

JAVA PROGRAMMING

Chapter 1

Introduction to Java

Study Notes for Engineering / Diploma Students

Includes: Detailed Notes • Tables • Code Examples • 15 MCQs with Answer Key • 10 Broad Questions

Contents

1.1 Fundamentals of Object Oriented Programming

Objects and Classes, Data Abstraction and Encapsulation, Inheritance, Polymorphism, Dynamic Binding

1.2 Java Features

Compiled and Interpreted, Platform Independent and Portable, Object Oriented, Distributed, Multithreaded and Interactive, High Performance

1.3 Constants, Variables and Data Types

Constant, Data Types, Scope of Variable, Symbolic Constant, Type Casting, Standard Default Values

1.4 Operators and Expressions

Arithmetic, Relational, Logical, Assignment, Increment/Decrement, Conditional, Bitwise, Special Operators

1.5 Decision Making and Branching

Simple if, if-else, else-if Ladder, switch, the ? : Operator

1.6 Decision Making and Looping

while, do-while, for, Jumps in Loops, Labeled Loops

Practice Section

15 Multiple Choice Questions, Answer Key, 10 Broad Questions

1.1 Fundamentals of Object Oriented Programming

Object Oriented Programming (OOP) is a programming paradigm in which a program is organised around **objects** (data together with the functions that operate on that data) rather than around functions and logic alone. Java is built on this paradigm, so understanding the basic OOP ideas is the first step toward writing good Java programs. OOP models real-world entities such as a student, a bank account or a vehicle, and makes large programs easier to design, extend and maintain.

1.1.1 Procedure Oriented vs Object Oriented Programming

Earlier languages such as C follow the *procedure oriented* approach, where a program is a list of instructions divided into functions and data is largely global. OOP treats data as a critical element and binds it tightly to the functions that use it.

Table 1.1: Comparison of POP and OOP

Basis	Procedure Oriented (POP)	Object Oriented (OOP)
Approach	Top-down	Bottom-up
Main focus	Functions / procedures	Objects and data
Data security	Data is global and freely accessible; less secure	Data is hidden inside objects; more secure
Code reuse	Limited	High, through inheritance and classes
Modification	Difficult in large programs	Easy, because of modularity
Real-world modelling	Poor	Good
Examples	C, Pascal, FORTRAN	Java, C++, Python, C#

1.1.2 Basic Concepts of OOP

- **Objects and Classes**
- **Data abstraction and Encapsulation**
- **Inheritance**
- **Polymorphism**
- **Dynamic binding**
- Message passing (objects communicate by calling each other's methods)

1.1.3 Objects and Classes

Object

An **object** is a real-world entity or a run-time instance that has *state* (data), *behaviour* (actions) and *identity* (a unique reference). For example, a *Student* object has state such as name and roll number, and behaviour such as *study()* and *writeExam()*. Objects occupy memory when they are created at run time.

Class

A **class** is a blueprint or template from which objects are created. It defines the variables (also called fields or attributes) and methods that all its objects will have. A class by itself does not occupy memory for data; memory is allocated only when an object is created using the **new** keyword. A class is a user-defined data type, and the relationship is: *class = type, object = variable of that type*.

Table 1.2: Difference between Class and Object

Class	Object
Blueprint / template / logical entity	Instance of a class / physical entity
Declared once using the class keyword	Created many times using the new keyword
No memory allocated for instance data at declaration	Memory allocated when created
Example: Car	Example: myCar, yourCar

Example 1.1: A class and its object

```
class Student {
    int roll;           // instance variable
    String name;

    void display() {    // method
        System.out.println(roll + " " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(); // object creation
        s1.roll = 101;
        s1.name = "Amit";
        s1.display();              // output: 101 Amit
    }
}
```

1.1.4 Data Abstraction and Encapsulation

Data Abstraction

Abstraction means showing only the essential features of an object and hiding the unnecessary implementation details. When you drive a car you use the steering wheel, accelerator and brake without knowing how the engine works internally. In Java, abstraction is achieved using **abstract classes** and **interfaces**.

Encapsulation

Encapsulation is the wrapping up of data (variables) and the code (methods) that operates on the data into a single unit called a class. It also restricts direct access to the data by declaring variables **private** and providing public *getter* and *setter* methods. This mechanism is also known as **data hiding**.

