ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

LECTURE NOTES

COURSE: Object Oriented Programming with Java

COURSE CODE: 1BCS302

SEMESTER: III

COURSE COORDINATOR: Mrs. DIVYA N

(ACADEMIC YEAR 2026-27)

INSTITUTIONAL VISION AND MISSION

Vision

Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

• To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
• To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
• To strive to attain ever-higher benchmarks of educational excellence.

DEPARTMENT VISION AND MISSION

Vision

To be a global leader in Computer Science and Design Engineering striving for design excellence, cultural awareness, a profound commitment to environmental stewardship, and societal responsibility.

Mission

M1: To establish a technology-enabled experiential learning environment, prioritizing and cultivating problem-solving and design thinking skills among students.

M2: To foster collaboration with industries, research and development organizations, jointly addressing socially relevant challenges in Computer Science with a core emphasis on design.

Program Educational Objective (PEO's)

- Demonstrate the proficiency and adaptability needed to seamlessly integrate with the latest advancements and changes in Computer Science and Design.
- Our graduates will possess the adaptability to explore various career paths, such as Research & Development, Manufacturing, and Entrepreneurship with independent design firms, Design Services, and Consultancies.
- Our students will be well-prepared to excel in collaborative, multi-disciplinary environments, showcasing their versatility and skills.

Program Specific Outcomes (PSO's)

- To develop stand-alone, embedded, and web-based solutions with easy-to-operate interfaces using software engineering practices and contemporary programming languages.
- Design and develop computer-based systems in various areas of Multimedia, Graphics data visualization and computer vision.

An Overview of Java: Object-Oriented Programming (Two Paradigms, Abstraction, The Three OOP Principles), Using Blocks of Code, Lexical Issues (Whitespace, Identifiers, Literals, Comments, Separators, The Java Keywords).
Data Types, Variables, and Arrays: The Primitive Types (Integers, Floating-Point Types, Characters, Booleans), Variables, Type Conversion and Casting, Automatic Type Promotion in Expressions, Arrays, Introducing Type Inference with Local Variables.
Operators: Arithmetic Operators, Relational Operators, Boolean Logical Operators, The Assignment Operator, The ? Operator, Operator Precedence, Using Parentheses.
Control Statements: Java's Selection Statements (if, The Traditional switch), Iteration Statements (while, do-while, for, The For-Each Version of the for Loop, Local Variable Type Inference in a for Loop, Nested Loops), Jump Statements (Using break, Using continue, return).

Java is a high-level, class-based, object-oriented programming language designed to have as few implementation dependencies as possible. It is widely used for building cross-platform applications, from mobile apps to enterprise-level systems.

KEY FEATURES OF JAVA

- **Platform Independence**: Java is a "write once, run anywhere" (WORA) language. Java programs are compiled into bytecode, which can run on any machine with a Java Virtual Machine (JVM), regardless of the underlying operating system.
- **Object-Oriented**: Java is based on the principles of OOP (Object-Oriented Programming). Key OOP concepts include:
    - **Encapsulation**: Bundling data (attributes) and methods (functions) that operate on the data into a single unit (class).
    - **Inheritance**: A mechanism that allows one class to inherit properties and behaviours from another.
    - **Polymorphism**: The ability to process objects differently based on their data type or class.
    - **Abstraction**: Hiding the implementation details and showing only the essential features of the object.
- **Simple**: Java has a clean syntax that is easy to understand, which makes it accessible for beginners.
- **Secure**: Java has built-in security features such as bytecode verification, exception handling, and garbage collection that make it less prone to common security risks.
- **Robust**: Java emphasizes early checking for possible errors, with features like strong memory management and exception handling, reducing the likelihood of crashes.
- **Multithreading**: Java supports multithreading, allowing developers to write programs that can perform several tasks simultaneously.
- **High Performance**: Though Java is not as fast as natively compiled languages like C or C++, its performance is optimized through the use of Just-In-Time (JIT) compilation.

JAVA ARCHITECTURE

Components of Java Architecture

- **Java Virtual Machine (JVM)**: The JVM is an abstract machine that enables a computer to run Java programs. It converts Java bytecode into machine-specific instructions. JVMs are available for many hardware and software platforms.

