

JRE = JVM + Libraries

---

## 5. Example Workflow

1. **Write Code:** Use a text editor or IDE to write `.java` files (requires JDK).
2. **Compile Code:** Use `javac` (Java compiler) to convert `.java` files into `.class` files (requires JDK).
3. **Run Code:** Use the `java` command to execute the `.class` files (requires JRE).



```
1 package com.singh;
2 import java.util.Scanner;
3 import java.util.*;
4
5 // Sigle
6 /* jkfiskdfjkd
7 * */
8
9 //package - 2 type
10 // user defined, another one is predefined
11 public class BasicDetails {
12     public static void main(String[] args) {
13         System.out.println("Ummed Singh");
14
15         System.out.println(args[0]);
16         System.out.println(args[1]);
17     }
18 }
```

- **Compilation:** `javac HelloWorld.java` → Generates `HelloWorld.class`.
- **Execution:** `java HelloWorld` → Runs the program using JVM.

## Data In the cart

### Data Types in Java

Data types specify the type of data that a variable can hold. Java is a strongly typed language, which means every variable must be declared with a data type before it can be used.

### Categories of Data Types

1. **Primitive Data Types:**

- The basic data types that are predefined by Java. They represent simple values and are not objects.
  - **Types:** `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`.
2. **Non-Primitive (Reference) Data Types:**
- These are not predefined; they are created by the programmer. They refer to objects and can hold multiple values.
  - **Examples:** `String`, `Arrays`, `Classes`, `Interfaces`.

## Detailed Breakdown of Primitive Data Types

1. **Integer Types:** Used to store whole numbers (no decimal).

Data Type	Size	Range	Example
<code>byte</code>	1 byte	-128 to 127	<code>byte b = 10;</code>
<code>short</code>	2 bytes	-32,768 to 32,767	<code>short s = 5000;</code>
<code>int</code>	4 bytes	-2,147,483,648 to 2,147,483,647	<code>int i = 100000;</code>
<code>long</code>	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	<code>long l = 100000L;</code>

2. **Floating-Point Types:** Used to store numbers with fractional parts (decimals)

Data Type	Size	Range	Example
<code>float</code>	4 bytes	Approximately $\pm 3.4e-038$ to $\pm 3.4e+038$	<code>float f = 3.14f;</code>
<code>double</code>	8 bytes	Approximately $\pm 1.7e-308$ to $\pm 1.7e+308$	<code>double d = 3.14;</code>

3. **Character Type:** Stores a single 16-bit Unicode character

Data Type	Size	Range	Example
<code>char</code>	2 bytes	0 to 65,535 (Unicode)	<code>char c = 'A';</code>

4. **Boolean Type:** Stores one of two values: `true` or `false`

Data Type	Size	Range	Example
<code>boolean</code>	1 bit	true or false	<code>boolean flag = true;</code>

## Non-Primitive Data Types

1. **Strings:** A sequence of characters used for text. Strings are objects in Java, and they have built-in methods for manipulation.
2. **Arrays:** A collection of variables of the same type stored in contiguous memory locations.
3. **Classes and Objects:** Classes are blueprints for creating objects. Objects are instances of classes and represent entities with attributes and behaviors.

```
public class DataTypesDemo { ByteXL

    public static void main(String[] args) {

// Primitive data types

        int age = 25;

        double salary = 55000.75;

        char grade = 'A';

        boolean isEmployed = true;

// Non-primitive data type

        String name = "Ummed Singh";

// Output the values

        System.out.println("Name: " + name);

        System.out.println("Age: " + age);

        System.out.println("Salary: " + salary);

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

        System.out.println("Employed: " + isEmployed);

    }

}
```

## Variables in Java

Variables are containers for storing data values. In Java, variables must be declared with a specific type, defining the kind of data they can hold. Variables are essential as they allow programs to store and manipulate data.