- Protects data from unauthorised or accidental modification.
- Gives the programmer full control over how data is accessed (validation inside setters).
- Makes code flexible, maintainable and easier to test.

- A fully encapsulated class has all private fields with public getters and setters.

Example 1.2: Encapsulation using private data

```
class Account {
    private double balance;           // hidden data

    public void setBalance(double b) { // setter
        if (b >= 0) balance = b;
    }
    public double getBalance() {      // getter
        return balance;
    }
}
```

Table 1.3: Abstraction vs Encapsulation

Abstraction	Encapsulation
Hides implementation complexity	Hides data and binds it with methods
Solves the problem at design level	Solves the problem at implementation level
Achieved by abstract classes and interfaces	Achieved by classes and access modifiers (private, etc.)
Focus: what an object does	Focus: how data is protected

1.1.5 Inheritance

Inheritance is the mechanism by which one class (the *subclass*, *derived class* or *child*) acquires the properties and methods of another class (the *superclass*, *base class* or *parent*). It represents an **IS-A** relationship (a Dog *is an* Animal) and is the main way to achieve **code reusability**. In Java, inheritance is implemented using the **extends** keyword.

Example 1.3: Inheritance

```
class Animal {
    void eat() { System.out.println("Eating..."); }
}
class Dog extends Animal {
    void bark() { System.out.println("Barking..."); }
}
public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();    // inherited method
        d.bark();
    }
}
```

Table 1.4: Types of inheritance

Type	Description	Supported in Java?
Single	One subclass inherits from one superclass (A to B)	Yes
Multilevel	A chain of inheritance (A to B to C)	Yes
Hierarchical	Many subclasses inherit from one superclass	Yes
Multiple	One subclass has more than one superclass	No, with classes (avoids ambiguity); possible through interfaces
Hybrid	Combination of two or more types	Only through interfaces

Advantages: reusability of code, reduced development time, method overriding for run-time polymorphism, and a natural hierarchical classification of real-world objects.

1.1.6 Polymorphism

The word *polymorphism* comes from Greek and means '**many forms**'. It is the ability of a single entity (a method or an operator) to behave differently in different situations. Java supports two kinds:

Table 1.5: Types of polymorphism

Feature	Compile-time (Static) Polymorphism	Run-time (Dynamic) Polymorphism
Also called	Early binding, method overloading	Late binding, method overriding
Resolved by	Compiler	JVM at run time
How achieved	Same method name with different parameter lists in the same class	Subclass redefines a method of its superclass with the same signature
Needs inheritance?	No	Yes
Speed	Faster	Slightly slower

Example 1.4: Method overloading (compile-time polymorphism)

```
class Calc {
    int add(int a, int b) { return a + b; }
    double add(double a, double b){ return a + b; } // overloading
    int add(int a, int b, int c) { return a + b + c; }
}
```

1.1.7 Dynamic Binding

Binding means linking a method call to the actual method body. In **static (early) binding** the link is made at compile time (for *private*, *static* and *final* methods and overloaded methods). In **dynamic (late) binding** the link is made at *run time* according to the actual type of the object, not the type of the reference variable. Dynamic binding is what makes method overriding and run-time polymorphism work.

Example 1.5: Dynamic binding

```
class Shape { void draw() { System.out.println("Drawing shape"); } }
class Circle extends Shape {
    void draw() { System.out.println("Drawing circle"); } // overriding
}
public class Demo {
    public static void main(String[] args) {
        Shape s = new Circle(); // superclass reference, subclass object
        s.draw();                // output: Drawing circle (decided at run time)
    }
}
```

Table 1.6: Static vs Dynamic binding

Static Binding	Dynamic Binding
Happens at compile time	Happens at run time
Uses the type of the reference	Uses the type of the actual object
Applies to private, static, final methods and overloading	Applies to overridden (virtual) methods
Faster execution	Provides flexibility

1.1.8 Benefits of OOP

- Reuse of existing classes through inheritance eliminates redundant code.
- Data hiding helps build secure programs.
- Multiple objects of a class can coexist without interference.
- Real-world problems map naturally to objects; easy to partition work in a project.
- Software complexity is easier to manage; upgrading from small to large systems is easy.