- **Java Development Kit (JDK)**: A software development environment used for developing Java applications and applets. The JDK includes the Java Runtime Environment (JRE), an interpreter/loader, and other development tools.
- **Java Runtime Environment (JRE)**: A package of software that provides the Java class libraries, JVM, and other components required to run Java applications.

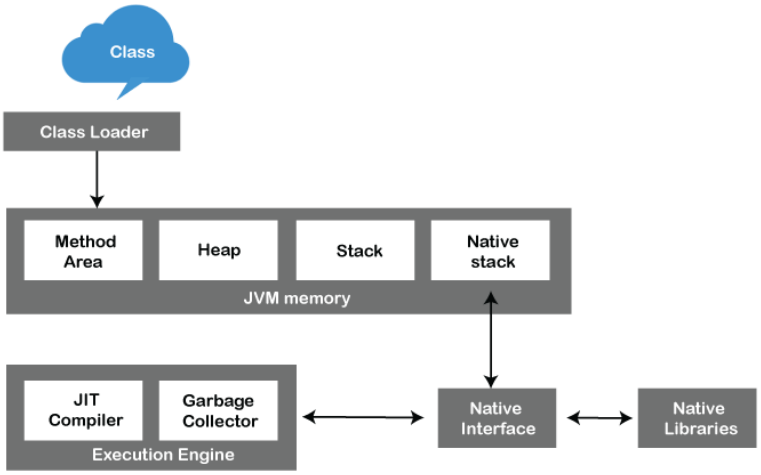

Fig 01: Components of Java Architecture

There are two processes in Java — Compilation and Interpretation.

- The Java source code goes to the compiler.
- The Java Compiler converts it into byte codes
- The bytes codes are then converted into machine by the JVM
- The Machine code is executed directly by the machine (Operating System)

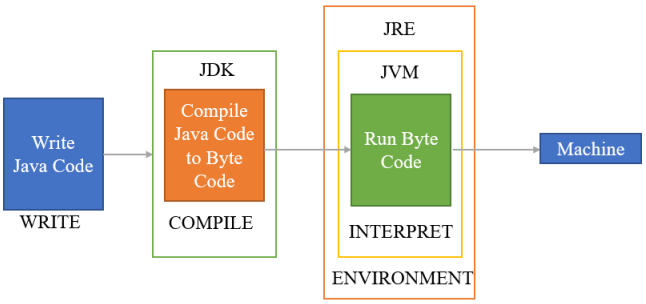

Fig 02: Main Processes involved in Java Program Execution

TWO PARADIGMS

In programming, paradigms refer to distinct styles or approaches to solving problems and organizing code. Here are two widely recognized paradigms:

1. Procedural Programming Paradigm

Definition: A programming paradigm that follows a step-by-step approach where a program is divided into a set of procedures or functions. These functions operate on data, perform operations, and produce results. The main focus is on the sequence of actions to be executed.

Key Concepts:

- Procedures/Functions: Blocks of code that perform a specific task.
- Sequence of Instructions: The program executes in a top-down manner, from one instruction to the next.
- Variables: Data is stored in variables, and the functions manipulate them.
- Modularity: A program is structured into smaller modules (functions), making it easier to maintain.

Languages: C, Fortran, Pascal, Basic.

2. Object-Oriented Programming (OOP) Paradigm

Definition: A programming paradigm that is based on the concept of "objects." Objects are instances of classes that combine both data (attributes) and methods (functions) to manipulate that data. OOP emphasizes modularity, code reuse, and abstraction.

Key Concepts:

- Class: A blueprint for creating objects, defining attributes and methods.
- Object: An instance of a class with specific values for its attributes.
- Encapsulation: Bundling data and methods together and restricting access to them.
- Inheritance: The ability of a new class to inherit properties and methods from an existing class.
- Polymorphism: The ability of different classes to respond to the same method call in different ways.
- Abstraction: Hiding the complex implementation details and showing only the essential features.

Languages: Java, C++, Python, Ruby.

THE THREE CORE PRINCIPLES OF OBJECT-ORIENTED PROGRAMMING (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that focuses on designing software using objects and classes. The three main principles of OOP—Encapsulation, Inheritance, and Polymorphism—play a fundamental role in designing modular, reusable, and maintainable code.

Encapsulation

Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on that data into a single unit, known as a **class**. It also restricts direct access to some of an object's components, which is known as **data hiding**. This ensures that the internal details of an object are hidden from the outside world, and access is controlled through **public methods** (getters and setters).