## Types of Variables in Java

### 1. Local Variables:

- **Definition:** Declared inside a method, constructor, or block and only accessible within that scope.
- **Scope:** Limited to the method or block in which they are defined.
- **Initialization:** Must be initialized before use.

```
public void display() {                                Ummed Singh
//            local variable                            ByteXL

    int lv=10;

    System.out.println(gv);

    System.out.println(lv);

}
```

○

### 2. Instance Variables:

- **Definition:** Declared inside a class but outside any method, they represent the properties of an object.
- **Scope:** Accessible throughout the class and with each object having its own copy.
- **Initialization:** Automatically initialized to default values (0, null, false, etc.) if not explicitly initialized.

```
public class Variables {                                ByteXL

//            Instance/Global Variable

    int gv;
```

### 3. Static (Class) Variables:

- **Definition:** Declared with the `static` keyword inside a class but outside methods. Shared among all instances of the class.

- ```
public class Variables {                                     Ummed Singh

    //      static variable

    static int count = 0;

}
```

- **Declaration:** Defines the variable type and name.
  - Syntax: `dataType variableName;`
  - Example: `int age;`
- **Initialization:** Assigns a value to the declared variable.
  - Syntax: `variableName = value;`
  - Example: `age = 25;`
- **Combined Declaration and Initialization:**
  - Syntax: `dataType variableName = value`
  - Example: `int age = 25;`

1. **Start with a Letter:** Variable names must start with a letter, underscore (`_`), or dollar sign (`$`), but not a number.
2. **Case Sensitivity:** Java is case-sensitive; `Age` and `age` are different variables.
3. **Descriptive Names:** Use meaningful names that describe the variable's purpose, like `studentAge` instead of `x`.

- **Scope:** Refers to the part of the program where the variable can be accessed.
- **Lifetime:** Refers to how long the variable exists in memory. Local variables last until the method exits, instance variables last as long as the object exists, and static variables last for the program's lifetime.

```
public class Variables {                                     ByteXL  
  
    // Instance variable
```

```

String name = "Ummmed Singh";

// Static variable

static int count = 0;

public void display() {
// Local variable

    int age = 25;

    count++; // Increment static variable

    System.out.println("Name: " + name);

    System.out.println("Age: " + age);

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

}

public static void main(String[] args) {

    Variables demo1 = new Variables();

    demo1.display();

    Variables demo2 = new Variables();

    demo2.display();

}

}

```

## Type Casting/conversion in Java

Type casting is converting a variable from one data type to another. There are two types of casting:

### 1. Implicit (Widening) Casting

- Automatically done by the compiler when converting a smaller type to a larger type.



- No data loss occurs.

```
byte b = 10;
int i = b;
long l = i;
System.out.println(l);
```

Ummmed Singh

- byte to short, int, long, float, and double
- char to int, long, float, and double
- int to long, float, and double
- long to float and double
- float to double

## 2. Explicit (Narrowing) Casting

- Must be done manually when converting a larger type to a smaller type.
- Risk of data loss.
- Use parentheses to specify the type.
- Example:

```
int a = 129;
byte b = (byte) a; // -128, -127.....0, 1,.....127
System.out.println(b); // a - b => 129 - 127 = 7
// OUTPUT => -127
```

ByteXL

- double to float, long, int, short, byte, and char
- float to long, int, short, byte, and char
- long to int, short, and byte
- int to short, byte, and char
- short to byte

## Automatic Type Promotions in Expressions

Automatic type promotion in expressions is a concept in Java where smaller data types are automatically promoted to larger data types during arithmetic operations. This ensures that calculations are done accurately without data loss or errors.

Java promotes smaller data types to larger ones when evaluating expressions. The promotion rules are applied automatically based on the hierarchy of data types.

## 1. Type Promotion Rules

### 1. Byte, Short, and Char Promotion:

- `byte`, `short`, and `char` are automatically promoted to `int` when used in arithmetic operations.

### 2. Promotion to Larger Types:

- If one operand is `long`, the whole expression is promoted to `long`.
- If one operand is `float`, the whole expression is promoted to `float`.
- If one operand is `double`, the whole expression is promoted to `double`.

### 3. Mixed-Type Expressions:

- The promotion follows the hierarchy: `byte` → `short` → `int` → `long` → `float` → `double`.

