

MODULE-1

CHAPTER 1

AN OVERVIEW OF JAVA

Introduction

Java is a high-level, object-oriented programming language. Its features are interconnected and work together as a complete language. Understanding one feature often requires knowledge of other features. This chapter provides a basic overview of Java to help beginners write and understand simple programs.

I. OBJECT-ORIENTED PROGRAMMING (OOP)

Object-Oriented Programming (OOP) is the core concept of Java. Almost all Java programs are based on OOP principles using classes and objects. OOP helps in developing software that is reusable, secure, and easy to maintain. Understanding OOP is essential before learning Java programming in detail.

i. Two Paradigms

Computer programs consist of two main elements: **code** and **data**, and they can be organized using two programming paradigms. The **process-oriented paradigm** focuses on code, where programs are executed as a sequence of steps that operate on data. Languages like **C** follow this approach, but it becomes difficult to manage as programs grow larger and more complex. To overcome these limitations, the **object-oriented paradigm** was introduced, which organizes programs around **objects (data)** and the methods that operate on them. In object-oriented programming, data controls access to code, making programs more organized, reusable, secure, and easier to maintain.

ii. Abstraction

Abstraction is one of the fundamental concepts of Object-Oriented Programming (OOP). It helps manage complexity by hiding unnecessary details and showing only the essential features of an object. For example, people use a car without knowing how its engine, transmission, or braking system works. They simply interact with the car as a single object, making it easier to use without understanding its internal complexity.

In Java, abstraction is achieved by representing complex systems as objects with specific behaviours and well-defined interfaces. Large programs are divided into smaller, manageable objects that communicate with each other through messages (methods). This approach improves code organization, simplifies program development, enhances reusability, and makes software easier to maintain and update throughout its life cycle.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Encapsulation

Encapsulation is one of the three fundamental principles of Object-Oriented Programming (OOP). It is the process of binding **data (variables)** and the **methods (functions)** that operate on the data into a single unit called a **class**. It also protects the data from unauthorized access by allowing it to be accessed only through a well-defined interface. This improves data security and prevents misuse.

In Java, encapsulation is implemented using **classes** and **objects**. A class defines the structure and behaviour of objects by containing member variables (data) and member methods (functions). Objects created from a class are called **instances** of that class. The methods control how the data is accessed and modified, ensuring proper interaction with the object's data.

Java provides **access modifiers** such as **private** and **public** to achieve encapsulation. Private members can only be accessed within the same class, while public members can be accessed from outside the class. This hides the internal implementation of a class, protects data from accidental modification, and makes programs more secure, modular, and easier to maintain.

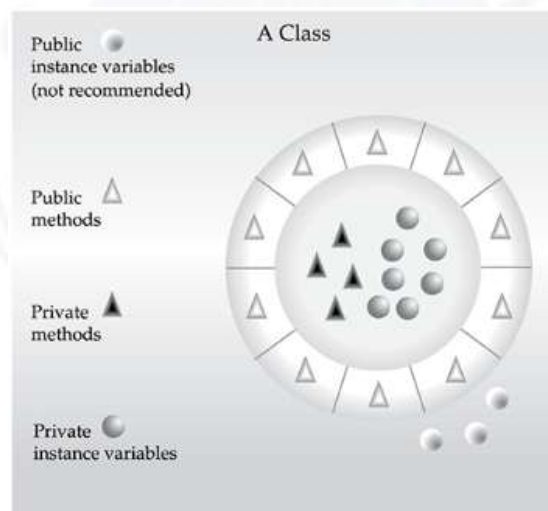


Figure 2-1 Encapsulation: public methods can be used to protect private data.

Inheritance

Inheritance is one of the fundamental principles of Object-Oriented Programming (OOP). It is the process by which one class acquires the properties and methods of another class. The existing class is called the **superclass (parent class)**, and the new class is called the **subclass (child class)**. Inheritance supports hierarchical classification, allowing common features to be defined once in the parent class and reused by child classes, reducing code duplication.

In Java, a subclass inherits all the accessible attributes and methods of its superclass and can also add new features or modify existing behavior.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

This promotes **code reusability**, simplifies program development, and makes applications easier to maintain and extend. Inheritance also works with encapsulation, enabling large software systems to grow in an organized and efficient manner.