Benefits:

- **Data protection**: Prevents accidental modification of data.
- **Code maintainability**: Changes to the internal implementation do not affect external code that uses the class.

Example:

```
class Student {
private String name;  // private field
// Getter method to access the private field
public String getName() {
return name;
}
// Setter method to modify the private field
public void setName(String name) {
this.name = name;
}
}
```

Inheritance

Inheritance is the principle by which one class (the subclass or derived class) acquires the properties and behaviours (fields and methods) of another class (the superclass or base class). It allows code reusability and the extension of existing functionality without modifying the original class.

Key Points:

- **Superclass**: The parent class whose attributes and methods are inherited.
- **Subclass**: The child class that inherits from the superclass.
- **Single and multiple inheritance**: Java supports single inheritance (a class inherits from one parent class), but multiple inheritance is achieved using interfaces.

Benefits:

- **Code reusability**: Allows common code to be written once in the superclass and reused by subclasses.
- **Extensibility**: Subclasses can extend or modify the behavior of the superclass without altering its original code.

Example

```
      class Vehicle // Superclass
       {
         int speed;
         void displaySpeed() {
          System.out.println("Speed: " + speed);
         }
      }
      class Car extends Vehicle// Subclass
      {
         String model;
      }
public class Main {
         public static void main(String[] args) {
            Car myCar = new Car();
            myCar.speed = 100;
            myCar.model = "Tesla";
            myCar.displaySpeed();  // Accessing inherited method
         }
      }
```

Polymorphism

Definition: Polymorphism allows one interface to be used for different types of objects. It enables a method to do different things based on the object it is acting upon, even though the method call remains the same. Polymorphism can be **compile-time (method overloading)** or **runtime (method overriding)**.

Key Points:

- **Method Overloading (Compile-time polymorphism)**: Multiple methods with the same name but different parameters (number or type) can coexist in the same class.
- **Method Overriding (Runtime polymorphism)**: A subclass can provide a specific implementation of a method that is already defined in its superclass.

Benefits:

- **Flexibility**: Allows different objects to respond to the same method call in a way specific to their type.
- **Code maintainability**: Makes it easier to modify code or add new functionalities without affecting the existing system.

Example:

```
class Animal
 {
   void sound()
   {
      System.out.println("Animal makes a sound");
   }
}

class Dog extends Animal {
   @Override
   void sound() {
      System.out.println("Dog barks");
   }
}

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

USING BLOCKS OF CODE

In Java, blocks of code are sections of code enclosed within curly braces {} that group together related statements. These blocks are used in various contexts, such as in method definitions, conditionals, loops, classes, and exception handling. They help structure programs, define the scope of variables, and control the flow of execution.

- **Method Block:** A method block contains code that is executed when the method is called. The method block is defined between {} and is used to group related operations together.
- **Conditional Block:** Conditional blocks are used inside control structures like if, else if, and else to execute a set of statements when a certain condition is true or false.
- **Loop Block:** Loops use blocks of code to repeatedly execute a group of statements as long as a certain condition is met. for, while, and do-while are the common loop structures in Java.
- **Class Block:** A class block defines a blueprint for objects, containing fields (attributes) and methods (behaviors). All code inside a class is grouped into the class block.
- **Static Block:** A static block is a block of code that runs once when the class is loaded into memory, before the creation of any objects. It is used for static initialization of class-level variables.
- **Try-Catch Block (Exception Handling):** Java provides try-catch blocks to handle exceptions (runtime errors). The try block contains the code that might throw an exception, while the catch block contains code to handle that exception.
- **Synchronized Block:** A synchronized block is used to control access to a block of code by multiple threads, ensuring that only one thread can execute that block at a time.
- **Initialization Block (Instance Initializer):** Instance initializers are blocks of code that are executed when an object is created, before the constructor is called. These blocks are used to initialize instance variables.

Blocks of code in Java enhance program structure, define the scope of variables, manage flow control, and ensure better readability and maintainability.

LEXICAL ISSUE

lexical issues refer to problems or errors that occur during the lexical analysis phase of compiling a program. Lexical analysis is the first step in the compilation process where the source code is broken down into tokens such as keywords, identifiers, literals, operators, and delimiters. If the code contains invalid tokens or violates the lexical structure of the language, lexical errors occur.

- **Whitespace:** Whitespace includes spaces, tabs, and newlines, and is generally ignored by the Java compiler unless it appears within tokens like identifiers or literals.
    - Purpose: Used to separate tokens and improve code readability.
    - Lexical Issues: Using whitespace within identifiers or literals can cause errors.

int my var = 10; // Invalid: Variable name cannot contain spaces

- **Identifiers: Identifiers** are names used for variables, methods, classes, etc. They follow specific rules:

  Rules for Identifiers:

  - Must begin with a letter (A-Z, a-z), an underscore (_), or a dollar sign ($).
  - Subsequent characters can include digits (0-9), letters, underscores, or dollar signs.
  - Cannot be a Java keyword or reserved word.

  Lexical Issues:

  - Using invalid characters in identifiers.
  - Starting identifiers with digits.
  - Using Java keywords as identifiers

int 1number = 10; // Invalid: Cannot start an identifier with a digit
int class = 20; // Invalid: 'class' is a keyword and cannot be used as an identifier

- **Literals:** Literals represent fixed values in Java, such as numbers, characters, and strings. There are several types of literals in Java: integer literals, floating-point literals, boolean literals, character literals, and string literals.
  Types of Literals:
  - **Integer Literals**: Represent whole numbers (e.g., 10, 0, -5).
  - **Floating-Point Literals**: Represent decimal numbers (e.g., 3.14, 0.5).
  - **Boolean Literals**: Represent true or false.
  - **Character Literals**: Represent a single character enclosed in single quotes (e.g., 'A').
  - **String Literals**: Represent a sequence of characters enclosed in double quotes (e.g., "Hello").

  Lexical Issues:
  - Incorrectly formatted literals (e.g., forgetting quotes).
  - Using invalid escape sequences in strings.
  - Using underscores (_) improperly in numeric literals.