## 2. Examples of Automatic Type Promotion

### 1. Promotion of `byte`, `short`, and `char` to `int`

```
byte b1 = 10;
byte b2 = 20;

// Both b1 and b2 are promoted to int before addition
int result = b1 + b2;

System.out.println("byte to int: "+ result); // Output: 30
```

### 2. Mixed Data Types in Expressions

```
int i = 10;
long l = 20L;

// i is promoted to long, and result is long
long result = i + l ;

System.out.println("int to long: "+ result); // Output: 30
```

### 3. Promotion to float

```
int i = 10;                                Ummmed Singh

float f = 3.14f;                            ByteXL

float result = i + f; // i is promoted to float, and result
is float

System.out.println("int to float: "+ result); // Output:
13.14
```

### 4. Promotion to double

```
float f = 2.5f;                              Ummmed Singh

double d = 4.5;                              ByteXL

double result = f + d; // f is promoted to double, and
result is double

System.out.println("float to double: " + result); //
Output: 7.0
```

### 5. char Promotion in Expressions

```
char c1 = 'A'; // Unicode value is 65        Ummmed Singh

char c2 = 'B'; // Unicode value is 66        ByteXL

int result = c1 + c2; // Both c1 and c2 are promoted to int

System.out.println("char to int: " + result); // Output:
131
```

### 6. Complex Expression with Multiple Promotions

```
byte b = 50;                                Ummmed Singh
```

```

char c = 'C'; // Unicode value is 67
int i = 1000;
long l = 1500001;
float f = 5.67f;
double d = 0.1234;

// Promotion hierarchy in this expression:
// b> int, c> int, int> long, long -> float, float ->
double

double result = (f*b) + (i/c) - (d* l);

System.out.println("Complex promotion: "+ result);

// Output: 283.47660000000003 (may vary slightly due to
floating-point arithmetic)

```

### 3. Keywords

Java keywords are reserved words that have predefined meanings in the language. They serve specific purposes and are an integral part of the syntax. Examples include:

| Keyword             | Purpose                                       |
|---------------------|-----------------------------------------------|
| <code>class</code>  | Declares a class                              |
| <code>public</code> | Access modifier for public visibility         |
| <code>static</code> | Indicates a static method or variable         |
| <code>return</code> | Exits a method and optionally returns a value |
| <code>void</code>   | Specifies a method with no return value       |
| <code>if</code>     | Begins a conditional branch                   |
| <code>else</code>   | Defines an alternate path in a conditional    |

|                    |                                                       |
|--------------------|-------------------------------------------------------|
| <code>while</code> | Starts a loop that runs while a condition is true     |
| <code>for</code>   | Starts a loop with an iterator                        |
| <code>try</code>   | Begins a block to test for exceptions                 |
| <code>catch</code> | Handles exceptions raised in a <code>try</code> block |

### Key Characteristics of Keywords

- There are over 50 reserved keywords in Java.
- They are case-sensitive.
- They cannot be used as identifiers for variables, methods, or classes.

Example of usage:

```

3 ▶ public class Keywords {
4 ▶     public static void main(String[] args) {
5         int number = 5; // 'int' is a keyword
6         if (number > 0) { // 'if' is a keyword
7             System.out.println("Positive number");
8         }
9     }
10 }

```

## 4. Identifiers

Identifiers are names used to identify variables, methods, classes, or other entities in Java. They are essential for naming and organizing elements in a program. Below are the key rules and additional details for identifiers:

### Rules for Identifiers:

- 1. Start Character:**
  - Must begin with a letter (A-Z or a-z), underscore (`_`), or dollar sign (`$`).
  - Cannot start with a digit (0-9).
- 2. Subsequent Characters:**
  - Can include letters, digits, underscores, and dollar signs.
  - Spaces are not allowed within an identifier.
- 3. Reserved Words:**

- Cannot use Java keywords or reserved words as identifiers (e.g., `class`, `int`, `public`).
4. **Case Sensitivity:**
- Identifiers are case-sensitive. For example, `Variable` and `variable` are considered different identifiers.
5. **Length and Readability:**
- While there is no strict limit on length, it is recommended to use meaningful and concise names for better readability.

### Best Practices for Identifiers:

- Use **camelCase** for variable and method names (e.g., `studentAge`, `calculateSum`).
- Use **PascalCase** for class names (e.g., `StudentDetails`, `InvoiceProcessor`).
- Avoid using single-character names except for loop counters or temporary variables.
- Use underscores sparingly, primarily in constants (e.g., `MAX_SPEED`).