1.2 Java Features

1.2.1 Introduction and History

Java was developed by **James Gosling** and his team (Mike Sheridan, Patrick Naughton) at **Sun Microsystems**. The project began in 1991 under the name *Green* and the language was first called *Oak* (after an oak tree outside Gosling's office) and later renamed **Java** (after Java coffee). The first public release, Java 1.0, came in **1995**. Sun Microsystems was acquired by Oracle Corporation in 2010, which now maintains Java. The language was designed to be small, simple, secure and, most importantly, **portable across different hardware and operating systems**.

1.2.2 Compiled and Interpreted

Most languages are either compiled (C, C++) or interpreted (Python, JavaScript). Java is unusual because it uses **both**. First, the Java compiler **javac** translates the source code (.java file) into an intermediate, machine-independent code called **bytecode** (.class file). Then the **Java Virtual Machine (JVM)** interprets (and, using the JIT compiler, compiles) the bytecode into machine code of the host computer and executes it.

Figure 1.1: Java compilation and execution process

Source Code (Hello.java)	javac ----> Compiler	Bytecode (Hello.class)	JVM ----> Interpreter/JIT	Output (Windows / Linux / macOS)
-----------------------------	-------------------------	---------------------------	------------------------------	-------------------------------------

Example 1.6: First Java program

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}
// Compile : javac Hello.java
// Run      : java Hello
```

Table 2.1: JDK, JRE, JVM and JIT

Term	Full Form	Role
JDK	Java Development Kit	Complete kit for developing Java programs: JRE + compiler (javac), debugger and tools
JRE	Java Runtime Environment	Provides libraries and JVM needed to run Java programs
JVM	Java Virtual Machine	Loads, verifies and executes bytecode; provides memory management and garbage collection
JIT	Just-In-Time Compiler	Part of JVM; converts frequently used bytecode to native code to improve speed

1.2.3 Platform Independent and Portable

Java follows the principle '**Write Once, Run Anywhere**' (**WORA**). Bytecode is not tied to any processor or operating system; it runs on any machine that has a JVM. Only the JVM is platform specific, not the Java program. Java is also **portable** because the size of primitive data types is fixed (an int is always 32 bits) on every platform, unlike C where it may vary, and because the Java libraries behave the same everywhere. A program developed on Windows can be copied to Linux or macOS and executed without any change.

1.2.4 Object Oriented

Java is a (nearly) pure object oriented language. Everything except the eight primitive types is an object, and all code is written inside classes. It supports all OOP principles: encapsulation, inheritance, polymorphism and abstraction. The Java class library (API) provides a large set of ready-made classes that can be extended and reused.

1.2.5 Distributed

Java is designed for the networked, internet environment. It has a rich set of classes in the *java.net* package for working with TCP/IP protocols and URLs, so accessing remote resources is as easy as accessing local files. Technologies such as **RMI (Remote Method Invocation)** and CORBA allow objects on different computers to interact, making it possible to build distributed applications.

1.2.6 Multithreaded and Interactive

A **thread** is a lightweight, independent path of execution within a program. **Multithreading** allows a program to perform many tasks simultaneously, for example downloading a file while playing music and accepting user input. Java provides built-in support through the *Thread* class and *Runnable* interface, along with synchronisation features to keep shared data safe. Multithreading gives better CPU utilisation and makes Java programs **interactive** and responsive, which is essential for GUI, games and animation.

1.2.7 High Performance

Although Java is interpreted, its performance is high because bytecode is compact and close to machine code, the **JIT compiler** converts hot spots into native code at run time, and automatic **garbage collection** reclaims unused memory efficiently. Java is slower than C/C++ for some tasks but fast enough for most enterprise, web and mobile applications.

1.2.8 Other Important Features

Table 2.2: Additional features of Java

Feature	Explanation
Simple	Syntax is similar to C/C++, but removes confusing features such as pointers, operator overloading, header files and multiple inheritance of classes
Secure	No explicit pointers; bytecode verifier and security manager; programs run inside the JVM sandbox
Robust	Strong type checking, exception handling and automatic garbage collection reduce errors and crashes
Dynamic	Classes are loaded at run time as needed; can adapt to an evolving environment
Architecture neutral	Bytecode does not depend on processor architecture

1.3 Constants, Variables and Data Types

1.3.1 Java Character Set, Keywords and Identifiers

A Java program is built from tokens: **keywords** (reserved words such as `class`, `int`, `if`, `for` with fixed meaning), **identifiers** (names of variables, methods and classes), **literals**, **operators** and **separators**. Java uses the Unicode character set, which supports characters from almost all world languages.

Rules for naming identifiers

- May contain letters, digits, underscore (`_`) and dollar sign (`$`).
- Must not begin with a digit.
- Must not be a Java keyword (for example, `int`, `class`).
- Are case sensitive: *total* and *Total* are different names.
- No spaces or other special symbols are allowed.
- Convention: classes start with uppercase (*StudentInfo*), variables and methods in camelCase (*studentName*), constants in UPPERCASE (*MAX_SIZE*).

1.3.2 Constants

A **constant** is a fixed value that does not change during program execution. Constants in Java are also called **literals**.

Table 3.1: Types of constants

Type of Constant	Description	Examples
Integer	Whole numbers: decimal, octal (prefix 0), hexadecimal (prefix 0x), binary (prefix 0b); long ends with L	25, 017, 0x1F, 0b101, 500L
Real (floating point)	Numbers with a decimal point or exponent; float ends with f	3.14, 0.5e3, 2.5f
Character	A single character in single quotes, or escape sequence	'A', '7', '\n', '\t'
String	A sequence of characters in double quotes	"Java", "Hello World"
Boolean	Logical values	true, false

Common escape sequences

Table 3.2: Backslash character constants

Sequence	Meaning	Sequence	Meaning
\n	New line	\\	Backslash
\t	Horizontal tab	\'	Single quote
\b	Backspace	\"	Double quote
\r	Carriage return	\0	Null character

1.3.3 Variables

A **variable** is a named memory location whose value can change during execution. Every variable must be *declared* with a data type before use. Syntax: **datatype variableName = value;**

Example 1.7: Declaring variables