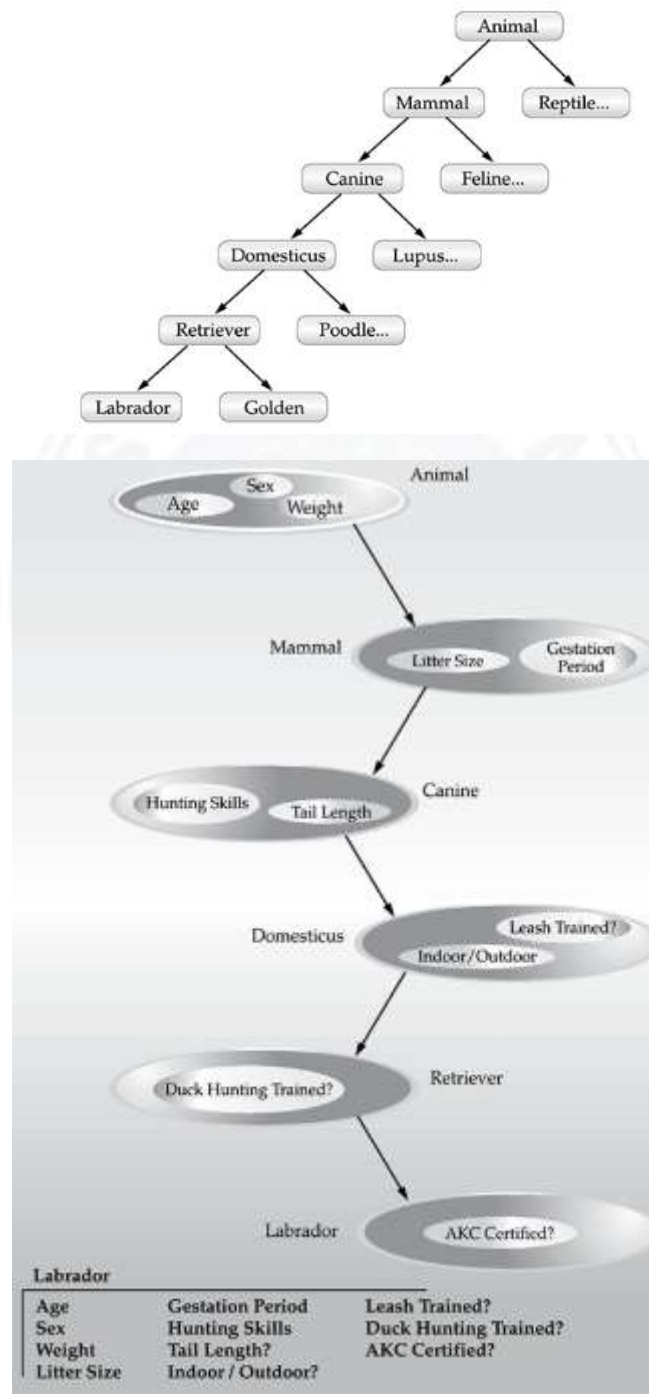


Figure 2-2 Labrador inherits the encapsulation of all its superclasses.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Polymorphism

Polymorphism, which means "**many forms**," is one of the three fundamental principles of Object-Oriented Programming (OOP). It allows a single interface or method name to perform different tasks depending on the object or data involved. In Java, the same method name can be used for different operations, making programs more flexible and reducing the need to create multiple methods with different names.

Polymorphism is often described as "**one interface, multiple methods**." The compiler or the Java runtime automatically selects the appropriate method based on the situation. This improves code reusability, simplifies programming, reduces complexity, and makes applications easier to maintain and extend.

iii. Polymorphism, Encapsulation, and Inheritance Work Together

Encapsulation, inheritance, and polymorphism are the three core principles of Object-Oriented Programming (OOP), and together they make Java programs more robust, reusable, scalable, and easy to maintain. **Encapsulation** protects data by hiding implementation details, **inheritance** enables code reusability by allowing one class to acquire the features of another, and **polymorphism** provides flexibility by allowing the same interface to perform different tasks.

A real-world example of these principles is an **automobile**. Drivers can operate different types of vehicles using the same controls, such as the steering wheel, brake pedal, and accelerator. The internal working of the engine, transmission, or braking system is hidden from the driver (encapsulation), different vehicle models inherit common features (inheritance), and the same controls work with different vehicle designs and technologies (polymorphism).

In Java, these three principles work together to build efficient, organized, and maintainable applications. They reduce code duplication, improve security, simplify software development, and allow programs to grow without affecting existing code. Therefore, every Java program is based on the combined use of encapsulation, inheritance, and polymorphism.

A First Simple Program

Now that the basic object-oriented underpinning of Java has been discussed, let's look at some actual Java programs. Let's start by compiling and running the short sample program shown here. As you will see, this involves a little more work than you might imagine.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
/*
   This is a simple Java program.
   Call this file "Example.java".
*/
class Example {
    // Your program begins with a call to main().
    public static void main(String[] args) {
        System.out.println("This is a simple Java program.");
    }
}
```

NOTE The descriptions that follow use the standard Java SE Development Kit (JDK), which is available from Oracle. (Open source versions are also available.) If you are using an integrated development environment (IDE), then you will need to follow a different procedure for compiling and executing Java programs. In this case, consult your IDE's documentation for details.

In Java, the source file is called a **compilation unit** and must have the **.java** file extension. By convention, the **filename should match the class name**, including the same capitalization, because Java is **case-sensitive**. For example, if the class name is `Example`, the source file must be named **Example.java**. Following this naming convention helps organize programs properly and is required in certain situations.

Compiling the Program

To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

```
C:\>javac Example.java
```

The **javac compiler** converts the Java source file into a **.class file** containing **bytecode**, which is executed by the **Java Virtual Machine (JVM)**.

To actually run the program, you must use the Java application launcher called **java**. To do so, pass the class name **Example** as a command-line argument, as shown here:

```
C:\>java Example
```

When the program is run, the following output is displayed:

```
This is a simple Java program.
```

Each Java class is compiled into a separate **.class file**, and the **java command** executes the specified class by locating and running its corresponding **.class file**.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

A Closer Look at the First Sample Program

Although **Example.java** is quite short, it includes several key features that are common to all Java programs. Let's closely examine each part of the program.

The program begins with the following lines:

The next line in the program is the *single-line comment*, shown here:

```
// Your program begins with a call to main().
```

```
/*  
    This is a simple Java program.  
    Call this file "Example.java".  
*/
```

Comments are explanatory statements in a Java program that are ignored by the compiler and help readers understand the code. Java supports **three types of comments**, including **multiline comments**, which begin with `/*` and end with `*/` and can span multiple lines.

The next line of code in the program is shown here:

```
class Example {
```

The **class** keyword is used to define a new class, and **all Java program code is written inside a class**, making Java an object-oriented language.

The next line in the program is the *single-line comment*, shown here:

```
// Your program begins with a call to main().
```

A **single-line comment** begins with `//` and continues to the end of the line, and is commonly used for brief explanations in the code.

The next line of code is shown here:

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

The main() Method

The **main()** method is the starting point of every Java application, and program execution begins from this method. It must be declared as **public static void main(String[] args)**, where **public** allows access from outside the class, **static** enables the method to run without creating an object, and **void** indicates that it does not return any value.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

The parameter **String[] args** is used to receive command-line arguments passed to the program during execution. Even if no arguments are provided, the parentheses and parameter declaration must be included in the method definition.

The body of the **main()** method is enclosed within curly braces { }, which contain the executable statements of the program. Although a Java program can have many classes and methods, only one **main()** method is required as the entry point for program execution.

The next line of code is shown here. Notice that it occurs inside **main()**.