### Examples:

#### Valid Identifiers:

```
public class Identifiers {                                // ByteXL

    int age = 25;  // 'age' is a valid identifier

    String $name = "Ummad Singh"; // '$name' is valid

    float _price = 19.99f; // '_price' is valid

}
```

#### Invalid Identifiers:

```
int 2ndNumber = 5;    // Invalid: Cannot start with a digit

String class = "Test"; // Invalid: 'class' is a reserved word

int first name = 10;   // Invalid: Spaces are not allowed
```

By following these rules and best practices, you can create identifiers that are both functional and maintainable, ensuring code clarity and consistency.

# Access Modifiers

Access modifiers in Java control the visibility and accessibility of classes, methods, and variables. They play a crucial role in defining the boundaries of an object-oriented program and ensuring encapsulation. Java provides four types of access modifiers:

## Types of Access Modifiers

### 1. Public

- **Visibility:** Accessible from any other class, regardless of the package.
- **Use Case:** When methods, variables, or classes need to be universally accessible.

Example:

```
public class AccessModifiers {                                Umed Singh  
  
    public int a = 10;          // public                      ByteXL  
  
    public void display() {  
        System.out.println(a);  
    }  
}
```

### 2. Private

- **Visibility:** Accessible only within the same class.
- **Use Case:** Used to enforce strict encapsulation and prevent access to sensitive data or methods from outside the class.

Example:

```
public class AccessModifiers {                                Umed Singh  
  
    private int a = 10;          // private                    ByteXL  
  
    public void display() {  
        System.out.println(a);  
    }  
}
```

```
}  
  
}
```

### 3. Protected

- **Visibility:** Accessible within the same package and subclasses (even in different packages).
- **Use Case:** Useful when sharing data with subclasses while restricting access to the rest of the program.

**Example:**

```
public class AccessModifiers {                                Ummed Singh  
  
    protected int value = 10;    // protected                ByteXL  
  
}  
  
class Subclass extends AccessModifiers {  
  
    void display() {  
  
        System.out.println("Value: " + value);  
  
    }  
  
}
```

### 4. Default (Package-Private)

- **Visibility:** Accessible only within the same package.
- **Use Case:** Ideal for package-level access where classes or members should not be exposed outside the package.

**Example:**

```
public class AccessModifiers {                                Ummed Singh  
  
    int a = 10;        // default                            ByteXL
```



```
public void display() {  
    System.out.println(a);  
}  
}
```

---

## Summary of Access Levels

| Modifier  | Same Class | Same Package | Subclass (Different Package) | Other Packages |
|-----------|------------|--------------|------------------------------|----------------|
| public    | Yes        | Yes          | Yes                          | Yes            |
| protected | Yes        | Yes          | Yes                          | No             |
| default   | Yes        | Yes          | No                           | No             |
| private   | Yes        | No           | No                           | No             |

---

## Best Practices

- Use **private** by default for member variables and provide controlled access through getter and setter methods.
- Use **protected** when designing classes for inheritance.
- Use **public** only when methods or variables are intended to be universally accessible.
- Avoid using the default modifier unless package-level access is explicitly required.

# The **static** Keyword in Java

The **static** keyword in Java is used to define class-level variables, methods, or blocks that can be accessed without creating an instance of the class. It is widely used for defining shared resources or utilities.

## Features of the **static** Keyword

1. **Class-Level Association:**
    - **static** members belong to the class rather than any specific instance.
  2. **Memory Management:**
    - **static** members are created once and shared among all instances, reducing memory usage.
  3. **Direct Access:**
    - **static** members can be accessed using the class name directly.
- 

## Usage of the **static** Keyword

### 1. Static Variables

- Used to define variables shared among all instances of a class.
- Useful for constants or counters.

Example:

```
public class Counter {    // cntNmb                                Ummed Singh

    static int cntNmb = 0;                                ByteXL

    int a;

    public void countNumber() {

        cntNmb++;

        int b = 10;

        System.out.println(cntNmb) ;

    }
```

```
}
```

```
public class BasicDetails {                                Ummmed Singh

    public static void main(String[] args) {                ByteXL

        Counter obj = new Counter(); //

        obj.countNumber();

        Counter obj1 = new Counter();

        obj1.countNumber();

    }

}
```

byteXL

## 2. Static Methods

- Used to define utility or helper methods that do not depend on instance variables.