```
int age = 20;           // declaration with initialisation
float marks;           // declaration only
marks = 85.5f;         // assignment later
int a = 5, b = 10, c;   // multiple variables of same type
```

1.3.4 Data Types

A data type specifies the size and kind of values a variable can hold. Java is a **strongly typed** language. Data types are of two categories: **primitive** (built-in) and **non-primitive** (reference or derived).

A. Primitive data types (8 types)

Table 3.3: Primitive data types in Java

Type	Size	Range / Values	Example
byte	1 byte (8 bits)	-128 to 127	byte b = 100;
short	2 bytes (16 bits)	-32,768 to 32,767	short s = 5000;
int	4 bytes (32 bits)	-2,147,483,648 to 2,147,483,647	int i = 100000;
long	8 bytes (64 bits)	-2^{63} to $2^{63}-1$	long l = 9999999999L;
float	4 bytes (32 bits)	About 6-7 decimal digits precision	float f = 3.5f;
double	8 bytes (64 bits)	About 15 decimal digits precision	double d = 3.14159;
char	2 bytes (16 bits)	0 to 65,535 (Unicode characters)	char c = 'A';
boolean	1 bit (JVM dependent)	true or false	boolean f = true;

Integer types: byte, short, int, long. **Floating-point types:** float, double. **Character type:** char. **Logical type:** boolean. The default type of an integer literal is *int* and the default type of a real literal is *double*.

B. Non-primitive (reference) data types

These types refer to objects and are created by the programmer or provided by the Java library. They include **String**, **Arrays**, **Classes**, **Interfaces** and **Enums**. A reference variable stores the address of the object, not the object itself, and its default value is *null*.

Table 3.4: Primitive vs Non-primitive types

Primitive	Non-primitive
Predefined by the language	Created by programmer (except String)
Stores the actual value	Stores the reference (address) of an object
Always starts with lowercase name (int)	Starts with uppercase name (String)
Cannot be null	Can be null
Fixed size	Size is not fixed

1.3.5 Scope of Variable

The **scope** of a variable is the region of the program where the variable can be accessed. Its **lifetime** is the period during which it exists in memory.

Table 3.5: Kinds of variables and their scope

Variable Kind	Declared	Scope / Lifetime	Default value?
Local variable	Inside a method, constructor or block	Only within that method/block; destroyed when it ends	No, must be initialised before use
Instance variable	Inside a class, outside methods, without static	Throughout the class via object; exists as long as object exists	Yes
Static (class) variable	Inside a class with the static keyword	Shared by all objects; exists for the whole program run	Yes
Block variable	Inside braces { } of loop or if	Only inside that block	No

Example 1.8: Scope of variables

