Programming With java Cs-22

James Gosling *Oak*
Sun Microsystems

- Java was developed by James Gosling and his team at Sun Microsystems in 1991
- Initially, it was called *Oak*, named after an oak tree outside Gosling's office. Later, it was renamed *Java*, inspired by Java coffee.
- The primary goal was to create a platform-independent language for consumer electronic devices.

Release Timeline

- 1995: Official launch of Java 1.0 with the tagline "Write Once, Run Anywhere" (WORA).
- 1997: Java 1.1 introduced new features like JDBC (Java Database Connectivity).
- 2004: Java 5.0 introduced generics, annotations, and enhanced for-loops.

- 2006: Sun Microsystems made Java open source under the GNU General Public License.
- 2010: Oracle Corporation acquired Sun Microsystems and took over Java's development.
- Present: Java has evolved with regular updates, the latest being Java 21 (released in September 2023).

Key Features of Java

- **Platform Independence**:
- Java programs are compiled into bytecode by the Java Compiler. This bytecode can run on any device with a Java Virtual Machine (JVM), regardless of the underlying hardware or operating system.
- **Object-Oriented:**
- java is built around the concepts of objects and classes, promoting reusability and modularity. Core principles include encapsulation, inheritance, and polymorphism.
- **Simple and Familiar:**
- Java was designed to be easy to learn, especially for developers familiar with C and C++. It removes complexities like pointers and manual memory management.
- **Secure**:Java provides built-in security features such as bytecode verification, a secure class-loading mechanism, and a security manager to control access to resources.
- **Robust**:Java emphasizes error checking and runtime error handling. It eliminates issues like memory leaks by managing memory through garbage collection.
- **Rich API:**Java offers a comprehensive set of libraries for data structures, networking, file I/O, database access (JDBC), and more.
- **Scalability**:Java is suitable for building applications of all sizes, from small utilities to large-scale enterprise systems.
-

Java Edition

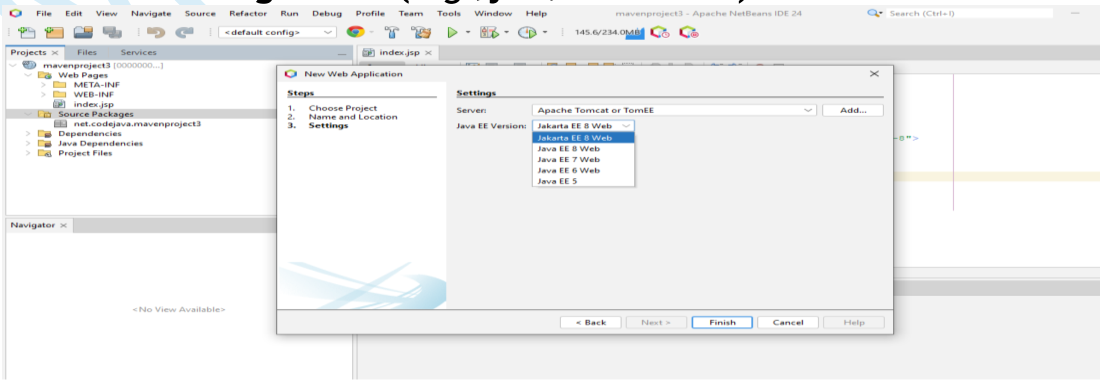
- Java has several editions, each designed for different types of applications. The main editions are

Java Standard Edition (Java SE)

- **Purpose**: Provides the core functionality of the Java programming language.
- **Key Features**
- Core libraries for basic programming.
- **Includes APIs for:**
- **Data structures** (e.g., Collections framework).
- **Networking** (e.g., Sockets, HTTP).
- Multithreading and concurrency.
- Input/Output operations.
- Database connectivity (JDBC).
- **JVM** (Java Virtual Machine) for running Java applications.
- Includes tools like javac (compiler), java (interpreter), and debugging tools.
- Use Cases:
- Desktop applications.
- Basic standalone programs.

Java Enterprise Edition (Java EE) (Now Jakarta EE)

- **Purpose**: Designed for developing large-scale, distributed, and enterprise-level applications.
- Adds frameworks and libraries on top of Java SE.
- APIs for:
- **Web development** (e.g., Servlets, JSP, JSF).
- **Enterprise applications** (e.g., EJB).
- **Web services** (REST and SOAP).
- **Messaging** (e.g., JMS).
- **Database integration** (e.g., JPA, Hibernate).



mavenproject3 [0000000...]
Web Pages
META-INF
WEB-INF
index.jsp
Source Packages
net.codejava.mavenproject3
Dependencies
Java Dependencies
Project Files

New Web Application

Steps
1. Choose Project
2. Name and Location
3. **Settings**

Settings
Server: Apache Tomcat or TomEE
Java EE Version: Jakarta EE 8 Web
Jakarta EE 8 Web
Java EE 8 Web
Java EE 7 Web
Java EE 6 Web
Java EE 5

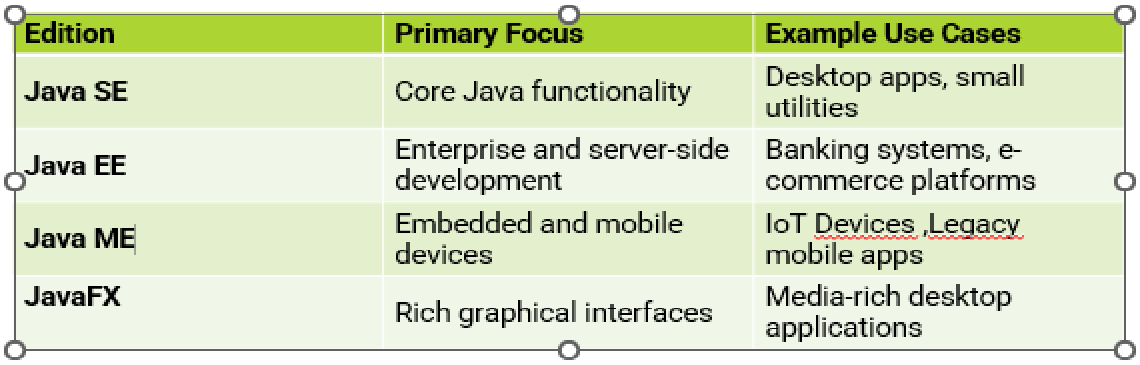
- **Use Cases**:
- Enterprise-level systems (e.g., banking, e-commerce).
- Server-side applications.

Java Micro Edition (Java ME)

- **Purpose**: Designed for resource-constrained devices.
- Lightweight libraries optimized for mobile and embedded devices.
- APIs for:
- **Graphics**.
- **Networking on limited resources.**
- **Device-specific functionalities (e.g., sensors, mobile hardware).**
- Supports Connected Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP).
- Use Cases:
- Legacy mobile applications (prior to moEmbedded systems (e.g., IoT devices, set-top boxes).
- dern smartphones).

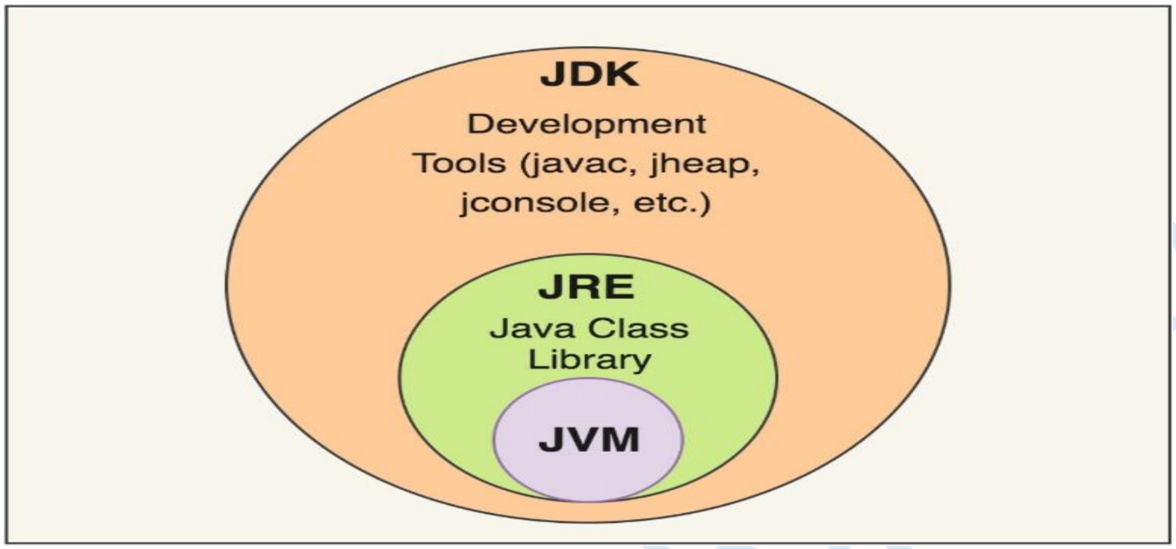
JavaFX

- **Purpose**: A platform for creating rich graphical user interfaces (GUIs).
- APIs for:
- 2D and 3D graphics.
- Media playback.
- Web integration.
- Animation and effects.
- Replaces the older Swing and AWT(Abstract Window Toolkit.) libraries for GUI development.
- Uses an XML-based language called FXML (XML-based user interface markup language)e for defining UI components.
- Use Cases:
- Desktop applications with rich multimedia and graphical interfaces.