Example:

```
AccessModifiers accObj = new AccessModifiers();            Ummmed Singh

System.out.println(AccessModifiers.a);
```

## 3. Static Blocks

- Executed when the class is loaded.
- Typically used for initializing static variables.

Example:

```
class StaticBlockExample {

    static int number;                                       Ummmed Singh

}
```

```

static {
    number = 10;

    System.out.println("Static block executed.");
}

public static void main(String[] args) {
    System.out.println("Number: " + number);
}
}

/* OUTPUT

    Static block executed.

    Number: 10

*/

```

## 4. Static Classes

- Inner classes can be declared as **static**, allowing them to be accessed without an outer class instance.

### Example:

```

class OuterClass {
    static class StaticInnerClass {
        void display() {
            System.out.println("Static Inner Class called.");
        }
    }
}

```

```
public class Test {  
  
    public static void main(String[] args) {  
  
        OuterClass.StaticInnerClass inner = new  
OuterClass.StaticInnerClass();  
  
        inner.display();  
  
    }  
  
}
```

## Key Points

- A **static** method cannot access non-static variables or methods directly.
- **this** and **super** cannot be used in static contexts.
- Static members are initialized once and persist throughout the program's lifetime.
- 

## Wrapper Classes in Java

### 1. What are Wrapper Classes?

Wrapper classes in Java provide a way to use primitive data types (like **int**, **double**, **boolean**) as objects. These classes are part of the **java.lang** package and are essential for converting primitive types into objects and vice versa. Each primitive type has a corresponding wrapper class:

| Primitive Type | Wrapper Class |
|----------------|---------------|
| <b>byte</b>    | <b>Byte</b>   |
| <b>short</b>   | <b>Short</b>  |

|         |           |
|---------|-----------|
| int     | Integer   |
| long    | Long      |
| float   | Float     |
| double  | Double    |
| char    | Character |
| boolean | Boolean   |

---

## 2. Why Use Wrapper Classes?

- **Object-Oriented Programming:** Collections in Java (like `ArrayList`, `HashMap`) work with objects, not primitives.
- **Utility Methods:** Wrapper classes provide methods to manipulate and perform operations on primitive types (e.g., parsing strings to numbers).
- Ex: `int a = 10` -> `"10"`
- **Autoboxing and Unboxing:** These features simplify the conversion between primitives and their corresponding wrapper classes.

---

## 3. Features of Wrapper Classes

- **Immutability:** Wrapper objects are immutable, meaning their values cannot be changed once created.
- **Caching:** Certain wrapper classes (e.g., `Integer`) cache frequently used values for better performance.
- **Interoperability:** Convert between data types and parse data from strings (e.g., `Integer.parseInt()`).

---

## 4. Examples

### a. Autoboxing and Unboxing

```
public class WrapperDemo {                                Ummed Singh

    public static void main(String[] args) {                ByteXL

        // Autoboxing: Primitive to Wrapper

        int primitive = 10;

        Integer wrapped = primitive;

        // Unboxing: Wrapper to Primitive

        Integer wrappedValue = 20;

        int primitiveValue = wrappedValue;

        System.out.println("Primitive: " + primitive + ",
Wrapped: " + wrapped);

        System.out.println("Wrapped: " + wrappedValue + ",
Primitive: " + primitiveValue);

    }

}
```

### b. Using Utility Methods

```
public class UtilityMethodDemo {                            Ummed Singh

    public static void main(String[] args) {                ByteXL

        String number = "123";

        // Convert String to Integer

        int num = Integer.parseInt(number);

        // Convert Integer to String
```

```

        String numStr = Integer.toString(num);

        System.out.println("String to Integer: " + num);

        System.out.println("Integer to String: " + numStr);

    }
}

```

### c. Wrapper Classes in Collections

```

public class WrapperInCollections {                                Ummed Singh

    public static void main(String[] args) {                        ByteXL

        ArrayList<Integer> list = new ArrayList<>();

        // Adding elements (Autoboxing)

        list.add(10);

        list.add(20);

        // Retrieving elements (Unboxing)

        int first = list.get(0);

        System.out.println("List: " + list);

        System.out.println("First Element: " + first);

    }

}