```
char c = "A"; // Invalid: Character literals must use single quotes
int number = 1_000_000_; // Invalid: Underscores cannot be at the end of a numeric literal
```

- **Comments:** Comments in Java are non-executable statements used for documentation purposes. They are ignored by the compiler.
  Types of Comments:
  - Single-line comment: Starts with //.
  - Multi-line comment: Starts with /* and ends with */.

  Lexical Issues:
  - Forgetting to close multi-line comments.

```
/* This is a comment // Invalid: Multi-line comment is not closed
int x = 5;
```

- **Separators:** Separators are symbols that separate or group individual code elements in Java. Java uses the following separators:
  Common Separators:
  - () - Parentheses: Used in method calls and control flow structures.
  - {} - Curly Braces: Used for blocks of code.
  - [] - Brackets: Used for arrays.
  - ; - Semicolon: Ends statements.
  - , - Comma: Separates variables in declarations.
  - . - Dot: Used to access object members.

  Lexical Issues:
  - Missing or misplaced separators can cause syntax errors during lexical analysis.

```
int x = 5 // Invalid: Missing semicolon
if (x > 5 { // Invalid: Missing closing parenthesis
System.out.println(x);
}
```

- **The Java Keywords**: Keywords are reserved words in Java that have predefined meanings and cannot be used as identifiers. Examples include class, int, if, else, for, while, try, catch, etc.
  Lexical Issues:
  - Using a keyword as an identifier or variable name will result in an error.
  - Misspelling keywords (e.g., writing Clas instead of class) can also cause errors.

abstract	double	int	strictfp
boolean	else	interface	super
break	extends	long	switch
byte	final	native	this
case	finally	new	throw
catch	float	package	throws
char	for	private	transient
class	goto	protected	try
const	if	public	void
continue	implements	return	volatile
default	import	short	while
do	instanceof	static	synchronized

int if = 10; // Invalid: 'if' is a reserved keyword and cannot be used as an identi□ier clas MyClass { // Invalid: 'clas' is not a valid keyword (should be 'class') int x = 5; }

DATA TYPES

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

- **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
- **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

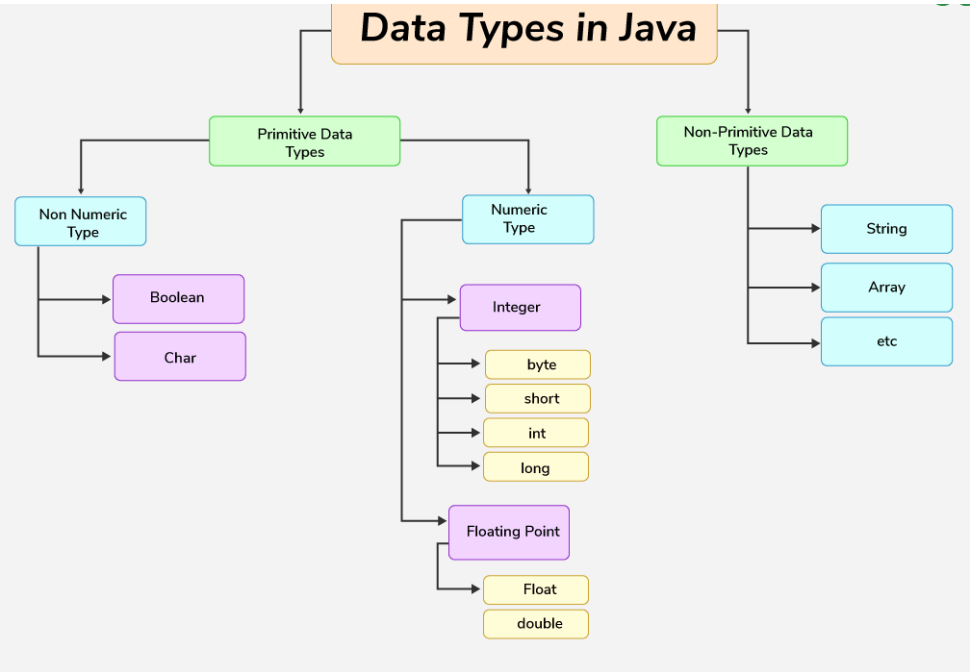

Fig 03: Data types available in java