Edition	Primary Focus	Example Use Cases
Java SE	Core Java functionality	Desktop apps, small utilities
Java EE	Enterprise and server-side development	Banking systems, e-commerce platforms
Java ME	Embedded and mobile devices	IoT Devices ,Legacy mobile apps
JavaFX	Rich graphical interfaces	Media-rich desktop applications

JDK, JVM and JRE



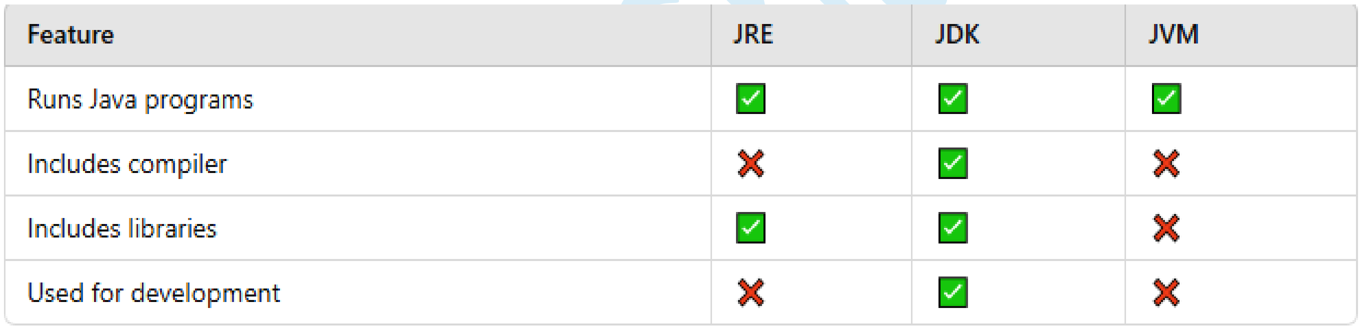
- JDK stands for Java Development Kit, which is a software package that provides tools and utilities for developing, testing, and deploying Java applications
- The JDK is available for Windows, macOS, Linux, and Solaris. You can download the JDK installation file from the Oracle website
-

JVM (Java Virtual Machine)

- Purpose: The JVM is an abstract machine that executes Java bytecode.
- Role:
- Converts compiled bytecode (.class files) into machine code for execution.
- Provides a platform-independent runtime by abstracting the underlying operating system.
- Features:
- Garbage collection
- Just-In-Time (JIT) compilation
- Memory management

JRE (Java Runtime Environment)

- **JRE (Java Runtime Environment)** is a part of the Java Development Kit (JDK) that provides the necessary components to run Java applications. It does **not** include development tools like compilers or debuggers; it's only for running Java programs.
- **Components of JRE:**
- **JVM (Java Virtual Machine)**
- Executes Java bytecode and ensures platform independence.
- Handles memory management and garbage collection.
- **Core Libraries**
- Pre-built Java classes that help with tasks like data structures, file handling, networking, etc.
- **Runtime Libraries & Supporting Files**
- Includes libraries for graphics, security, and input/output operations.



Feature	JRE	JDK	JVM
Runs Java programs	✅	✅	✅
Includes compiler	❌	✅	❌
Includes libraries	✅	✅	❌
Used for development	❌	✅	❌

Compiling and Executing a Basic Java Program

Step 1: Install Java (JDK)

Download and install the Java Development Kit (JDK) from Oracle or use OpenJDK.