```

---

## 5. Key Methods of Wrapper Classes

Each wrapper class provides several methods. Here are some examples:



| Wrapper Class | Common Methods         | Description                                       |
|---------------|------------------------|---------------------------------------------------|
| Integer       | parseInt(String s)     | Converts a string to an integer.                  |
|               | valueOf(String s)      | Returns an Integer object representing the value. |
| Double        | parseDouble(String s)  | Converts a string to a double.                    |
|               | doubleValue()          | Returns the value as a primitive double.          |
| Boolean       | parseBoolean(String s) | Parses a string to a boolean.                     |
|               | booleanValue()         | Returns the value as a primitive boolean.         |
| Character     | isDigit(char ch)       | Checks if the character is a digit.               |
|               | isLetter(char ch)      | Checks if the character is a letter.              |

---

## 6. Common Use Cases

- Data Structures:** Using wrapper classes in `ArrayList`, `HashMap`, etc.
- Parsing Data:** Converting user input (String) to primitive types.
- Default Values:** Using null for uninitialized objects.

---

## 7. Performance Considerations

- **Memory Usage:** Wrapper classes consume more memory than primitives.
  - **Autoboxing/Unboxing Overhead:** May lead to performance issues in critical applications if overused.
- 

# Operators

Operators are special symbols that perform operations on variables and values.

## Detailed Explanation of Each Operator

**1. Arithmetic Operators:** Arithmetic operators are used to perform basic mathematical operations.

- **Operators:** + (Addition), - (Subtraction), \* (Multiplication), / (Division), % (Modulus)

```
int a = 10;
int b = 5;

System.out.println("Addition: " + (a + b)); // 15
System.out.println("Subtraction: " + (a - b)); // 5
System.out.println("Multiplication: " + (a * b)); // 50
System.out.println("Division: " + (a / b)); // 2
System.out.println("Modulus: " + (a % b)); // 0
}
```

**2. Relational Operators:** Relational operators compare two values and return a boolean result (**true** or **false**).

- **Operators:** == (Equal to), != (Not equal to), > (Greater than), < (Less than), >= (Greater than or equal to), <= (Less than or equal to)

```
int a = 10;                                Ummed Singh
int b = 20;                                ByteXL

System.out.println(a == b); // false
System.out.println(a != b); // true
System.out.println(a > b); // false
System.out.println(a < b); // true
```

**3. Logical Operators:** Logical operators are used to combine multiple conditions.

- **Operators:** `&&` (Logical AND), `||` (Logical OR), `!` (Logical NOT)

```
boolean a = true;                          Ummed Singh
boolean b = false;                         ByteXL

System.out.println(a && b); // false
System.out.println(a || b); // true
System.out.println(!a); // false
```

**4. Bitwise Operators** Bitwise operators work on bits and perform bit-level operations.

- **Operators:** `&` (Bitwise AND), `|` (Bitwise OR), `^` (Bitwise XOR), `~` (Bitwise Complement), `<<` (Left shift), `>>` (Right shift), `>>>` (Unsigned right shift)

```
int a = 5; // 0101 in binary                Ummed Singh
int b = 3; // 0011 in binary                ByteXL

System.out.println(a & b); // 1 (0001)
System.out.println(a | b); // 7 (0111)      Ummed Singh
System.out.println(a ^ b); // 6 (0110)      ByteXL
System.out.println(~a); // -6
```

**5. Assignment Operators:** Assignment operators assign values to variables.

- **Operators:** `=`, `+=`, `-=`, `*=`, `/=`, `%=`

```
int c = 10;                                     Ummed Singh
c += 5; // c = c + 5                             ByteXL
System.out.println(c); // 15
```

**6. Unary Operators:** Unary operators work with a single operand.

- **Operators:** `+` (Unary plus), `-` (Unary minus), `++` (Increment), `--` (Decrement), `!` (Logical NOT)

```
int d = 5;                                     Ummed Singh
System.out.println(++d); // 6                  ByteXL

System.out.println(d--); // 6
System.out.println(-d); // -5
```

**7. Ternary Operator:** The ternary operator is a shorthand for an `if-else` statement.

- **Syntax:** `condition ? if-true : if-false`

```
int e = 10;                                     Ummed Singh
int f = (e > 5) ? 100 : 200;                   ByteXL
System.out.println(f);
```

## Practical Activities

- **Activity 1:** Create a calculator program using arithmetic operators.