- **Integer Types:** Used for storing whole numbers (without fractions).

Data Type	Size	Minimum Value	Maximum Value	Default Value
byte	8 bits	-128	127	0
short	16 bits	-32,768	32,767	0
int	32 bits	-2^31	2^31-1	0
long	64 bits	-2^63	2^63-1	0L

long is used when int is not large enough to hold a specific value. It is often used with the suffix **L** (e.g., 100L).

- **Floating-Point Types:** Used for storing numbers with decimal points.

Data Type	Size	Range	Precision	Default Value
float	32 bits	±1.4E-45 to ±3.4E38	7 digits	0.0f
double	64 bits	±4.9E-324 to ±1.7E308	15 digits	0.0d

double is the default for decimal values. **float** requires the suffix **f** (e.g., 3.14f).

- **Character Type:** Used for storing a single 16-bit Unicode character.

Data Type	Size	Range	Default Value
char	16 bits	0 to 65,535	'\u0000'

Unicode allows Java to represent international characters like Chinese, Japanese, etc.

- **Boolean Type:** Represents two possible states: true or false

Data Type	Size	Values	Default Value
boolean	1 bit	true, false	false

```java
public class DataTypes {
    public static void main(String[] args) {

        // Integer data types
        byte byteVar = 100;
        short shortVar = 30000;
        int intVar = 500000;
        long longVar = 9000000000L;  // 'L' suffix for long literal
        // Floating-point data types
        float floatVar = 5.75f;  // 'f' suffix for float literal
        double doubleVar = 19.99; // No suffix needed for double
        // Character data type
        char charVar = 'A';  // Single character in single quotes
        // Boolean data type
        boolean boolVar = true;
        // Array data type (Integer array)
        int[] intArray = {10, 20, 30, 40, 50};

        // Displaying values
        System.out.println("=== Integer Data Types ===");
        System.out.println("Byte: " + byteVar);
        System.out.println("Short: " + shortVar);
        System.out.println("Integer: " + intVar);
        System.out.println("Long: " + longVar);

        System.out.println("\n=== Floating-Point Data Types ===");
        System.out.println("Float: " + floatVar);
        System.out.println("Double: " + doubleVar);

        System.out.println("\n=== Character Data Type ===");
        System.out.println("Character: " + charVar);

        System.out.println("\n=== Boolean Data Type ===");
        System.out.println("Boolean: " + boolVar);