```
System.out.println("This is a simple Java program.");
```

System.out.println() Statement

The **System.out.println()** statement is used to display output on the console, followed by a new line. Here, **System** is a predefined class, **out** is the standard output stream, and **println()** prints the specified text or value. Each Java statement ends with a **semicolon (;)**, and curly braces { } are used to mark the beginning and end of methods and classes.

A Second Short Program

Variables

A **variable** is a named memory location used to store data that can change during program execution. In Java, every variable must be **declared before it is used**. A variable declaration specifies its **data type** and **name**, such as `int num;`, where `int` represents an integer type. Multiple variables of the same type can be declared in a single statement using commas.

Variable Assignment and Output

A value is assigned to a variable using the **assignment operator (=)**. For example, `num = 100;` stores the value 100 in the variable `num`. The **System.out.println()** method displays output on the console, and the **+** **operator** joins strings with variable values. The **System.out.print()** method is similar to `println()`, but it does not move the cursor to the next line after printing.

Two Control Statements

The if Statement

The **if statement** is a decision-making statement that executes a block of code only when a specified **Boolean condition** is true. If the condition evaluates to **false**, the statement is skipped.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

The general syntax is:

```
if(condition)
```

```
    statement;
```

Java supports several **relational operators** such as <, >, <=, >=, ==, and != to compare values in conditional expressions.

The for Loop

The **for loop** is used to repeatedly execute a block of code for a specified number of times. It consists of three parts: **initialization**, **condition**, and **iteration**. The loop continues executing as long as the condition remains true.

```
for(initialization; condition; iteration)
```

```
    statement;
```

The **increment operator** (++) increases a variable by one, while the **decrement operator** (--) decreases it by one. These operators are commonly used in for loops because they make the code shorter and more efficient.

iv. Using Blocks of Code

A **block of code** is a group of statements enclosed within **curly braces** {}, which acts as a single logical unit and can be used wherever a single statement is allowed, such as in **if** and **for** statements.

```
if(x < y) { // begin a block
    x = y;
    y = 0;
} // end of block
```

If the condition is **true**, all the statements inside the **code block** are executed together, making the block a single logical unit where each statement executes as part of the same group.

Let's look at another example. The following program uses a block of code as the target of a **for** loop:

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
/*
    Demonstrate a block of code.

    Call this file "BlockTest.java"
*/
class BlockTest {
    public static void main(String[] args) {
        int x, y;

        y = 20;

        // the target of this loop is a block
        for(x = 0; x<10; x++) {
            System.out.println("This is x: " + x);
            System.out.println("This is y: " + y);
            y = y - 2;
        }
    }
}
```

The output generated by this program is shown here:

```
This is x: 0
This is y: 20
This is x: 1
This is y: 18
This is x: 2
This is y: 16
This is x: 3
This is y: 14
This is x: 4
This is y: 12
This is x: 5
This is y: 10
This is x: 6
This is y: 8
This is x: 7
This is y: 6
This is x: 8
This is y: 4
This is x: 9
This is y: 2
```

When a **block of code** is used with a **for loop**, **all the statements inside the block are executed during each iteration**, allowing multiple statements to function as a single logical unit.

v. Lexical Issues

Lexical issues refer to the basic building blocks of a Java program. A Java program is made up of **whitespace, identifiers, literals, comments, operators, separators, and keywords**, which together form the syntax of the language.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

a. Whitespace

Whitespace refers to spaces, tabs, newlines, and form feeds used to improve the readability of Java programs. Since Java is a **free-form language**, programmers can format code in any style without affecting program execution.

b. Identifiers

Identifiers are names given to classes, variables, methods, and other program elements. They can contain letters, digits, underscores (_), and dollar signs (\$), but **must not begin with a number**, and Java treats uppercase and lowercase letters as different (case-sensitive).

AvgTemp	count	a4	\$test	this_is_ok
---------	-------	----	--------	------------

Invalid identifier names include these:

2count	high-temp	Not/ok
--------	-----------	--------

NOTE Beginning with JDK 9, the underscore cannot be used by itself as an identifier.

c. Literals

A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:

100	98.6	'X'	"This is a test"
-----	------	-----	------------------

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

d. Comments

Java supports **three types of comments**: **single-line** (//), **multiline** (/* ... */), and **documentation** (/** ... */) comments. Comments are ignored by the compiler and are used to explain the program or generate documentation.

Separators

e. Separators

These are special characters that organize Java code, with the **semicolon (;)** being the most commonly used to terminate statements. Other separators include **parentheses ()**, **braces {}**, **brackets []**, **commas ,**, and **periods .**

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Symbol	Name	Purpose
()	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference.
...	Ellipsis	Indicates a variable-arity parameter.
@	At-sign	Begins an annotation.

f. Java Keywords

Java keywords are predefined words with special meanings that form the foundation of the Java language. Java currently has **68 keywords**, including **51 reserved words**, which cannot be used as identifiers, and **17 contextual keywords**, which have special meanings only in specific contexts.