Step 2: Write a Basic Java Program

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, Java!"); // Output message
   }
}
```

Step 3: Compile the Java Program

- Open Command Prompt (Windows) or Terminal (Mac/Linux).

- Navigate to the folder where your .java file is saved.
- Run the following command to compile
- **javac HelloWorld.java**
- This generates a file named HelloWorld.class (bytecode).
- OUTPU: Hello, Java!

Step-by-Step Explanation

- **class HelloWorld** → Defines a class named HelloWorld.
- **public** → This means the class is accessible from anywhere.
- The filename (HelloWorld.java) must match the class name.
- **public static void main(String[] args)**
- **public** → Allows the method to be accessed from outside the class.
- **static** → No need to create an object to call this method.
- **void** → Means this method does not return any value.
- **main** → The name of the method that JVM calls to start execution.
- **String[] args** → Allows command-line arguments (optional input from the user).

Program Execution Flow

- Java Compiler (javac HelloWorld.java) → Converts the source code into bytecode (HelloWorld.class).
- Java Virtual Machine (JVM) (java HelloWorld) → Executes the bytecode.

Java IDEs (Integrated Development Environments)

- An IDE (Integrated Development Environment) is a software application that provides tools to help programmers write, test, and debug code more efficiently
- Key Features of an IDE:
- Code Editor – Helps write and edit code with features like syntax highlighting and auto-completion.
- Compiler/Interpreter – Translates code into machine language so it can run on a computer.
- Debugger – Helps find and fix errors in the code.
- **Build Automation** – Manages tasks like compiling and packaging code.
- **Version Control Integration** – Works with tools like Git to track changes in the code.

- **Why Use an IDE?**
- Makes coding **faster and easier** with intelligent suggestions.
- Reduces errors with **debugging tools**.
- Helps manage **large projects** efficiently.

1 Eclipse

- **Eclipse** is a popular open-source IDE mainly used for Java development. It is powerful, customizable, and supports large-scale projects
- Free & Open-Source – No cost, widely used in the industry
- Extensive Plugin Support – Can be extended for web, mobile, and enterprise development.
- Built-in Debugger – Helps find and fix errors efficiently.
- Code Auto-Completion – Speeds up coding with intelligent suggestions.
- Version Control Integration – Supports Git, SVN etc.
- Cross-Platform – Works on Windows, macOS, and Linux.
- Supports Multiple Languages – Java, C++, Python, PHP, and more.

2 NetBeans

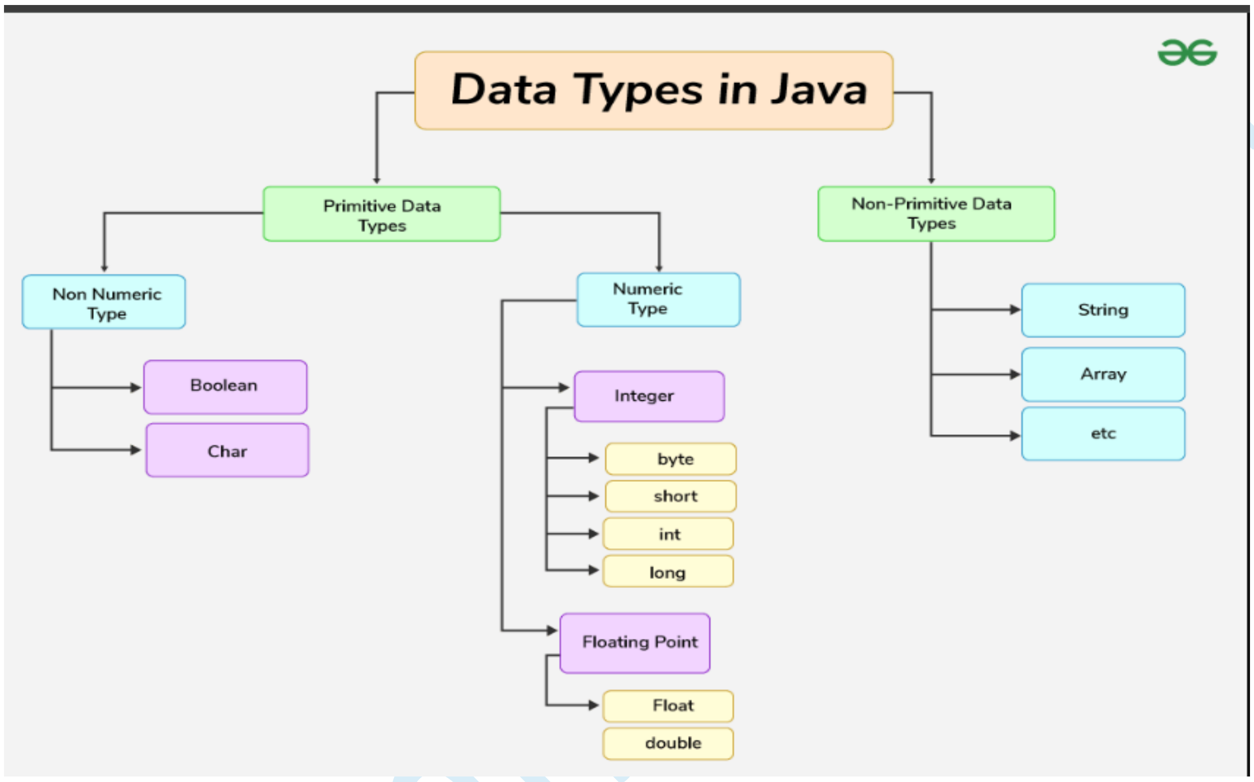
NetBeans

- NetBeans is a free, open-source Integrated Development Environment (IDE) used for developing Java applications. It is maintained by Apache and supports multiple programming languages like Java, C++, PHP, and HTML
- Developing Java SE & Java EE applications.
- Creating Desktop Applications using Java Swing.
- Developing Web Applications with Java, HTML, and JavaScript.
- Writing and debugging C/C++ and PHP programs.
- **NetBeans** is a free, open-source **IDE** used for Java development. It is beginner-friendly and provides excellent tools for coding, debugging, and managing projects.
- **Free & Open-Source** – Maintained by Apache.
- **Easy to Use** – Simple interface, great for beginners.
- **Supports Multiple Languages** – Java, C/C++, PHP, HTML, JavaScript, and more.
- **Built-in GUI Builder** – Drag-and-drop tools for Java Swing applications.
- **Smart Code Completion** – Speeds up development.

Java Data Types

- Java is statically typed and also a strongly typed language because, in Java, each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the Java data types.
- Data types in Java are of different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases. Java has two categories in which data types are segregated

- **Primitive Data Type**: such as boolean, char, int, short, byte, long, float, and double. The Boolean with uppercase B is a wrapper class for the primitive data type boolean in Java.
- **Non-Primitive Data Type or Object Data type**: such as String, Array, etc.



1 int Data Type

- It is a 32-bit signed two's complement integer.
- Syntax=> **int intVar; (Example: int number=30;)**

2 Float

- The float data type is a single-precision 32-bit IEEE 754 floating-point. Use a float (instead of double) if you need to save memory in large arrays of floating-point numbers. The size of the float data type is 4 bytes (32 bits).
- Syntax=> **float floatVar; (Example: float floatnumber=30.3f;)**

3 char Data Type

- The char data type is a single 16-bit Unicode character with the size of 2 bytes (16 bits).
- Syntax=> **char charVar; (Example: char charvar='a';)**

4 Boolean Data Type

- The boolean data type represents a logical value that can be either true or false. Conceptually, it represents a single bit of information, but the actual size used by the virtual machine is implementation-dependent and typically at least one byte (eight bits) in practice. Values of the boolean type are not implicitly or explicitly converted to any other type using casts. However, programmers can write conversion code if needed.
- Syntax=> **Boolean booleanVar; (Example:Boolean booleanVAR=true;)**

  Example:
```
class GFG {
  public static void main(String args[]) {
    char a = 'G';
     int i = 89;
     byte b = 4;
     short s = 56;
     double d = 4.355453532;
     float f = 4.7333434f;
     long l = 12121; System.out.println("char: " + a);
     System.out.println("integer: " + i);
     System.out.println("byte: " + b);
     System.out.println("short: " + s);
     System.out.println("float: " + f);
     System.out.println("double: " + d);
     System.out.println("long: " + l);
}
}
```

Java Tokens

- In Java, tokens are the smallest elements of a program that are meaningful to the compiler. They are the basic building blocks of Java code. Java has the following types of tokens:

1 Keywords

- Keywords are reserved words in Java that have predefined meanings and cannot be used as variable, class, or method names. They help define the syntax and structure of Java programs.

(1) Categories of Keywords in Java

- **Data Type Keywords**
- byte - 8-bit integer
- short - 16-bit integer
- int - 32-bit integer
- long - 64-bit integer
- float - 32-bit floating point
- double - 64-bit floating point
- char - 16-bit Unicode character
- boolean - Represents true or false

(2)Control Flow Keywords

- if - Conditional statement
- else - Alternative block in conditional statement
- switch - Multi-condition selection
- case - Defines a case inside a switch
- default - Default case in switch
- while - Looping statement
- do - Used with do-while loop
- for - Looping statement
- break - Exits a loop or switch statement
- continue - Skips the current iteration in a loop
- return - Exits from a method and optionally returns a value

(3)Access Modifiers

- public - Accessible from anywhere
- private - Accessible only within the class
- protected - Accessible within the same package or subclasses
- default (no keyword) - Accessible only within the same package

(4) Class-Related Keywords

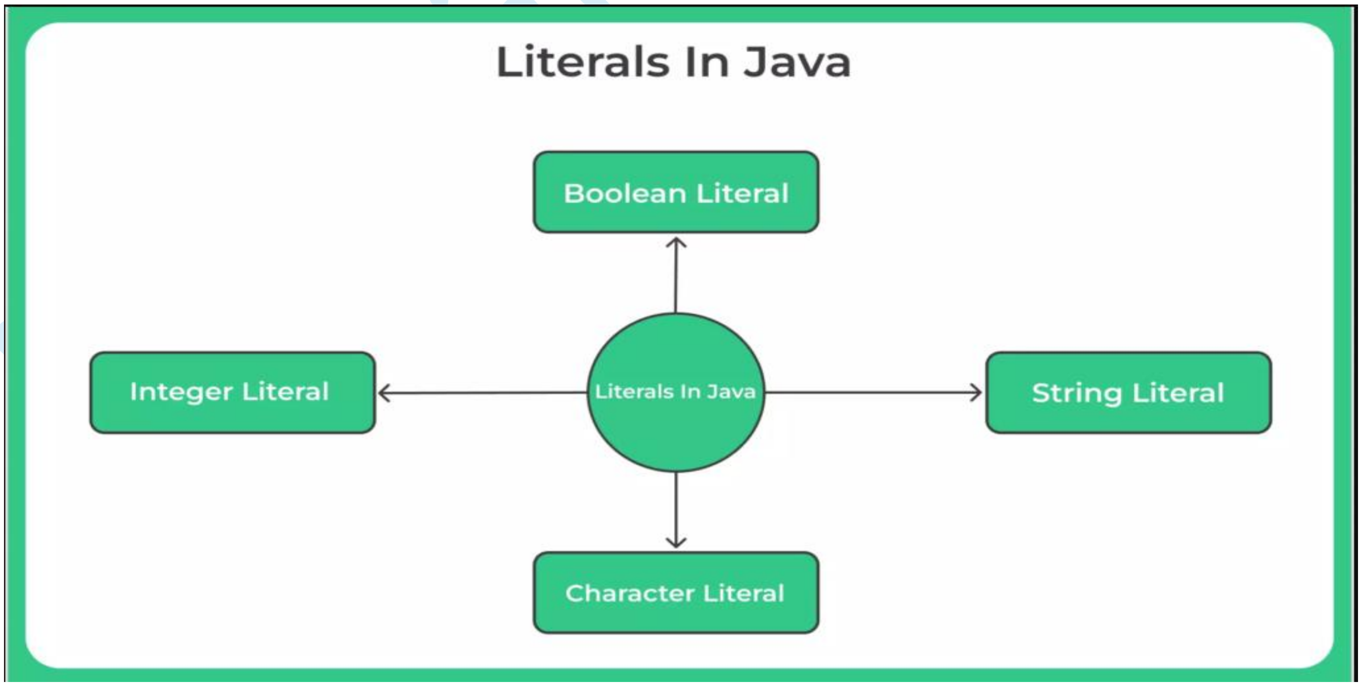
- Used to create and manage objects.
- new - Creates a new object
- this - Refers to the current object
- super - Refers to the parent class

(5) Object Keywords

- new - Creates a new object
- this - Refers to the current object
- super - Refers to the parent class

Literal In Java

- A literal in Java is a fixed value that is assigned to a variable in a program. Literals are the fundamental building blocks of Java programs.
- In simple words, Literals in Java is a representation of Boolean, numeric, character, or string data.
- Readability: Literals make code easier to understand and maintain.
- Efficiency: Literals are evaluated during compilation, which reduces runtime overhead.
- Consistency: Literals ensure uniformity across a program.



- Integer literals: Represent whole numbers, such as 42, -10, or 0

- Character literals: Represent single characters, such as 'A' or 'a'
- String literals: Represent sequences of characters, such as "Hello, World!" or "12345"
- Boolean literals: Represent the values TRUE or FALSE

Syntax:

- **int decimal = 100;   // Decimal (base 10)**
- **int binary = 0b1010; // Binary (base 2)**
- **int octal = 0123;   // Octal (base 8)**
- **int hex = 0x1A;   // Hexadecimal (base 16)**
- **long bigNumber = 123456789L; // Long literal**
- **char letter = 'A';**
- **char unicodeChar = '\u0041'; // Unicode for 'A'’**
- **String message = "Hello, Java!";**

- **String literals**: Represent sequences of characters enclosed in double quotes. For example, ‘Literals String’.
- **Numeric literals**: Represent numbers, which can be exact or approximate.
- **Exact numeric literals**: Whole numbers without a decimal point, such as 65.
- **Approximate numeric literals**: Numbers with decimal points or exponential components, such as 5.7
- **Boolean literals**: Represent the boolean values true or false
- Character literals: Represent single characters enclosed in single quotes.
- **Null literals**: Represent the absence of a value.
- **For Integral data types** (byte, short, int, long), we can specify literals in 4 ways:-
- **Decimal literals** (Base 10): In this form, the allowed digits are 0-9.
- int x = 101;

Octal literals (Base 8): In this form, the allowed digits are 0-7.\

- The octal number should be prefix with 0.
- int x = 0146;
- **Hexa-decimal literals (Base 16):** In this form, the allowed digits are 0-9, and characters are a-f. We can use both uppercase and lowercase

characters as we know that java is a case-sensitive programming language, but here java is not case-sensitive

- **int x = 0X123F;(with 0X or 0x)**
- **Binary literals**: we can specify literal value even in binary form also, allowed digits are 0 and 1. Literals value should be prefixed with 0b or 0B.
- **int x = 0b1111;**
- **float myFloatNum = 5.99f;**
- **char myLetter = 'D';**
- **boolean myBool = true;**
- **String myText = "Hello"**

Example:
```
public class LiteralDemo {
   public static void main(String[] args) {
      // Integer literals
      int decimal = 10;      // Decimal
      int binary = 0b1010;   // Binary (Prefix: 0b)
      int octal = 012;       // Octal (Prefix: 0)
      int hex = 0xA;         // Hexadecimal (Prefix: 0x)

      // Floating-point literals
      float floatNum = 3.14F; // 'F' suffix for float
      double doubleNum = 2.5e3; // Exponential notation (2.5 × 10^3)

      // Character and String literals
      char charLiteral = 'A';
      String stringLiteral = "Hello, Java!";

      // Boolean literals
      boolean boolTrue = true;
      boolean boolFalse = false;

      // Null literal
      String nullString = null;

      // Printing the literals
      System.out.println("Integer Literals: " + decimal + ", " + binary +
", " + octal + ", " + hex);
```