        System.out.println("\n=== Array Data Type ===");
        System.out.print("Array elements: ");
        for (int num : intArray) {
            System.out.print(num + " ");
        }
    }
}
```

Example: Demonstrating Datatypes available in Java

VARIABLES IN JAVA: A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type. Variable is a name of memory location. Variables store data that can be used and manipulated throughout the program. Java variables must be declared with a data type before use.

There are three types of variables in java: local, instance and static.

1. **Local Variables**: Declared inside methods, constructors, or blocks. These variables are not accessible outside their specific method or block.
    - No default value; must be initialized before use.
2. **Instance Variables**: Declared in a class but outside of any method, constructor, or block. Each object of the class has its own copy.
    - Default values depend on the data type (0 for numbers, null for objects).
3. **Static Variables**: Declared using the static keyword. Shared by all instances of the class.
    - Can be accessed directly using the class name without creating an object.

int age = 25; // Declaring and initializing an int variable
double salary = 50000.50; // Declaring and initializing a double variable

Variable Naming Rules:

- Can contain letters, digits, underscores, and dollar signs.
- Must begin with a letter (or _, $).
- No spaces or keywords are allowed.

```java
public class VariableTypesDemo {

    // Instance variable (non-static)
    String instanceVar = "I am an instance variable";

    // Static variable
    static int staticVar = 100;

    public void displayVariables() {
        // Local variable
        int localVar = 50;

        System.out.println("=== Variable Types Demo ===");
        System.out.println("Local Variable: " + localVar);
        System.out.println("Instance Variable: " + instanceVar);
        System.out.println("Static Variable: " + staticVar);
    }

    public static void main(String[] args) {
        // Creating an object of the class
        VariableTypesDemo demo = new VariableTypesDemo();

        // Calling method to display variable values
        demo.displayVariables();

        // Accessing static variable directly from the class
        System.out.println("Accessing Static Variable from Class: " + VariableTypesDemo.staticVar);
    }
}
```

Example: Demonstrating Variable Types in Java

Type Conversion and Type Casting

The two terms **type casting** and the **type conversion** are used in a program to convert one data type to another data type. The conversion of data type is possible only by the compiler when they are compatible with each other.

If a data type is automatically converted into another data type at compile time is known as ***type conversion.*** The conversion is performed by the compiler if both data types are compatible with each other. Remember that the destination data type should not be smaller than the source type. It is also known as **widening** conversion of the data type.

When a data type is converted into another data type by a programmer or user while writing a program code of any programming language, the mechanism is known as ***type casting***. The programmer manually uses it to convert one data type into another. It is used if we want to change the target data type to another data type. Remember that the destination data type must be smaller than the source data type. Hence it is also called a narrowing conversion.

SLNO	TYPE CASTING	TYPE CONVERSION
1	Type casting is a mechanism in which one data type is converted to another data type using a casting () operator by a programmer.	Type conversion allows a compiler to convert one data type to another data type at the compile time of a program or code.
2	It can be used both compatible data type and incompatible data type.	Type conversion is only used with compatible data types, and hence it does not require any casting operator.
3	It requires a programmer to manually casting one data into another type.	It does not require any programmer intervention to convert one data type to another because the compiler automatically compiles it at the run time of a program.
4	It is used while designing a program by the programmer.	It is used or take place at the compile time of a program.
5	When casting one data type to another, the destination data type must be smaller than the source data.	When converting one data type to another, the destination type should be greater than the source data type.
6	It is also known as narrowing conversion because one larger data type converts to a smaller data type.	It is also known as widening conversion because one smaller data type converts to a larger data type.
7	It is more reliable and efficient.	It is less efficient and less reliable.
8	There is a possibility of data or information being lost in type casting.	In type conversion, data is unlikely to be lost when converting from a small to a large data type.
9	*float b = 3.0;* *int a = (int) b*	*int x = 5, y = 2, c;* *float q = 12.5, p; p = q/x;*

Arrays: Array is an object which contains elements of a similar data type. In Java, an **array** is a data structure that allows you to store multiple values of the same type in a single variable. It is a fixed-size, ordered collection of elements that can be accessed using an index.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Declaration and Initialization

- **Declaration**: To declare an array, you specify the type of elements it will hold, followed by square brackets.
  Syntax
  - ***datatype[ ] arrayname;***
  - ***datatype arrayname [ ];***
  - ***datatype [ ]arrayname;***

- **Initialization**: You can initialize an array at the time of declaration or separately.
  - **Static Initialization**:
    int[] numbers = {1, 2, 3, 4, 5};
  - **Dynamic Initialization**:
    int[] numbers = new int[5];

- **Accessing Array Elements:** Array elements can be accessed using their index, which starts at 0.

  int firstElement = numbers[0]; // Accessing the first element numbers[1] = 10; // Modifying the second element

- **Array Length:** The length of an array can be found using the length property.

  int length = numbers.length; // Gets the length of the array

Multidimensional Arrays

Java also supports multidimensional arrays, primarily two-dimensional arrays (matrices).

Declaration and Initialization:

int[][] matrix = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

Common Operations

- **Traversing an Array:** You can use a loop to iterate over an array.