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
module	native	new	non-sealed	open	opens
package	permits	private	protected	provides	public
record	requires	return	sealed	short	static
strictfp	super	switch	synchronized	this	throw
throws	to	transient	transitive	try	uses
var	void	volatile	when	while	with
yield	-				

Table 2-1 Java Keywords

The keywords **const** and **goto** are **reserved** but are **not used** in Java. Additionally, **true**, **false**, and **null** are reserved literal values and **cannot be used as identifiers** such as variable, class, or method names.

CHAPTER-2

DATA TYPES, VARIABLES, AND ARRAYS

This chapter introduces the three fundamental elements of Java: data types, variables, and arrays. Java provides several built-in data types that are used to declare variables and create arrays, offering a simple, efficient, and well-organized approach to data handling.

Java is a Strongly Typed Language

Java is a **strongly typed language**, meaning that **every variable, expression, and value has a specific data type**. The Java compiler checks **type compatibility** during assignments and method calls, and any type mismatch results in a **compile-time error**, ensuring safer and more reliable programs.

I. The Primitive Types

Java defines **eight primitive (simple) data types**: **byte, short, int, long, float, double, char, and boolean**. These are grouped into **integers** (byte, short, int, long), **floating-point types** (float, double), **character type** (char), and **Boolean type** (boolean). They are used to store simple values and serve as the foundation for arrays and user-defined classes.

Primitive data types are **not objects**, which makes them faster and more memory-efficient. Each primitive type has a **fixed size and range**, ensuring that Java programs behave consistently on all platforms. For example, an **int is always 32 bits**, which improves Java's portability and reliability.

i. Integer

Java provides **four integer data types**: **byte, short, int, and long**, all of which store **signed (positive and negative) whole numbers**. Java does **not support unsigned integers**, as it uses other mechanisms, such as the **unsigned right shift operator (>>>)**, to handle unsigned operations.

Each integer type has a **fixed size and range**, ensuring consistent behavior across all platforms. The width of an integer type defines its **behavior and range of values**, rather than the amount of memory used internally by the Java runtime.

Name	Width	Range
long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	-2,147,483,648 to 2,147,483,647
short	16	-32,768 to 32,767
byte	8	-128 to 127

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Byte

The **byte** data type is the **smallest integer type** in Java. It is an **8-bit signed** type with a range of **-128 to 127** and is commonly used to store **raw binary data** or data from **files and networks**, helping save memory. For example, the following declares two **byte** variables called **b** and **c**:

```
byte b, c;
```

Short

short is a signed 16-bit type. It has a range from **-32,768 to 32,767**. It is probably the least-used Java type. Here are some examples of **short** variable declarations:

```
short s;  
short t;
```

int

int is a signed **32-bit** integer data type with a range of **-2,147,483,648 to 2,147,483,647**. It is the **most commonly used integer type** for variables, loops, and array indexing.

long

long is a signed **64-bit** integer data type used to store **very large whole numbers**. It is useful when the value exceeds the range of the **int** data type.

Floating-point types

These are used to store **real numbers with decimal (fractional) values** and support calculations requiring precision. Java provides two floating-point data types: **float (single precision)** and **double (double precision)**, based on the **IEEE 754** standard.

Name	Width in Bits	Approximate Range
double	64	4.9e-324 to 1.8e+308
float	32	1.4e-045 to 3.4e+038

float

float is a **32-bit single-precision** floating-point data type used to store decimal numbers. It requires less memory than double and is suitable when moderate precision is sufficient.

Here are some sample **float** variable declarations:

```
float hightemp, lowtemp;
```

double

double is a **64-bit double-precision** floating-point data type that provides greater accuracy for decimal calculations. It is commonly used in scientific and mathematical applications where high precision is required.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Here is a short program that uses **double** variables to compute the area of a circle:

```
// Compute the area of a circle.
class Area {
    public static void main(String[] args) {
        double pi, r, a;
        r = 10.8; // radius of circle
        pi = 3.1416; // pi, approximately
        a = pi * r * r; // compute area

        System.out.println("Area of circle is " + a);
    }
}
```

char (Character)

char is a **16-bit Unicode** data type used to store a single character. It supports characters from many languages, with a range of **0 to 65,535**, making Java suitable for international applications.

Here is a program that demonstrates **char** variables:

```
// Demonstrate char data type.
class CharDemo {
    public static void main(String[] args) {
        char ch1, ch2;

        ch1 = 88; // code for X
        ch2 = 'Y';

        System.out.print("ch1 and ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

This program displays the following output:

ch1 and ch2: X Y

boolean

boolean is a primitive data type that stores only two values: **true** or **false**. It is mainly used in **logical operations, relational comparisons, and decision-making statements** such as if, while, and for.

Here is a program that demonstrates the **boolean** type:

```
// Demonstrate boolean values.
class BoolTest {
    public static void main(String[] args) {
        boolean b;

        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);

        // a boolean value can control the if statement
        if(b) System.out.println("This is executed.");

        b = false;
        if(b) System.out.println("This is not executed.");
    }
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
// outcome of a relational operator is a boolean value
System.out.println("10 > 9 is " + (10 > 9));
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

II. VARIABLES

A variable is the basic unit of storage in Java that holds a value of a specific data type. It is defined by an identifier (name), data type, optional initial value, scope, and lifetime.

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

type identifier [= value][, identifier [= value] ...];

A variable declaration specifies the **data type**, **variable name (identifier)**, and an optional **initial value**. Multiple variables of the same type can be declared in a single statement using **commas**, and the assigned value must be compatible with the variable's data type.

Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c;           // declares three ints, a, b, and c.
int d = 3, e, f = 5;    // declares three more ints, initializing
                        // d and f.
byte z = 22;           // initializes z.
double pi = 3.14159;    // declares an approximation of pi.
char x = 'x';          // the variable x has the value 'x'.
```

An **identifier** is the **name given to a program element** such as a variable, method, or class.

Dynamic Initialization

Dynamic initialization is the process of assigning a variable a value using an **expression**, **method call**, or **another variable** at runtime instead of a constant.

Example:

```
double a = 3, b = 4;
double c = Math.sqrt(a * a + b * b);
```

Scope of Variables

The **scope** of a variable defines the region of the program where it can be accessed. A variable declared inside a block is accessible **only within that block**.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example:

```
if (true) {  
    int x = 10;  
    System.out.println(x); // Valid  
}  
// System.out.println(x); // Error: x is out of scope
```

Lifetime of Variables

The **lifetime** of a variable is the duration for which it exists in memory. A local variable is **created when its block starts** and **destroyed when the block ends**.

Example:

```
{  
    int y = 20;  
    System.out.println(y);  
} // y is destroyed here
```

III. TYPE CONVERSION AND CASTING

Type Conversion

Type conversion is the process of **converting a value from one data type to another**. Java supports both **automatic type conversion (widening)** and **explicit type conversion (casting)**.

Example:

```
int num = 100;  
long value = num; // Automatic type conversion
```

Automatic Type Conversion (Widening)

Java automatically converts a **smaller data type to a larger compatible data type** without data loss. This occurs only when the destination type is larger than the source type.

Example:

```
byte b = 10;  
int i = b; // byte → int  
double d = i; // int → double
```

Casting (Narrowing Conversion)

Casting is the **explicit conversion of a larger data type to a smaller data type**. It requires a cast operator and may result in **data loss**.

Example:

```
int i = 130;  
byte b = (byte) i; // Explicit casting
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
    public static void main(String[] args) {
        byte b;
        int i = 257;
        double d = 323.142;

        System.out.println("\nConversion of int to byte.");
        b = (byte) i;
        System.out.println("i and b " + i + " " + b);

        System.out.println("\nConversion of double to int.");
        i = (int) d;
        System.out.println("d and i " + d + " " + i);

        System.out.println("\nConversion of double to byte.");
        b = (byte) d;
        System.out.println("d and b " + d + " " + b);
    }
}
```

This program generates the following output:

```
Conversion of int to byte.
i and b 257 1
```

```
Conversion of double to int.
d and i 323.142 323
```

```
Conversion of double to byte.
d and b 323.142 67
```

Let's look at each conversion. When the value 257 is cast into a **byte** variable, the result is the remainder of the division of 257 by 256 (the range of a **byte**), which is 1 in this case. When the **d** is converted to an **int**, its fractional component is lost. When **d** is converted to a **byte**, its fractional component is lost, *and* the value is reduced modulo 256, which in this case is 67.

IV. AUTOMATIC TYPE PROMOTION IN EXPRESSIONS

Java automatically performs type conversion in expressions to ensure accurate calculations. Smaller data types are promoted to a larger compatible type when required.

For the given example:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Type Promotion

In Java, **byte, short, and char** values are automatically promoted to **int** when used in arithmetic expressions. Therefore, the expression $a * b / c$ is evaluated as an **int**, and the result is stored in the integer variable **d**.

Automatic Promotion and Casting

In arithmetic expressions, Java automatically promotes **byte, short, and char** to **int**. Therefore, the result of the expression cannot be assigned back to a byte without an explicit cast.

Without Casting (Compile-time Error):

```
byte b = 50;  
// b = b * 2; // Error: result is int
```

With Casting (Correct):

```
byte b = 50;  
b = (byte)(b * 2);  
System.out.println(b); // Output: 100
```

Here, $b * 2$ is promoted to **int**, so **casting** is required to store the result back into a byte.

Type Promotion Rules

Java automatically promotes data types in expressions to ensure accurate computation. Smaller types (**byte, short, and char**) are first promoted to **int**, and the entire expression is then promoted to the largest data type present.

Type Promotion Order:

- **byte, short, char** → **int**
- If one operand is **long** → result is **long**
- If one operand is **float** → result is **float**
- If one operand is **double** → result is **double**

Example:

```
byte b = 10;  
char c = 'A';  
short s = 20;  
int i = 30;  
float f = 40.5f;  
double d = 50.5;
```

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

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

Here, **b, c, and s** are first promoted to **int**. Since the expression contains a double value (**d**), the final result is automatically promoted to **double**.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

V. ARRAYS

An **array** is a collection of **elements of the same data type** stored under a single name. Each element is accessed using an **index**, making it easy to store and manage related data.

One-Dimensional Array

A **one-dimensional array** is a linear collection of elements of the same type. It is declared by specifying the **data type**, **array name**, and **square brackets ([])**.

Syntax:

```
data_type[] array_name;
```

Example:

```
int[] marks;
```

One-Dimensional Array

A **one-dimensional array** is a linear collection of elements of the same data type. It is declared using square brackets ([]) and memory is allocated using the new keyword.

Syntax:

```
int[] arr = new int[5];
```

Example:

```
int[] marks = {80, 85, 90, 95, 88};  
System.out.println(marks[2]); // Output: 90
```

Multidimensional Array

A **multidimensional array** is an array of arrays used to store data in rows and columns. The most common type is the **two-dimensional array**.

Example:

```
int[][] matrix = {  
    {1, 2, 3},  
    {4, 5, 6}  
};  
System.out.println(matrix[1][2]); // Output: 6
```

Alternative Array Declaration Syntax

Java also allows arrays to be declared by placing the brackets after the variable name. Both declaration styles are equivalent.

Example:

```
int[] a = new int[5];  
int b[] = new int[5];
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Local Variable Type Inference (var)

Java introduced the **var** keyword (JDK 10) to let the compiler automatically determine the data type of a local variable from its initializer.