```
        System.out.println("Floating-point Literals: " + floatNum + ", " +
doubleNum);
        System.out.println("Character Literal: " + charLiteral);
        System.out.println("String Literal: " + stringLiteral);
        System.out.println("Boolean Literals: " + boolTrue + ", " +
boolFalse);
        System.out.println("Null Literal: " + nullString);
    }
}
```

Identifier

- In Java, an identifier is a name given to a programming element, such as a variable, class, method, or object. Identifiers are used to make code more readable and maintainable.
- Names can contain letters, digits, underscores, and dollar signs
- Names must begin with a letter
- Names should start with a lowercase letter, and cannot contain whitespace
- Names can also begin with $ and _
- Names are case-sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like Java keywords, such as int or boolean) cannot be used as names

Whitespace & separators

- Whitespace, including spaces, tabs, and newlines, is used to separate identifiers, keywords, operators, and other tokens in Java code, making it readable for the compiler.
- Java operators are special symbols that perform operations on variables or values. They can be classified into several categories based on their functionality. These operators play a crucial role in performing arithmetic, logical, relational, and bitwise operations etc

- **Arithmetic Operators**
- **Unary Operators**
- **Assignment Operator**
- **Relational Operators**
- **Logical Operators**
- **Ternary Operator**

- **Bitwise Operators**
- **Shift Operators**
- **instance of operator**

Arithmetic Operators

- Arithmetic Operators are used to perform simple arithmetic operations on primitive and non-primitive data types
- **: Multiplication**
- **/ : Division**
- **% : Modulo**
- **+ : Addition**
- **: Subtraction**

Example :

```
public class ArithmeticDemo {
    public static void main(String[] args) {
        int a = 10, b = 5;
 // Addition
 System.out.println("Addition: " + (a + b));

        // Subtraction
        System.out.println("Subtraction: " + (a - b));

        // Multiplication
        System.out.println("Multiplication: " + (a * b));

        // Division
        System.out.println("Division: " + (a / b));  // Integer division

        // Modulus (Remainder)
        System.out.println("Modulus: " + (a % b));

        // Floating-point division
        double x = 10, y = 3;
        System.out.println("Floating-point Division: " + (x / y));
    }
}
```

Unary Operators

- Unary Operators need only one operand. They are used to increment, decrement, or negate a value.
- ++ , Increments by 1.
- Post-Increment: Uses value first, then increments.
- Pre-Increment: Increments first, then uses value.
  --, Decrements by 1.
- Post-Decrement: Uses value first, then decrements.
- Pre-Decrement: Decrements first, then uses value.
- ! , Inverts a Boolean value.

Example:

```
public class UnaryOperatorsDemo {

    public static void main(String[] args) {

        int a = 5;

        boolean flag = true;

        // Unary Plus & Minus

        System.out.println("Unary Plus: " + (+a));

        System.out.println("Unary Minus: " + (-a));

        // Pre-Increment (++ before variable)
        System.out.println("Pre-Increment: " + (++a)); // a
becomes 6

        // Post-Increment (variable before ++)

        System.out.println("Post-Increment: " + (a++)); //
Prints 6, then a becomes 7

        // Pre-Decrement (-- before variable)

        System.out.println("Pre-Decrement: " + (--a)); // a
becomes 6

        // Post-Decrement (variable before --)

        System.out.println("Post-Decrement: " + (a--)); //
Prints 6, then a becomes 5



      // Logical NOT operator

        System.out.println("Logical NOT (!flag): " + (!flag));

    }

}
```

- Assignment Operator
- '=' Assignment operator is used to assign a value to any variable. It has right-to-left associativity, i.e. value given on the right-hand side of the operator is assigned to the variable on the left, and therefore right-hand side value must be declared before using it or should be a constant.
- The general format of the assignment operator is:
- variable = value;
- In many cases, the assignment operator can be combined with others to create shorthand compound statements. For example, a += 5 replaces a = a + 5. Common compound operators include:

- += , Add and assign.
- -= , Subtract and assign.
- *= , Multiply and assign.
- /= , Divide and assign.
- %= , Modulo and assign.

Example:

```
public class AssignmentOperatorsDemo {
   public static void main(String[] args) {
      int a = 10, b = 5;

      // Simple Assignment
      System.out.println("Initial Value: a = " + a);

      // Addition Assignment
      a += b;
      System.out.println("After a += b: " + a); // a = 10 + 5 = 15

      // Subtraction Assignment
      a -= b;
      System.out.println("After a -= b: " + a); // a = 15 - 5 = 10

      // Multiplication Assignment
      a *= b;
      System.out.println("After a *= b: " + a); // a = 10 * 5 = 50

      // Division Assignment
      a /= b;
      System.out.println("After a /= b: " + a); // a = 50 / 5 = 10

      // Modulus Assignment
      a %= b;
      System.out.println("After a %= b: " + a); // a = 10 % 5 = 0
   }
}
```

Relational operators

- Relational operators in Java are symbols that compare two values and return a Boolean value. They are used to determine if two values are equal, greater than, less than, and so on.
- Java Relational Operators are a bunch of binary operators used to check for relations between two operands, including equality, greater than, less than, etc. They return a boolean result after the comparison and are extensively used in looping statements as well as conditional if-else statements and so on. The general format of representing relational operator is:
- variable1 relation operator variable2
- **Equal to' operator (==)**
- This operator is used to check whether the two given operands are equal or not. The operator returns true if the operand at the left-hand side is equal to the right-hand side, else false.
- **'Not equal to' Operator(!=)**
- This operator is used to check whether the two given operands are equal or not. It functions opposite to that of the equal-to-operator. It returns true if the operand at the left-hand side is not equal to the right-hand side, else false.
- **'Greater than' operator(>)**
- This checks whether the first operand is greater than the second operand or not. The operator returns true when the operand at the left-hand side is greater than the right-hand side.
- **'Less than' Operator(<)**
- This checks whether the first operand is less than the second operand or not. The operator returns true when the operand at the left-hand side is less than the right-hand side. It functions opposite to that of the greater-than operator.

Example:

```
public class RelationalOperatorsDemo {
   public static void main(String[] args) {
      int a = 10, b = 5;

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

      // Equal to
      System.out.println("a == b : " + (a == b));

      // Not Equal to
      System.out.println("a != b : " + (a != b));
```

```
        // Greater than
        System.out.println("a > b  : " + (a > b));

        // Less than
        System.out.println("a < b  : " + (a < b));

        // Greater than or equal to
        System.out.println("a >= b : " + (a >= b));

        // Less than or equal to
        System.out.println("a <= b : " + (a <= b));
    }
}
```

Logical operators

- Logical operators are used to perform logical "AND", "OR" and "NOT" operations, i.e. the function similar to AND gate and OR gate in digital electronics
- They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition under particular consideration. One thing to keep in mind is, while using AND operator, the second condition is not evaluated if the first one is false. Whereas while using OR operator, the second condition is not evaluated if the first one is true, i.e. the AND and OR operators have a short-circuiting effect. Used extensively to test for several conditions for making a decision.
- AND Operator (&&) – if(a && b) [if true execute else don't]
- OR Operator (||) – if(a || b) [if one of them is true to execute else don't]
- NOT Operator (!) – !(a<b) [returns false if a is smaller than b]

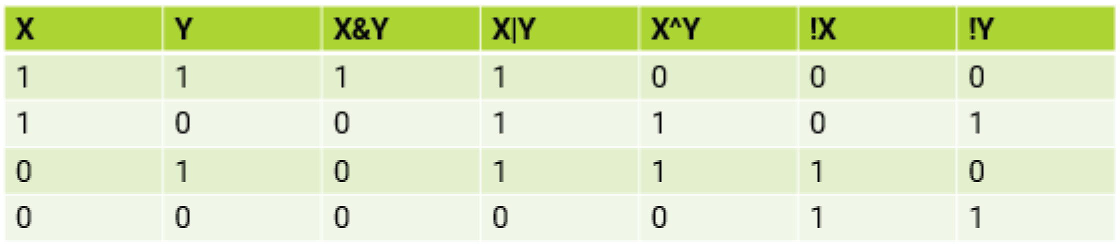
Example:

```
public class LogicalOperatorsDemo {
    public static void main(String[] args) {
        boolean x = true, y = false;
        System.out.println("x = " + x + ", y = " + y);
        // Logical AND
        System.out.println("x && y : " + (x && y)); // false (both
must be true)
        // Logical OR
        System.out.println("x || y : " + (x || y)); // true (at least
one is true)
        // Logical NOT
        System.out.println("!x : " + (!x)); // false (negation of true)
        System.out.println("!y : " + (!y)); // true (negation of false)
}
}
```

Bitwise operators

- Bitwise operators in Java are used to perform operations at the bit level. These operators work directly on binary representations of integers (or compatible types like char, byte, and short) and perform operations bit by bit.
- Bitwise AND,OR,XOR,and NOT use same as the logic operators

operator	description
&	bitwise AND operator (&)
|	The bitwise OR operator
^	Bitwise exclusive OR (XOR)
~	Bitwise Complement (unary not)
<<	bitwise shift left
>>	bitwise shift right



X	Y	X&Y	X|Y	X^Y	!X	!Y
1	1	1	1	0	0	0
1	0	0	1	1	0	1
0	1	0	1	1	1	0
0	0	0	0	0	1	1

- Bitwise exclusive OR (XOR) is a binary operation that operates on corresponding bits of two numbers. The result of XOR for each bit is 1 if the bits are different, and 0 if the bits are the same.
- int a = 5; // Binary: 0101 int b = 3; // Binary: 0011
- int result = a ^ b; // Binary: 0110
- The bitwise complement operator in Java is represented by the tilde (~). It is a unary operator that inverts each bit of its operand. A 0 becomes 1, and a 1 becomes 0.
- int number = 5; // Binary: 00000000 00000000 00000000 00000101
- int complement = ~number; // Binary: 11111111 11111111 11111111 11111010
- The bitwise shift left operator in Java is represented by <<. It shifts the bits of a number to the left by a specified number of positions, filling the rightmost bits with zeros.
- int number = 5; // Binary: 00000000 00000000 00000000 00000101 int result = number << 2; // Shift left by 2 positions
- 00000000 00000000 00000000 00000101=>5
- 00000000 00000000 00000000 00010100=>shift left 2
- Shifts the bits of the number to the right, filling the leftmost bits with the sign bit (preserving the number's sign).
- Positive numbers: Leftmost bits are filled with 0.
- Negative numbers: Leftmost bits are filled with 1 to maintain the sign.
- int number = 16; // Binary: 00000000 00000000 00000000 00010000 int result = number >> 2; // Shift right by 2 positions
- Binary representation of 16: 00000000 00000000 00000000 00010000.
- Shift right by 2: 00000000 00000000 00000000 00000100