- **Activity 2:** Write a program to determine the largest of three numbers using relational and logical operators.
- **Activity 3:** Implement a simple grading system using the ternary operator.

## 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.

### Why is Operator Precedence Important?

Understanding operator precedence helps in predicting how an expression will be evaluated and allows writing expressions that produce the intended results.

### Hierarchy of Operator Precedence

Below is the list of Java operators arranged in decreasing order of precedence:

1. **Parentheses:** `()`, `[]`
2. **Postfix Operators:** `expr++`, `expr--`
3. **Unary Operators:** `++expr`, `--expr`, `+`, `-`, `~`, `!`
4. **Multiplicative Operators:** `*`, `/`, `%`
5. **Additive Operators:** `+`, `-`
6. **Shift Operators:** `<<`, `>>`, `>>>`
7. **Relational Operators:** `<`, `>`, `<=`, `>=`, `instanceof`
8. **Equality Operators:** `==`, `!=`
9. **Bitwise AND:** `&`
10. **Bitwise XOR:** `^`
11. **Bitwise OR:** `|`
12. **Logical AND:** `&&`
13. **Logical OR:** `||`
14. **Ternary Operator:** `?:`
15. **Assignment Operators:** `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`

```
int x = 5+2 * 3; // Multiplication has higher precedence than
addition

System.out.println(x);
```

```
int y = (5+2) * 3; // Parentheses change the order of operations
System.out.println(y); // 21
```

## Selection Statements in Java

Selection statements control the flow of execution based on certain conditions. They allow the program to make decisions and execute different code paths.

### Types of Selection Statements in Java

1. **if Statement**
2. **if-else Statement**
3. **if-else-if Ladder**
4. **switch Statement**

### Detailed Explanation with Examples

#### 1. **if Statement**

The **if** statement evaluates a condition and executes a block of code if the condition is **true**.

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

int number = 10;

if(number > 5){
    System.out.println("The number is greater than 5.");
}
```

#### 2. **if-else Statement**

The **if-else** statement provides an alternative path if the condition is **false**.

```

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

int number = 3;
if (number % 2 == 0) {
    System.out.println("The number is even.");
}else {
    System.out.println("The number is odd");
}

```

### 3. if-else-if Ladder

The if-else-if ladder is used when there are multiple conditions to test.

Syntax:

```

if (condition1) {
    // Code if condition1 is true
} else if (condition2) {
    // Code if condition2 is true
} else {
    // Code if all conditions are false
}

```

Example:

```

int marks = 85;

if (marks >= 90) {

```

```

    System.out.println("Grade: A");
} else if (marks >= 75) {
    System.out.println("Grade: B");
} else if (marks >= 50) {
    System.out.println("Grade: C");
} else {
    System.out.println("Grade: F");
}

```

#### 4. switch Statement

The `switch` statement is used when you need to select one among many blocks of code based on a value.

**Syntax:**

```
switch (expression) {
```

```

    case value1:
        // Code block for value1
        break;
    case value2:
        // Code block for value2
        break;
        // Add more cases as needed
    default:
        // Code if none of the cases match
}

```



Example:

```
int day = 3;
```

```
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
}
```

Ummed Singh

ByteXL

## Key Points to Emphasize

- Use **if-else** statements when checking conditions that require logical evaluations.
- Use **switch** statements when checking against specific values of an integer, character, string, or enumerated type for better readability and maintainability.
- **break Statement in switch:** The **break** statement is crucial to prevent fall-through behavior, where subsequent cases execute without checking their conditions.

## Common Pitfalls

- **Missing break in switch statements:** This can lead to unintentional fall-through, executing multiple cases.

- **Incorrect placement of brackets:** Ensure proper use of curly braces `{ }` to define code blocks.
- **Overuse of nested `if` statements:** Nested `if` statements can make code complex and harder to read. Use `if-else-if` or `switch` for clarity.

## Practical Activities

- **Activity 1:** Write a program to determine if a number is positive, negative, or zero using an `if-else-if` ladder.
- **Activity 2:** Create a menu-driven program using a `switch` statement that performs different arithmetic operations based on user input.