Example:

```
var num = 100;    // int
var name = "Java"; // String
```

Restrictions of var

The **var** keyword can only be used for **local variables** and **must be initialized** during declaration. It cannot be used for class variables, method parameters, or with an array initializer.

Example:

```
var x = 10;        // Valid
```

```
// Invalid:
```

```
// var y;
```

```
// var arr = {1, 2, 3};
```

CHAPTER-3

OPERATORS

Operators are special symbols that perform operations on variables and values. Java provides different types of operators such as arithmetic, relational, logical, bitwise, assignment, and special operators to perform various operations in a program.

I. ARITHMETIC OPERATORS

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition (also unary plus)
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
- =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.

Basic Arithmetic Operators

Arithmetic operators are used to perform **mathematical calculations** on numeric values. The basic arithmetic operators in Java are + (**addition**), - (**subtraction**), * (**multiplication**), / (**division**), and % (**modulus**). Integer division returns only the **quotient**, while floating-point division returns the **decimal value**.

Example:

```
int a = 10, b = 3;
System.out.println(a + b); // 13
System.out.println(a - b); // 7
System.out.println(a * b); // 30
System.out.println(a / b); // 3
System.out.println(a % b); // 1
double x = 10.0, y = 3.0;
System.out.println(x / y); // 3.3333333333333335
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Output:

```
13
7
30
3
1
3.3333333333333335
```

Modulus Operator (%)

The **modulus operator (%)** returns the **remainder** after dividing one number by another. It can be used with both **integer** and **floating-point** data types.

Example:

```
int a = 10, b = 3;
System.out.println(a % b); // Output: 1
double x = 10.5, y = 3.0;
System.out.println(x % y); // Output: 1.5
```

Arithmetic Compound Assignment Operators

Compound assignment operators combine an **arithmetic operation** with **assignment**, making the code shorter and easier to read.

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

Example:

```
int a = 10;
a += 5; // a = a + 5
a *= 2; // a = a * 2
System.out.println(a); // Output: 30
```

Increment and Decrement Operators

The **increment (++)** operator increases a variable by **1**, while the **decrement (--)** operator decreases a variable by **1**. They can be used in **prefix** or **postfix** form.

Prefix Increment (++x)

The variable is incremented **before** its value is used.

Example:

```
int x = 10;
int y = ++x;
System.out.println(x); // 11
System.out.println(y); // 11
```

Postfix Increment (x++)

The current value is used **first**, and then the variable is incremented.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example:

```
int x = 10;  
int y = x++;  
System.out.println(x); // 11  
System.out.println(y); // 10
```

Prefix Decrement (--x)

The variable is decremented **before** its value is used.

Example:

```
int x = 10;  
int y = --x;
```

```
System.out.println(x); // 9  
System.out.println(y); // 9
```

Postfix Decrement (x--)

The current value is used **first**, and then the variable is decremented.

Example:

```
int x = 10;  
int y = x--;  
System.out.println(x); // 9  
System.out.println(y); // 10
```

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types: **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Bitwise operators perform operations **directly on the binary bits** of integer data types (byte, short, int, long). Java stores signed integers using the **two's complement** representation.

Binary Representation

All integer values are stored in **binary form**. Each bit represents a power of 2, and the value is obtained by adding the positions where the bits are 1.

Example:

$42 = 00101010_2$
 $= 32 + 8 + 2$
 $= 42$

Two's Complement Representation

Java represents **negative numbers** using the **two's complement** method. A negative number is obtained by **inverting all bits** of the positive number and **adding 1**.

Example:

$42 = 00101010$
Invert $\rightarrow 11010101$
 $+1 \rightarrow 11010110$ (-42)

Bitwise Logical Operators

Bitwise logical operators perform operations on the **individual bits** of integer values. The main bitwise logical operators are **& (AND)**, **| (OR)**, **^ (XOR)**, and **~ (NOT)**.

Bitwise NOT (~)

The **bitwise NOT (~)** operator inverts all the bits of its operand, changing **1s to 0s** and **0s to 1s**.

Example:

```
int a = 42;  
System.out.println(~a); // Output: -43
```

Bitwise AND (&)

The **AND (&)** operator returns **1** only if **both corresponding bits are 1**; otherwise, it returns **0**.

Example:

```
int a = 12, b = 10;  
System.out.println(a & b); // Output: 8
```

Bitwise OR (|)

The **OR (|)** operator returns **1** if **either of the corresponding bits is 1**.

Example:

```
int a = 12, b = 10;  
System.out.println(a | b); // Output: 14
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Bitwise XOR (^)

The **XOR (^)** operator returns **1** only if the corresponding bits are **different**.

Example:

```
int a = 12, b = 10;
```

```
System.out.println(a ^ b); // Output: 6
```

Left Shift Operator (<<)

The **left shift (<<)** operator shifts all bits to the **left** by a specified number of positions. Each left shift generally **multiplies the value by 2**.

Example:

```
int a = 5;
```

```
System.out.println(a << 1); // Output: 10
```

Right Shift Operator (>>)

The **right shift (>>)** operator shifts all bits to the **right** while preserving the sign bit. Each right shift generally **divides the value by 2**.

Example:

```
int a = 16;
```

```
System.out.println(a >> 2); // Output: 4
```

Unsigned Right Shift (>>>)

The **unsigned right shift (>>>)** operator shifts bits to the **right** and fills the leftmost bits with **0**, regardless of the sign.

Example:

```
int a = -1;
```

```
System.out.println(a >>> 24); // Output: 255
```

Bitwise Compound Assignment Operators

Bitwise compound assignment operators combine a **bitwise operation** with **assignment**, making the code shorter.