```
for (int i = 0; i < numbers.length; i++)
{
    System.out.println(numbers[i]); // Prints each element
}
```

- **Enhanced For Loop:** Java provides a simpler way to iterate through arrays.

```
for (int number : numbers) {
    System.out.println(number); // Prints each element
}
```

Refer: LAB: Program-01

OPERATORS

Operators are special symbols or keywords in programming languages that are used to perform operations on variables and values. They are fundamental building blocks that allow you to manipulate data, control the flow of a program, and perform calculations.

Types of Operators

1. **Arithmetic Operators**: Used to perform mathematical calculations, such as addition, subtraction, multiplication, and division.

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder)	a % b

Example Snippet

```
int a = 10, b = 3;
System.out.println("Addition: " + (a + b));       // 13
System.out.println("Subtraction: " + (a - b));    // 7
System.out.println("Multiplication: " + (a * b)); // 30
System.out.println("Division: " + (a / b));        // 3
System.out.println("Modulus: " + (a % b));         // 1
```

2. **Relational Operators**: Used to compare two values or expressions. They return a boolean result (true or false).

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b

Example Snippet:

```
int a = 5, b = 10;
System.out.println("Is a equal to b? " + (a == b)); // false
System.out.println("Is a not equal to b? " + (a != b)); // true
System.out.println("Is a greater than b? " + (a > b)); // false
System.out.println("Is a less than b? " + (a < b)); // true
```

3. **Boolean Logical Operators:** Boolean logical operators are used to combine multiple boolean expressions

Operator	Description	Example
&&	Logical AND	a && b
`		`
!	Logical NOT	!a

Example Snippet

```
boolean a = true, b = false;
System.out.println("Logical AND: " + (a && b)); // false
System.out.println("Logical OR: " + (a || b));  // true
System.out.println("Logical NOT: " + (!a));      // false
```

4. **Bitwise Operators**: Operate on binary representations of integers, allowing manipulation of individual bits.
   - ***& (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), >> (right shift)***
5. **Assignment Operators**: Used to assign values to variables. These can also include compound assignment operators that combine an arithmetic operation with an assignment.

Operator	Description	Example
=	Assigns the right-hand value to the left-hand variable	a = 10

Compound Assignment Operators:

Operator	Description	Example
+=	Add and assign	a += 5 (same as a = a + 5)
-=	Subtract and assign	a -= 3 (same as a = a - 3)
*=	Multiply and assign	a *= 2 (same as a = a * 2)
/=	Divide and assign	a /= 2 (same as a = a / 2)
%=	Modulus and assign	a %= 2 (same as a = a % 2)

Example Snippet

```
int a = 10;
a += 5; // a = 15
System.out.println("After += : " + a);
a -= 3; // a = 12
System.out.println("After -= : " + a);
```

6. **Ternary Operator**: A shorthand for if-else statements that evaluates a condition and returns one of two values based on whether the condition is true or false.

condition ? expression1 : expression2

Example Snippet

```
int a = 10, b = 20;
int max = (a > b) ? a : b; // Assigns the greater of a or b to max
System.out.println("Max value is: " + max); // Output: 20
```

7. **Unary Operators**: Operate on a single operand to produce a new value. These can include incrementing, decrementing, negating, etc.
   - ***++, --, - (unary negation)***
8. **Logical Operators**: Used to combine multiple boolean expressions and perform logical operations.
   - ***&& (AND), || (OR), ! (NOT)***
9. **Instanceof Operator**: Used to test whether an object is an instance of a specific class or subclass.
   - ***object instanceof ClassName***

Operator Precedence

Operator precedence determines the order in which operators are evaluated in an expression. Higher precedence operators are evaluated first.

- **Common Precedence** (from highest to lowest):
    1. Parentheses ()
    2. Unary operators ++, --, -, +, !
    3. Multiplication, Division, Modulus *, /, %
    4. Addition, Subtraction +, -
    5. Relational operators <, >, <=, >=, ==, !=
    6. Logical AND &&
    7. Logical OR ||
    8. Assignment =

Using Parentheses: Using parentheses can clarify expressions and override the default precedence.

Example Snippet