```
Example:
public class BitwiseOperatorsDemo {
  public static void main(String[] args) {
    int a = 5, b = 3; // 5 -> 0101, 3 -> 0011

    System.out.println("a = " + a + ", b = " + b);
    // Bitwise AND
```

```
        System.out.println("a & b  :" + (a & b));  // 0001 -> 1
        // Bitwise OR
        System.out.println("a | b  :" + (a | b));  // 0111 -> 7
        // Bitwise XOR
        System.out.println("a ^ b  :" + (a ^ b));  // 0110 -> 6
        // Bitwise Complement
        System.out.println("~a     :" + (~a));    // 1010 -> -6
        // Left Shift
        System.out.println("a << 1 :" + (a << 1)); // 1010 -> 10
        // Right Shift
        System.out.println("a >> 1 :" + (a >> 1)); // 0010 -> 2
        // Unsigned Right Shift
        System.out.println("a >>> 1:" + (a >>> 1)); // 0010 -> 2
    }
}
Output:
a = 5, b = 3
a & b  : 1
a | b  : 7
a ^ b  : 6
~a     : -6
a << 1 : 10
a >> 1 : 2
a >>> 1: 2
```

Java keywords(Assertion)

- An assertion allows testing the correctness of any assumptions that have been made in the program.
- An assertion is achieved using the assert statement in Java
- While executing assertion, it is believed to be true. If it fails, JVM throws an error named Assertion Error. It is mainly used for testing purposes during development.
- The assert statement is used with a Boolean expression and can be written in two different ways.
  Why use Assertions ?

- Wherever a programmer wants to see if his/her assumptions are wrong or not.
- To make sure that an unreachable-looking code is actually unreachable.
- To make sure that assumptions written in comments are right.
- To make sure the default switch case is not reached.

Syntax of Assertions in Java

```
assert condition;        // Simple assertion
assert condition : message; // Assertion with error message
```

Example

```
public class AssertionsDemo {
   public static void main(String[] args) {
      int age = 17; // Change to test different cases

      // Assertion to check if age is valid for voting
      assert age >= 18 : "Age must be 18 or above to vote";

      System.out.println("You are eligible to vote.");
   }
}
```

Command for run programe:

```
java -ea AssertionsDemo
```

Output:

(1) If age = 18 or higher:

```
You are eligible to vote.
```

(2) If age < 18 (Assertion fails):

```
Exception in thread "main" java.lang.AssertionError:
Age must be 18 or above to vote
```

- **assert expression;**
- **java –ea =>enable assertion**
- **java –da Test =>disable assertion**

Java keywords(Strictfp)

- The strictfp keyword in Java was used to limit floating-point calculations to the IEEE 754 standard. This ensured that the results of floating-point calculations were consistent across different platforms. However, the strictfp keyword was removed in Java version 17, and is now considered obsolete and redundant

- The value of the floating point calculations may be very with different operating system.

Example Without strictfp:

```
Public static void main(String[] args)
{
System.out.println(10.00/3)
}
OUTPUT:
3.33335 FOR LINUX
3.334 FOR WINDOWS
3.3333 FOR MAC
```

Strictfp With

- **Class**
- **concrete method**
- **Interface**
- **Abstract class**

Strictfp Without

Variable, constructor ,abstract method

Example With strictfp:

```
strictfp class StrictfpExample {
 public static void main(String[] args) {
       double num1 = 10.0 / 3.0;
       double num2 = num1 * 3.0;
       System.out.println("Value of num1: " + num1);
       System.out.println("num1 * 3.0 = " + num2);
    }
}
```

Explanation

- The strictfp keyword was introduced in Java version 1.2.
- It was used as a modifier for classes, interfaces, and methods.
- The strictfp keyword ensured that floating-point calculations produced the same results on every platform.
- The strictfp keyword could not be used with abstract methods.
- The strictfp keyword was removed in Java version 17 because IEEE 754 semantics are now required by default.

Enum in Java

- Enum in java stand for enumeration.it is a data type that comes with set of pre-define constant values these values are separated by a comma.
- Enumeration or Enum in java is a special kind of data type defined by the user.
- An enum type is a special data type that enables for a variable to be a set of predefined constants.
- In Java Enum serve the purpose of representing a group of named constants in a programming language. Java Enums are used when we know all possible values at compile time, such as choices on a menu, rounding modes, command-line flags, etc.
- A Java enumeration is a class type. we don't need to instantiate an enum using new, it has the same capabilities as other classes
- This fact makes Java enumeration a very powerful tool. Just like classes, you can give them constructors, add instance variables and methods, and even implement interfaces.
- Enum declaration can be done outside a class or inside a class but not inside a method.
- Declaration outside the class:

Syntax:

```
enum Color {
      RED,
      GREEN,
      BLUE;
   }
```

Defines an enum Color with three fixed values: RED, GREEN, and BLUE.
Example:

```
// Define an enum for colors
enum Color {
   RED, GREEN, BLUE, YELLOW;
}

public class EnumExample {
   public static void main(String[] args) {
      // Assigning enum values
      Color color1 = Color.RED;
      Color color2 = Color.GREEN;
      Color color3 = Color.BLUE;
      Color color4 = Color.YELLOW;
```

```
        // Printing enum values
        System.out.println("Selected Colors:");
        System.out.println(color1);
        System.out.println(color2);
        System.out.println(color3);
        System.out.println(color4);
    }
}
```

Properties of Enum in Java=>

- Class Type: Every enum is internally implemented using the Class type.
- Enum Constants: Each enum constant represents an object of type enum.
- Switch Statements: Enum types can be used in switch statements.

Java Type Casting

- Type casting is when you assign a value of one primitive data type to another type.
- In Java, there are two types of casting:
- Widening Casting (automatically) - converting a smaller type to a larger type size
- byte -> short -> char -> int -> long -> float -> double
- Narrowing Casting (manually) - converting a larger type to a smaller size type
- double -> float -> long -> int -> char -> short -> byte
- In Java, type casting is a method or process that converts a data type into another data type in both ways manually and automaticall
- When you assign a value from one primitive data type to another type, this is known as type casting

  Syntax
- **<datatype> variableName = (<datatype>) value;**

Example:(Automatic)

```
public class Main {
 public static void main(String[] args) {
   int myInt = 9;
  double myDouble = myInt; // Automatic casting: int to double
    System.out.println(myInt);      // Outputs 9
    System.out.println(myDouble);   // Outputs 9.0
  }
}
```

Example:(Manual Automatic)

```
public static void main(String[] args)
  {
double i = 100.245;
// Narrowing Type Casting
        short j = (short)i;
  int k = (int)i;
System.out.println("Original Value before Casting" + i);
 System.out.println("After Type Casting to short "+ j);
     System.out.println("After Type Casting to int "+ k);
  }
```

Get User value

- In Java, you can get user input using the Scanner class, BufferedReader, or Console class. The most common way is using Scanner
- String name = scanner.nextLine();
- int age = scanner.nextInt();
- double salary = scanner.nextDouble();
- char ch = scanner.next().charAt(0);
- import java.util.Scanner;

Method	Description
`nextBoolean()`	Reads a `boolean` value from the user
`nextByte()`	Reads a `byte` value from the user
`nextDouble()`	Reads a `double` value from the user
`nextFloat()`	Reads a `float` value from the user
`nextInt()`	Reads a `int` value from the user
`nextLine()`	Reads a `String` value from the user
`nextLong()`	Reads a `long` value from the user
`nextShort()`	Reads a `short` value from the user

Example:

```
import java.util.Scanner;

public class ScannerExample {
  public static void main(String[] args) {
    // Create a Scanner object to read input
    Scanner scanner = new Scanner(System.in);
    // Read a string
    System.out.print("Enter your name: ");
    String name = scanner.nextLine();
    // Read an integer
    System.out.print("Enter your age: ");
    int age = scanner.nextInt();

    // Read a double
 System.out.print("Enter your height (in cm): ");
    double height = scanner.nextDouble();

    // Display the input values
    System.out.println("\nHello, " + name + "!");
    System.out.println("You are " + age + " years old and " + height + " cm
tall.");