Operators: `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`

Example:

```
int a = 12;
```

```
a &= 10; // a = a & 10
```

```
System.out.println(a); // Output: 8
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

II. RELATIONAL OPERATORS

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements. Relational operators are used to **compare two values**. They return a **boolean value** (true or false) based on the comparison.

Relational Operators: ==, !=, >, <, >=, <=

Example:

```
int a = 10, b = 20;
System.out.println(a == b); // false
System.out.println(a != b); // true
System.out.println(a > b); // false
System.out.println(a < b); // true
System.out.println(a >= b); // false
System.out.println(a <= b); // true
```

Boolean Result of Relational Operators

The result of every relational operation is a **boolean** (true or false), which is commonly used in decision-making statements such as **if**, **while**, and **for**.

Example:

```
int a = 5, b = 10;
boolean result = (a < b);
System.out.println(result); // Output: true
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

III. BOOLEAN LOGICAL OPERATOR

The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	Result
&	Logical AND
	Logical OR
^	Logical XOR (exclusive OR)
	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
=	OR assignment
^=	XOR assignment

Operator	Result
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, **&**, **|**, and **^**, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state: **!true == false** and **!false == true**. The following table shows the effect of each logical operation:

A	B	A B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Logical operators are used to **perform logical operations on boolean values**. The main logical operators are **AND (&)**, **OR (|)**, **XOR (^)**, and **NOT (!)**, and they return either true or false.

Example:

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

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Output:

false
true
true
false
true

These operators follow the same logical rules as the corresponding bitwise operators, but they operate on **boolean values** instead of binary bits.

Short-Circuit Logical Operators (&& and ||)

Short-circuit logical operators **&&** (AND) and **||** (OR) evaluate the **second condition only if necessary**, improving efficiency and preventing runtime errors.

- **&&** stops if the first condition is **false**.
- **||** stops if the first condition is **true**.

Example:

```
int num = 100;  
int denom = 0;
```

```
if (denom != 0 && num / denom > 10)  
    System.out.println("Valid Division");  
else  
    System.out.println("Division by zero avoided");
```

Output:

Division by zero avoided

Here, `num / denom` is **not evaluated** because `denom != 0` is false, preventing a **division-by-zero** error.

IV. THE ASSIGNMENT OPERATOR

The **assignment operator (=)** is used to **assign a value or the result of an expression to a variable**. Java also supports **chained assignment**, where the same value is assigned to multiple variables in a single statement.

Example:

```
int x, y, z;  
x = y = z = 100;
```

```
System.out.println(x); // 100  
System.out.println(y); // 100  
System.out.println(z); // 100
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

V. THE ? OPERATOR

The **ternary operator** (`? :`) is a shorthand for the **if-else** statement. It evaluates a **boolean condition** and returns one of two values depending on whether the condition is **true** or **false**.

Syntax:

```
condition ? expression1 : expression2;
```

Example:

```
int num = -10;
int abs = (num < 0) ? -num : num;
```

```
System.out.println(abs); // Output: 10
```

Here, if `num` is negative, `-num` is returned; otherwise, `num` is returned.

VI. OPERATOR PRECEDENCE

Below table shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left). Although they are technically separators, the `[]`, `()`, and `.` can also act like operators. In that capacity, they would have the highest precedence. Also, notice the arrow operator (`->`). It is used in lambda expressions.

Highest						
<code>++</code> (postfix)	<code>--</code> (postfix)					
<code>++</code> (prefix)	<code>--</code> (prefix)	<code>~</code>	<code>!</code>	<code>+</code> (unary)	<code>-</code> (unary)	<code>(type-cast)</code>
<code>*</code>	<code>/</code>	<code>%</code>				
<code>+</code>	<code>=</code>					
<code>>></code>	<code>>>></code>	<code><<</code>				
<code>></code>	<code>>=</code>	<code><</code>	<code><=</code>	<code>instanceof</code>		
<code>==</code>	<code>!=</code>					
<code>&</code>						
<code>^</code>						
<code> </code>						
<code>&&</code>						
<code> </code>						
<code>?:</code>						
<code>-></code>						
<code>=</code>	<code>op=</code>					
Lowest						

The Precedence of the Java Operators

VII. USING PARENTHESES

Parentheses (`()`) are used to **change the order of evaluation** in an expression by giving higher precedence to the operations inside them. They also make complex expressions **easier to read and understand**.

Example:

```
int a = 16, b = 2;
System.out.println(a >> (b + 1)); // 2
System.out.println((a >> b) + 1); // 5
```

Here, parentheses change the order of evaluation, producing different results.

CHAPTER-4

CONTROL STATEMENTS

Program control statements determine the flow of execution in a Java program. They allow the program to make decisions, repeat statements, and transfer control to different parts of the program.

Java control statements are classified into three categories:

1. Selection Statements – Used for decision-making (if, if-else, switch).
2. Iteration Statements – Used to repeat a block of code (for, while, do-while).
3. Jump Statements – Used to alter the normal flow of execution (break, continue, return).

I. JAVA'S SELECTION STATEMENTS (IF, THE TRADITIONAL SWITCH)

Selection statements allow a program to **choose different execution paths** based on a condition. Java provides two selection statements: **if** and **switch**.

if Statement

The **if statement** is used to execute a block of code only when a specified **condition is true**. An optional **else** block executes when the condition is false.

Syntax:

```
if (condition) {  
    // statements  
}  
else {  
    // statements  
}
```

Example:

```
int a = 10, b = 20;
```

```
if (a < b)  
    System.out.println("a is smaller");  
else  
    System.out.println("b is smaller");
```

Nested if Statement

A **nested if** is an **if statement** inside another **if** or **else** statement. It is used to check multiple conditions.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example:

```
int x = 10, y = 20;
```

```
if (x > 0) {  
    if (y > 0)  
        System.out.println("Both are positive");  
}
```

if-else-if Ladder

The **if-else-if ladder** is used to test **multiple conditions**. The first condition that evaluates to **true** is executed, and the remaining conditions are skipped.

Example:

```
int marks = 75;
```

```
if (marks >= 90)  
    System.out.println("Grade A");  
else if (marks >= 75)  
    System.out.println("Grade B");  
else if (marks >= 50)  
    System.out.println("Grade C");  
else  
    System.out.println("Fail");
```

switch Statement

The **switch statement** is a **multi-way selection statement** that executes one block of code based on the value of an expression. It is often used as an alternative to multiple if-else-if statements.

Syntax:

```
switch(expression) {  
    case value1:  
        statements;  
        break;  
    case value2:  
        statements;  
        break;  
    default:  
        statements;  
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example:

```
int day = 2;
```

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

Nested switch Statement

A **nested switch** is a **switch statement** placed inside another switch statement. It is useful for handling multiple levels of selection.

Example:

```
int department = 1;  
int year = 2;  
switch(department) {  
    case 1:  
        switch(year) {  
            case 2:  
                System.out.println("CSE - Second Year");  
                break;  
        }  
        break;  
}
```

II. ITERATION STATEMENTS

Iteration statements are used to **execute a block of code repeatedly** until a specified condition becomes false. Java provides three looping statements: **while**, **do-while**, and **for**.

while Loop

The **while loop** executes a block of code **as long as the given condition is true**. The condition is checked **before** executing the loop body.

Syntax:

```
while(condition) {  
    // statements  
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example:

```
int i = 1;
```

```
while(i <= 5) {  
    System.out.println(i);  
    i++;  
}
```

Output:

```
1  
2  
3  
4  
5
```

do-while Loop

The **do-while loop** executes the loop body **at least once**, because the condition is checked **after** executing the statements.

Syntax:

```
do {  
    // statements  
} while(condition);
```

Example:

```
int i = 1;
```

```
do {  
    System.out.println(i);  
    i++;  
} while(i <= 5);
```

Output:

```
1  
2  
3  
4  
5
```

for Loop

The **for loop** is used when the **number of iterations is known**. It contains initialization, condition, and update expressions in a single statement.

Syntax:

```
for(initialization; condition; update) {  
    // statements  
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example:

```
for(int i = 1; i <= 5; i++) {  
    System.out.println(i);  
}
```

Output:

```
1  
2  
3  
4  
5
```

Loop Control Variable Inside for

The loop control variable can be **declared inside the for loop**, limiting its scope to the loop itself.

Example:

```
for(int i = 1; i <= 5; i++) {  
    System.out.println(i);  
}
```

Using Comma in for Loop

Java allows **multiple variables** to be initialized and updated using the **comma operator** in a for loop.

Example:

```
for(int i = 1, j = 5; i <= j; i++, j--) {  
    System.out.println(i + " " + j);  
}
```

Output:

```
1 5  
2 4  
3 3
```

Variations of for Loop

The initialization, condition, or update expressions in a for loop can be omitted. An empty for(;;) creates an **infinite loop**.

Example:

```
int i = 1;
```

```
for( i <= 5; ) {  
    System.out.println(i);  
    i++;  
}
```

Enhanced for Loop (For-Each Loop)

The **enhanced for loop** is used to **iterate through arrays or collections** without using an index.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Syntax:

```
for(dataType variable : array) {  
    // statements  
}
```

Example:

```
int[] nums = {10, 20, 30, 40};
```

```
for(int n : nums) {  
    System.out.println(n);  
}
```

Output:

```
10  
20  
30  
40
```

Enhanced for Loop with Multidimensional Arrays

The enhanced for loop can also **iterate through multidimensional arrays** using nested loops.

Example:

```
int[][] arr = {{1,2},{3,4}};
```

```
for(int[] row : arr) {  
    for(int value : row) {  
        System.out.print(value + " ");  
    }  
    System.out.println();  
}
```

Output:

```
1 2  
3 4
```

Local Variable Type Inference (var) in for Loop

The **var** keyword (JDK 10+) lets the compiler infer the type of the loop variable automatically.

Example:

```
for(var i = 1; i <= 5; i++) {  
    System.out.println(i);  
}
```

Nested Loops

A **nested loop** is a loop inside another loop. The inner loop completes all its iterations for each iteration of the outer loop.

Example:

```
for(int i = 1; i <= 3; i++) {  
    for(int j = 1; j <= 3; j++) {  
        System.out.print(j + " ");  
    }  
    System.out.println();  
}
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Output:

1 2 3
1 2 3
1 2 3

III. JUMP STATEMENTS

Jump statements transfer program control to another part of the program. Java provides three jump statements: break, continue, and return.

1. **break**

The break statement immediately terminates a loop or exits a switch statement. It can also be used with labels to exit nested loops.

Example:

```
for(int i=1; i<=10; i++) {  
    if(i==5)  
        break;  
    System.out.println(i);  
}
```

Output:

1
2
3
4

2. **continue**

The continue statement skips the current iteration of a loop and continues with the next iteration.

Example:

```
for(int i=1; i<=5; i++) {  
    if(i==3)  
        continue;  
    System.out.println(i);  
}
```

Output:

1
2
4
5
3.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

return

The return statement terminates a method and optionally returns a value to the caller.

Example:

```
public class Demo {  
    public static void main(String args[]) {  
        System.out.println("Before return");  
        return;  
        // System.out.println("After return"); // Unreachable  
    }  
}
```

Output: Before return