```
int a = 10, b = 5, c = 3;
int result = a + b * c; // Without parentheses: result = 10 + (5 * 3) = 25
System.out.println("Result without parentheses: " + result);

result = (a + b) * c; // With parentheses: result = (10 + 5) * 3 = 45
System.out.println("Result with parentheses: " + result);
```

CONTROL STATEMENTS IN JAVA

Control statements allow you to dictate the flow of execution in your Java programs. They can be categorized into three main types: selection statements, iteration statements, and jump statements.

1. **Selection Statements:** Selection statements enable the program to execute different blocks of code based on certain conditions.

- **if Statement:** The if statement evaluates a boolean expression. If the expression is true, it executes the block of code within the statement.

Syntax and Code Snippet

```
if (condition) {
    // code to be executed if condition is true
}

int number = 10;
if (number > 0) {
    System.out.println("The number is positive.");
}
```

- **switch Statement:** The switch statement allows you to evaluate a variable against a list of values (cases). It is useful when you have multiple conditions based on the same variable.

Syntax and Code Snippet

```
switch (variable) {
    case value1:
        // code to execute if variable == value1
        break;
    case value2:
        // code to execute if variable == value2
        break;
    // ...
    default:
        // code to execute if none of the cases match
}

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");
}
```

2. **Iteration Statements:** Iteration statements allow you to execute a block of code multiple times.

- **while Loop:** The while loop continues to execute as long as the specified condition is true.

Syntax and Code Snippet

```
while (condition) {
    // code to be executed
}
int i = 0;
while (i < 5) {
    System.out.println("i: " + i);
    i++;
}
```

- **do-while loop:** The do-while loop is similar to the while loop, but it guarantees that the code block will execute at least once, as the condition is checked after the execution.

Syntax and Code Snippet

```
do {
    // code to be executed
} while (condition);
int j = 0;
do {
    System.out.println("j: " + j);
    j++;
} while (j < 5);
```

- **for Loop:** The for loop provides a concise way to iterate through a range of values.

Syntax and Code Snippet

```
for (initialization; condition; increment) {
    // code to be executed
}
for (int k = 0; k < 5; k++) {
    System.out.println("k: " + k);
}
```

- **For-Each Loop:** The for-each loop is specifically used to iterate over arrays or collections. It simplifies the iteration process.

Syntax and Code Snippet

```
for (dataType item : array) {
    // code to be executed for each item
}
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
    System.out.println(num);
}
```

- **Local Variable Type Inference in a For Loop:** From Java 10 onwards, you can use var to infer the type of the loop variable.

Syntax and Code Snippet

```
var numbersList = List.of(1, 2, 3, 4, 5);
for (var num : numbersList) {
    System.out.println(num);
}
```

- **Nested Loops:** You can also nest loops inside one another to perform more complex iterations.

Syntax and Code Snippet

```
for (int m = 0; m < 3; m++) {
    for (int n = 0; n < 3; n++) {
        System.out.println("m: " + m + ", n: " + n);
    }
}
```

3. **Jump Statements:** Jump statements are used to alter the flow of control within loops or switch statements.

- **break Statement:** The break statement is used to exit a loop or switch statement prematurely.

Syntax and Code Snippet

```
for (int p = 0; p < 5; p++) {
    if (p == 3) {
        break; // exit the loop when p is 3
    }
    System.out.println("p: " + p);
}
```

- **continue Statement:** The continue statement skips the current iteration of a loop and continues with the next iteration.

Syntax and Code Snippet

```
for (int q = 0; q < 5; q++) {
    if (q == 2) {
        continue; // skip the rest of the loop when q is 2
    }
    System.out.println("q: " + q);
}
```

- **return Statement:** The return statement is used to exit from a method and optionally return a value.

Syntax and Code Snippet

```
public int sum(int a, int b) {
    return a + b; // exits the method and returns the sum
}
```