    // Close the scanner
    scanner.close();
  }
}
```

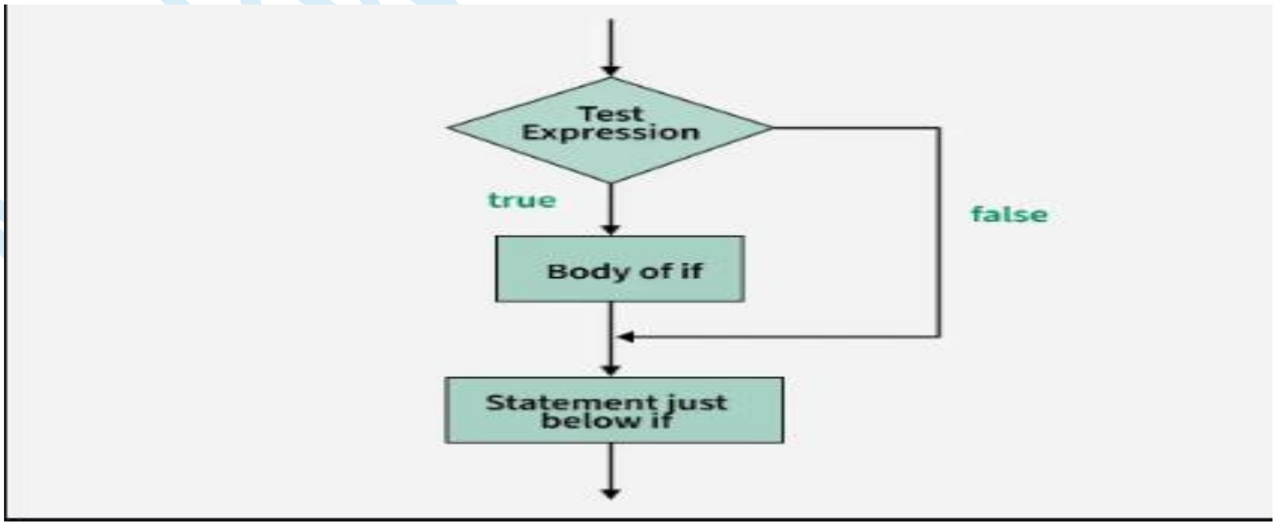
Decision-making statements

- Decision-making statements in Java execute a block of code based on a condition. Decision-making in programming is similar to decision-making in real life. In programming, we also face situations where we want a certain block of code to be executed when some condition is fulfilled.
- A programming language uses control statements to control the flow of execution of a program based on certain conditions. These are used to cause the flow of execution to advance and branch based on changes to the state of a program.

- **if**
- **if-else**
- **switch-case**
- **jump – break, continue, return**

➢ IF statement

- The if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e. if a certain condition is true then a block of statements is executed otherwise not.



Syntax:
if(condition) {

```
                // Statements to execute if
                condition is true
                }
```

Example:

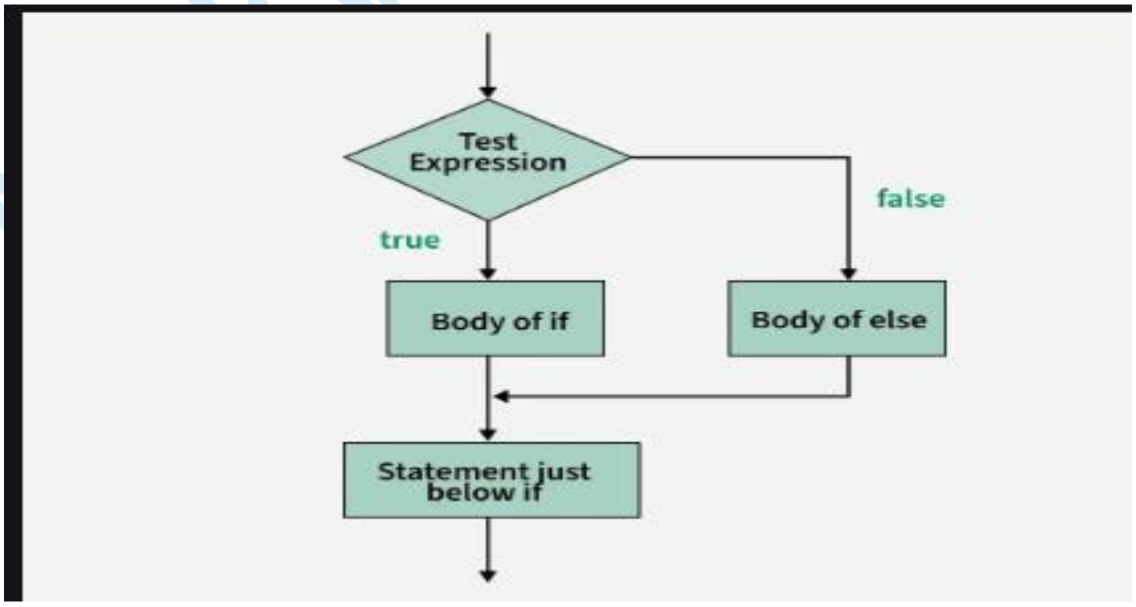
```
public class IfExample {
   public static void main(String[] args) {
      int number = 50;

      // Check if the number is positive, negative, or zero
      if (number > 0) {
         System.out.println("The number is positive.");
      } else if (number < 0) {
         System.out.println("The number is negative.");
      } else {
         System.out.println("The number is zero.");
      }

      // Close scanner
      scanner.close();
   }
}
```

➢ If else

- The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false? Here, comes the "else" statement. We can use the else statement with the if statement to execute a block of code when the condition is false.



Syntax:

```
if(condition){
// Executes this block if
// condition is true
}else{
// Executes this block if
// condition is false
}
```

Example:

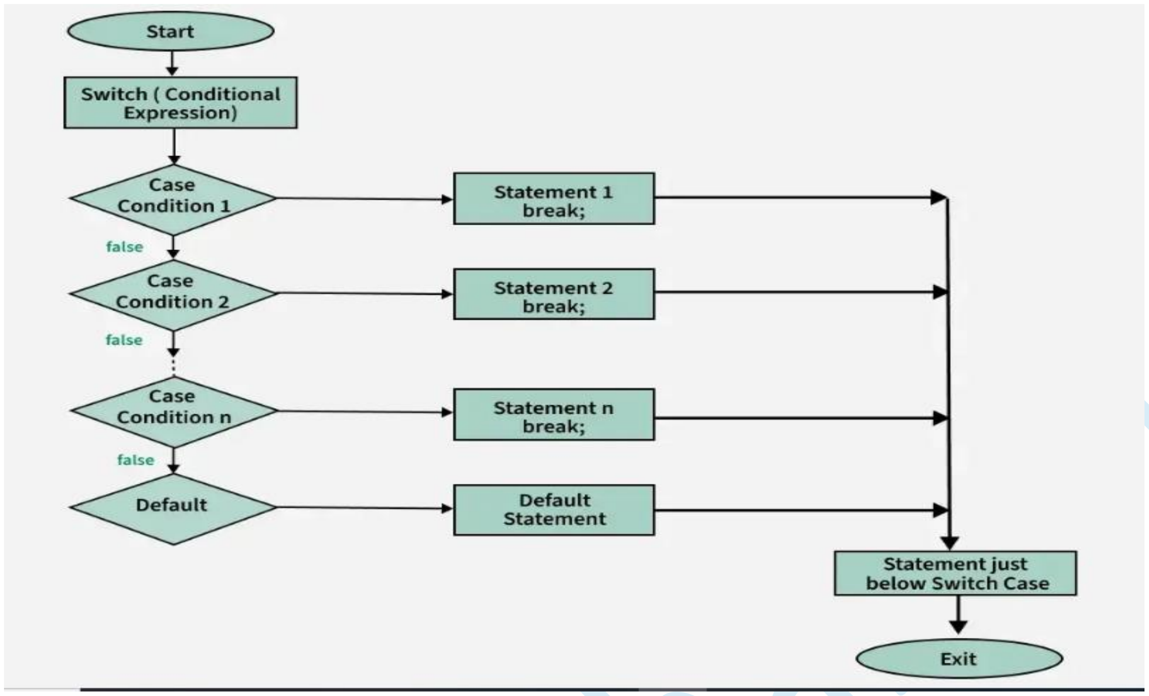
```
public class IfElseExample {
   public static void main(String[] args) {
      int number = 50;

      // Check if the number is even or odd
      if (number % 2 == 0) {
         System.out.println(number + " is even.");
      } else {
         System.out.println(number + " is odd.");
      }

      // Close scanner
      scanner.close();
   }}
```

- **Java Switch Case**

- The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- In Java, a switch statement does not support floating-point numbers (float or double) as case values. This is because floating-point arithmetic can introduce precision errors, making comparisons unreliable



Syntax:

```
switch (expression) {
case value1:
// code to be executed if expression == value1
break;
case value2:
 // code to be executed if expression == value2
 break;
// more cases...
default:
// code to be executed if no cases match
}
```

Example:

```
public class SwitchCaseExample {
   public static void main(String[] args) {
      int day = 20;
      // Switch case to determine the day
      switch (day) {
         case 1:
            System.out.println("Sunday");
            break;
         case 2:
            System.out.println("Monday");
            break;
         case 3:
            System.out.println("Tuesday");
            break;
         case 4:
            System.out.println("Wednesday");
            break;
         case 5:
            System.out.println("Thursday");
            break;
         case 6:
            System.out.println("Friday");
            break;
         case 7:
            System.out.println("Saturday");
            break;
         default:
            System.out.println("Invalid number! Please
enter a number between 1 and 7.");
      }

      // Close scanner
      scanner.close();
   }
}
```

Looping Statement(for, while, do....while)

- Looping statements are control flow structures in programming languages that repeat a set of instructions or code block.
- Types of looping statements:
- For loop: A repetition control structure that executes a loop a specific number of times.
- While loop: A control flow statement that executes code repeatedly based on a Boolean condition.

For Loop in Java

- A for loop in Java is used to execute a block of code multiple times. It has three components:
- Initialization – Declares and initializes a loop variable.
- Condition – Checks if the loop should continue.
- Update – Modifies the loop variable after each iteration.

Syntax:

```
        for(initialization; condition; update)
 {
// Code to execute in the loop
 }
```

Example:

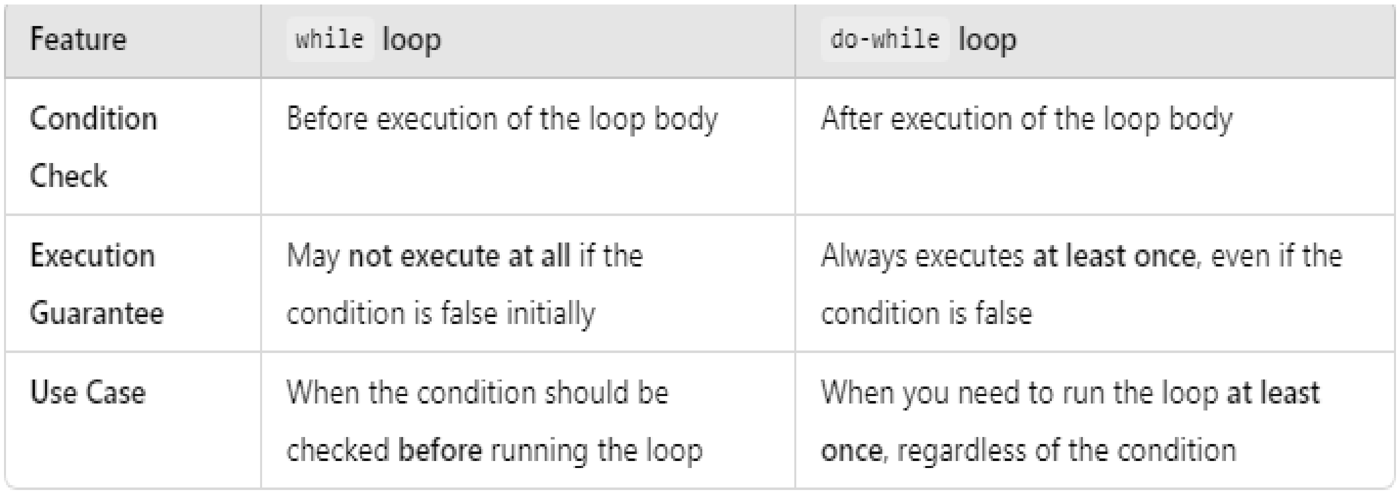
```
public static void main(String[] args) {
for(int i = 1; i <= 5; i++) {
System.out.println("Number: " + i);
}
}
```

while loop

- The condition is checked before executing the loop body.
- If the condition is false initially, the loop body won't execute even once.
- while (condition) { // Code to execute }
- int i = 1; while (i <= 5) { System.out.println("Count: " + i); i++; }

do-while loop

- The condition loop body executes at least once before checking the condition.
- on is checked after the first execution.
- do { // Code to execute} while (condition);
- int i = 1; do { System.out.println("Count: " + i); i++; } while (i <= 5);



Feature	`while` loop	`do-while` loop
Condition Check	Before execution of the loop body	After execution of the loop body
Execution Guarantee	May **not execute at all** if the condition is false initially	Always executes **at least once**, even if the condition is false
Use Case	When the condition should be checked **before** running the loop	When you need to run the loop **at least once**, regardless of the condition

Jumping statements

- Jumping statements are control statements that transfer execution control from one point to another point in the program. There are three Jump statements that are provided in the Java programming language
- **Break statement.**
- **Continue statement.**
- **Return Statement**

Break:

- In java, the break statement is used to terminate the execution of the nearest looping statement or switch statement. The break statement is widely used with the switch statement, for loop, while loop, do-while loop.

```
break;
int n = 10;
for (int i = 0; i < n; i++) {
        if (i == 6)
           break;
 System.out.println(i);

   }
```

- As you see, the code is meant to print 1 to 10 numbers using for loop, but it prints only 1 to 5 . as soon as i is equal to 6, the control terminates the loop.
- In a switch statement, if the break statement is missing, every case label is executed till the end of the switch.

Continue

- **In Java, the continue statement is used inside loops to skip the current iteration and move to the next iteration of the loop. It is typically used when you want to skip specific cases without breaking out of the loop entirely**

```
continue;
for (int i = 1; i <= 5; i++) {
            if (i == 3) {
            continue; // Skips iteration when i == 3
            }
            System.out.println("Number: " + i);
    }
```

Return

- The "return" keyword can help you transfer control from one method to the method that called it. Since the control jumps from one part of the program to another, the return is also a jump statement.
- "return" is a reserved keyword means we can't use it as an identifier.
- It is used to exit from a method, with or without a value.

Arrays

- Arrays are fundamental structures in Java that allow us to store multiple values of the same type in a single variable. They are useful for storing and managing collections of data. Arrays in Java are objects, which makes them work differently from arrays in C/C++ in terms of memory management
- Array Declaration
- type[] arrayName;
- type: The data type of the array elements (e.g., int, String).
- arrayName: The name of the array.

```
    int[] arr = { 1, 2, 3, 4, 5 };
     // size of array
      int n = arr.length;
    // traversing array
    for (int i = 0; i < n; i++)
  System.out.print(arr[i] + " ");
```

one-dimensional

- A one-dimensional array in Java is a collection of elements of the same data type stored in a contiguous memory location. It allows you to store multiple values in a single variable.
- Data_Type array_name [];
- Ex:int mark
- int[] numbers = {10, 20, 30, 40, 50}; // Direct initialization

Multidimensional

- A multidimensional array in Java is an array of arrays, allowing you to store data in a table-like structure. The most commonly used multidimensional array is a 2D array (matrix), but Java supports arrays of higher dimensions as well.
- datatype[][] arrayName = { {value1, value2}, {value3, value4} };
- int[][] numbers = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

Rectangular array

- A rectangular array in Java is a 2D array where all rows have the same number of columns.

```
datatype[][] arrayName = new datatype[rows][columns];
datatype[][] arrayName = {
{value1, value2, value3},
{value4, value5, value6},
{value7, value8, value9}
};
int[][] matrix = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

Rectangular arrays have a fixed number of columns in each row

Different from jagged arrays, where rows can have different lengths.

Jagged

- A jagged array (also called a ragged array) in Java is a multidimensional array where rows can have different numbers of columns. Unlike rectangular arrays, where each row has the same number of elements, jagged arrays allow flexibility in row size

- int[][] jaggedArray = new int[3][]; // Initializing rows with different column sizes jaggedArray[0] = new int[2]; // Row 0 has 2 columns jaggedArray[1] = new int[4]; // Row 1 has 4 columns jaggedArray[2] = new int[3]; // Row 2 has 3 columns

Example:

```
public class JaggedArrayDemo {
   public static void main(String[] args) {
      // Declare a jagged array
      int[][] jaggedArray = new int[3][];

      // Initialize rows with different lengths
      jaggedArray[0] = new int[3]; // First row has 3 elements
      jaggedArray[1] = new int[2]; // Second row has 2 elements
      jaggedArray[2] = new int[4]; // Third row has 4 elements

      // Assign values to the jagged array
      int value = 1;
      for (int i = 0; i < jaggedArray.length; i++) {
         for (int j = 0; j < jaggedArray[i].length; j++) {
            jaggedArray[i][j] = value++;
         }
      }

      // Print the jagged array
      System.out.println("Jagged Array:");
      for (int i = 0; i < jaggedArray.length; i++) {
         for (int j = 0; j < jaggedArray[i].length; j++) {
            System.out.print(jaggedArray[i][j] + " ");
         }
         System.out.println(); // Move to the next line
      }
   }
}
```

Command Line Arguments in Java

- Command line arguments allow you to pass values to a Java program when running it from the terminal or command prompt.
  Example:
```
public class CommandLineExample {
  public static void main(String[] args) {
System.out.println("Number of arguments: " +
args.length);
   // Loop through the arguments and print them
  for (int i = 0; i < args.length; i++) {
    System.out.println("Argument " + i + ": " +
args[i]);
  }
   }
}
```
- Command line arguments are passed as String[] args.
- You must convert arguments if using numbers (Integer.parseInt()).
- Always handle errors to prevent crashes.

OOP

- Object-Oriented Programming (OOP) is a programming that uses objects and classes to design and develop software. Java is a fully object-oriented language that follows OOP principles.
- Class: A blueprint/template for creating objects. It defines attributes (variables) and behaviors (methods).
- Object: An instance of a class with real values assigned to its attributes.

Encapsulation

- Encapsulation is restricting direct access to class members using private and providing access via getter and setter methods.
- Protects data from unwonted modifications.
- Improves code maintainability.

Key Features of Encapsulation

- **Data Hiding:**

- Fields (name and age) are private, so they can't be accessed directly from outside the class.
- **Controlled Access:**
- Public getter and setter methods (getName(), setName(), getAge(), setAge()) control how the data is accessed and modified.
- **Validation:**
- In the setAge() method, we added a condition to prevent negative values.

Advantages of Encapsulation

- Better Data Security (prevents direct modification)
- Improves Code Maintainability (easy to update logic without breaking other parts)
- Enhances Code Reusability (can be used in different applications)

Syntax of Encapsulation

```
// Step 1: Create an encapsulated class
class ClassName {
   // Private variables (Data Hiding)
   private DataType variableName;

   // Public getter method (to retrieve the value)
   public DataType getVariableName() {
      return variableName;
   }

   // Public setter method (to modify the value)
   public void setVariableName(DataType variableName) {
      this.variableName = variableName;
   }
}

// Step 2: Create a main class to access encapsulated data
public class MainClass {
   public static void main(String[] args) {
      // Create an object of the encapsulated class
      ClassName obj = new ClassName();

      // Set value using setter
```

```
            obj.setVariableName(value);

            // Get value using getter
            System.out.println(obj.getVariableName());
        }
    }
```

Example: Encapsulation in Java

```
// Encapsulated class
class Student {
    // Private variables (data hiding)
    private String name;
    private int age;

    // Getter method for name
    public String getName() {
        return name;
    }

    // Setter method for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter method for age
    public int getAge() {
        return age;
    }

    // Setter method for age (with validation)
    public void setAge(int age) {
        if (age > 0) {  // Ensure valid age
            this.age = age;
        } else {
            System.out.println("Age must be positive!");
        }
    }
}

// Main class
public class EncapsulationDemo {
    public static void main(String[] args) {
        // Create an object of Student class
        Student student = new Student();

        // Set values using setter methods
        student.setName("Alice");
        student.setAge(20);

        // Get values using getter methods
        System.out.println("Student Name: " + student.getName());
        System.out.println("Student Age: " + student.getAge());
    }
}
```

Inheritance (Reusability)

- Inheritance allows a child class (subclass) to inherit attributes and behaviors from a parent class (superclass).
- Promotes code reuse.
- Establishes a hierarchical relationship.
- Inheritance is one of the four fundamental principles of Object-Oriented Programming (OOP) in Java. It allows a child class (subclass) to inherit properties and behaviors (fields and methods) from a parent class (superclass).
- **Single Inheritance → One class inherits from another.**
- **Multilevel Inheritance → A class inherits from another child class.**
- **Hierarchical Inheritance → Multiple classes inherit from one parent.**

Key Features of Inheritance

- Code Reusability → Common code is written once in the parent class and reused in child classes.
- Method Overriding → Child classes can modify inherited methods.
- Hierarchy Representation → Represents real-world relationships (e.g., Vehicle → Car).

Example:

```
// Parent class (Superclass)
class ParentClass {
   // Parent class properties and methods
}
```

```
// Child class (Subclass) inheriting from ParentClass
class ChildClass extends ParentClass {
   // Additional properties and methods of the child class
}
```

Example: Inheritance in Java

```
// Parent class
class Animal {
   void makeSound() {
      System.out.println("Animals make sounds...");
   }
}

// Child class inheriting from Animal
class Dog extends Animal {
   void bark() {
      System.out.println("Dog barks: Woof Woof!");
   }
}

// Main class
public class InheritanceDemo {
   public static void main(String[] args) {
      Dog myDog = new Dog(); // Create an object of Dog
      myDog.makeSound();  // Inherited method from Animal class
      myDog.bark();      // Method of Dog class
   }}
```

Polymorphism

- Polymorphism allows the same method to have different implementations.
- Compile-time (Method Overloading) → Same method name, different parameters.
- Runtime (Method Overriding) → Same method in parent & child, different behavior\
- Compile-time (Method Overloading) → Same method name, different parameters.
- Runtime (Method Overriding) → Same method in parent & child, different behavior
- **Code Reusability** → Write a method once and reuse it for different data types or behaviors.

- **Method Overloading** (Compile-time Polymorphism) → Same method name with different parameters in the same class.
- **Method Overriding** (Runtime Polymorphism) → Same method name in parent and child classes with different behavior.
- **Increases Flexibility** → A single interface can be used for multiple types.
- **Supports Dynamic Method Dispatch** → Method calls are resolved at runtime.
- **Enhances Maintainability** → Reduces code duplication and improves readability.
- Works with Inheritance and Interfaces → Allows objects of different classes to be treated as the same type.

Method Overloading (Compile-time Polymorphism)

Syntax:

```
class ClassName {
   // Method with different parameters (Overloading)
   returnType methodName(Type1 param1) { }
   returnType methodName(Type1 param1, Type2 param2) { }
}

Example:
class MathOperations {
   // Method with two parameters
   int add(int a, int b) {
      return a + b;
   }

   // Overloaded method with three parameters
   int add(int a, int b, int c) {
      return a + b + c;
   }
}

public class OverloadingDemo {
   public static void main(String[] args) {
      MathOperations obj = new MathOperations();
      System.out.println("Sum (2 numbers): " + obj.add(5, 10));
      System.out.println("Sum (3 numbers): " + obj.add(5, 10, 15));
```

```
    }
}
```

Method Overriding (Runtime Polymorphism)

```
class ParentClass {
    void show() {
        // Parent method
    }
}

class ChildClass extends ParentClass {
    @Override
    void show() {
        // Overridden method
    }
}
```

Example:

```
// Parent class
class Animal {
    void makeSound() {
        System.out.println("Animals make different
sounds...");
    }
}

// Child class overriding the method
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks: Woof Woof!");
    }
}

// Main class
public class OverridingDemo {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Upcasting
        myAnimal.makeSound();  // Calls overridden method
in Dog
    }
}
```

static (class) members and non-static (instance) members.

- **Static Variable (static Counter):**
- Belongs to the class itself.
- Shared among all instances of Static Non Static Demo.
- Changes to this variable are reflected across all objects.
- Static Method (increment Static Counter()):
- Can be called directly using the class name (e.g., Static Non Static Demo. Increment Static Counter()).
- Can only directly access other static members.

- **Non-static Members:**
- Instance Variable (instance Counter):
- Each object (instance) of the class has its own separate copy.
- Changes to one object's instance Counter do not affect another object's value.
- Instance Method (increment Instance Counter()):
- Must be called on an instance of the class (e.g., obj1.incrementInstanceCounter()).
- Can access both instance variables and static variables.

Example :

```
class Student {
    // Static variable (shared by all objects)
    static String schoolName = "ABC High School";

    // Non-static variables (unique for each object)
    String name;
    int age;

    // Static method (can be called without object)
    static void showSchool() {
        System.out.println("School Name: " + schoolName);
    }
    // Non-static method (requires object)
    void showDetails() {
        System.out.println("Student Name: " + name + ", Age: " + age);
    }
}

// Main class
public class StaticDemo {
    public static void main(String[] args) {
        // Accessing static method directly (without object)
        Student.showSchool();

        // Creating objects
        Student s1 = new Student();
        Student s2 = new Student();

        // Assigning values to non-static variables
        s1.name = "Alice";
        s1.age = 18;

        s2.name = "Bob";
        s2.age = 19;

        // Calling non-static method (requires objects)
        s1.showDetails();
        s2.showDetails();

        // Accessing static variable through class name
        System.out.println("School (via class): " +
Student.schoolName);

        // Changing static variable affects all objects
        Student.schoolName = "XYZ Academy";
        System.out.println("Updated School: " + Student.schoolName);
    }
}
```

Garbage Collection

- In java garbage means unreferenced object .

- Garbage Collection is process of reclaiming the run time unused memory automatically.
- In other words, it is a way to destroy the unused objects.

Different Ways to Collect Garbage

- **By Making Reference to object null**
- **By Using an anonymous object**
- **By Pointing one's reference to another object or reference**
- **Example:**
- **Demo xyz=new demo();**
- **Xyz=null**
- **function demo(Student s){**
- **}**
- **Demo(Student s);**

Finalize method

- Finalize is a method of object class is a method that the garbage Collector always class just before the deletions of the object which is eligible for Garbage collections to perform clean up activity.
- Clen up activity means closing the resource associated with that object like Database connections ,Network connections or we can say resource de-allocations.
- Once the finalize() method complete, Garbage Collector destroys that objects
- Syntax: protected void finalize() throw Throwable
- Since it is available for every java class ,Garbage Collector can call the finalize methods on any java object
- Not Guaranteed to Run: There is no guarantee when (or if) finalize will be called.
- Advantages of finalize() Method
- Automatic Cleanup: Helps clean up resources (e.g., closing files, network connections) before an object is destroyed.

- Fallback Mechanism: Can act as a safety net in case resource management was overlooked.

Syntax of finalize() Method

```
class ClassName {
   // Override finalize method
   @Override
   protected void finalize() throws Throwable {
      // Cleanup code
   }
}
```

Example: Using finalize() in Java

```
class Demo {
   // Constructor
   Demo() {
      System.out.println("Object Created");
   }

   // Override finalize method
   @Override
   protected void finalize() throws Throwable {
      System.out.println("Object is being destroyed...");
   }
}

public class FinalizeDemo {
   public static void main(String[] args) {
      Demo obj1 = new Demo();  // Object created
      Demo obj2 = new Demo();

      // Nullify references (eligible for garbage collection)
      obj1 = null;
      obj2 = null;

      // Suggest JVM to run Garbage Collector
      System.gc();

      System.out.println("End of main method");
   }
}
```

IMP Questions:

- ❖ **Explain features of java(Buzzwrods)**
- ❖ **Explain history of java**
- ❖ **Explain Components of JDK**
- ❖ **Explain Data type in java**
- ❖ **Explain Java Operators**
- ❖ **Explain Abstract class and method**
- ❖ **Explain method overriding**
- ❖ **What is oop in java**
- ❖ **Explain Encapsulation**
- ❖ **Explain Inheritance**
- ❖ **Explain Polymorphism**
- ❖ **Explain Finalize method**
- ❖ **Explain Enum.**
- ❖ **Explain JAVA jump statement**
- ❖ **Explain jagged array**
- ❖ **Different between compile time and runtime Polymorphism**
- ❖ **Explain java operators**
- ❖ **Explain liters and keyword in java**

One liner Questions

- Java was developed in______
- JVM is _________
- JDK__________
- JRE____________
- __________Command for compile program.
  java is platform ______language
- Java extension file of java source file_____
- In Java, a literal is a fixed value that is assigned to a ____.
- A Boolean literal in Java can have only two values: ____ and ____.

- A character literal in Java is enclosed within ____ quotes.
- The null literal in Java is used to represent a ____ reference.
- The ____ statement is used to exit a loop or switch statement immediately.
- The continue statement is used to ____ the current iteration and proceed to the next iteration.
- The break statement is commonly used inside ____ statements to terminate case execution.
- The return statement is mainly used to exit from a ____ and return a value.