```
class ScopeDemo {
    static int count = 0;    // static variable
    int id = 1;             // instance variable

    void show() {
        int x = 10;         // local variable
        for (int i = 0; i < 3; i++) { // i is a block variable
            System.out.println(i);
        }
        // System.out.println(i); // ERROR: i is out of scope
    }
}
```

1.3.6 Symbolic Constant

A **symbolic constant** is a named constant whose value cannot be changed after assignment. It is declared with the **final** keyword. Using symbolic constants improves readability (use PI instead of 3.14159), makes modification easy (change the value at one place only) and avoids accidental changes. By convention their names are written in uppercase.

Example 1.9: Symbolic constants

```
final double PI = 3.14159;
final int MAX_MARKS = 100;
// PI = 3.14; // ERROR: cannot assign a value to final variable

static final int DAYS_IN_WEEK = 7; // class-level constant
```

1.3.7 Type Casting (Type Conversion)

Type casting is converting a value of one data type into another. Java has two kinds.

(a) Widening (implicit / automatic) conversion

A smaller type is converted to a larger type automatically, with no loss of data. Order: **byte** > **short** > **int** > **long** > **float** > **double** (char can also widen to int).

(b) Narrowing (explicit) conversion

A larger type is converted to a smaller type manually by writing the target type in parentheses. Data may be lost (fraction dropped or value truncated).

Example 1.10: Type casting

```
int i = 100;
double d = i;           // widening -> 100.0

double pi = 9.78;
int n = (int) pi;       // narrowing -> 9 (fraction lost)

int big = 130;
byte b = (byte) big;    // result: -126 (overflow)

char ch = 'A';
int code = ch;          // widening -> 65
```

Table 3.6: Widening vs Narrowing

Widening Conversion	Narrowing Conversion
Small type to large type	Large type to small type
Automatic, done by compiler	Manual, uses cast operator (type)
No loss of information	Possible loss of data or precision
Safe	Needs care

Type promotion in expressions: in an expression, byte, short and char operands are promoted to *int*; if any operand is long, float or double, the whole expression is promoted to that type. Hence *byte a = 10, b = 20; byte c = a + b;* gives an error, and must be written as *byte c = (byte)(a + b);*.

1.3.8 Standard Default Values

Instance and static variables are automatically given a default value if not initialised. **Local variables do not get default values;** using them without initialisation causes a compile-time error.

Table 3.7: Default values of Java data types

Data Type	Default Value
byte, short, int	0
long	0L
float	0.0f
double	0.0d
char	"\u0000" (null character)
boolean	false
Reference types (String, arrays, objects)	null

1.4 Operators and Expressions

An **operator** is a symbol that tells the compiler to perform a specific mathematical, logical or relational operation on **operands**. An **expression** is a combination of operands and operators that evaluates to a single value, for example $a + b * 2$. Based on the number of operands, operators are **unary** (one operand), **binary** (two) and **ternary** (three).

1.4.1 Arithmetic Operators

Table 4.1: Arithmetic operators

Operator	Meaning	Example (a = 10, b = 3)	Result
+	Addition (also string concatenation)	$a + b$	13
-	Subtraction	$a - b$	7
*	Multiplication	$a * b$	30
/	Division (integer division drops fraction)	a / b	3
%	Modulus (remainder)	$a \% b$	1

Note: dividing two integers gives an integer ($10/4 = 2$), while $10/4.0 = 2.5$. Division of an integer by zero throws *ArithmeticException*, but a floating-point division by zero gives Infinity. The % operator also works on floating-point values in Java ($10.5 \% 3 = 1.5$).

1.4.2 Relational Operators

Relational operators compare two values and produce a **boolean** result (true or false).

Table 4.2: Relational operators

Operator	Meaning	Example (a = 10, b = 3)	Result
<	Less than	$a < b$	false
>	Greater than	$a > b$	true
<=	Less than or equal to	$a <= 10$	true
>=	Greater than or equal to	$b >= a$	false
==	Equal to	$a == b$	false
!=	Not equal to	$a != b$	true

1.4.3 Logical Operators

Table 4.3: Logical operators

Operator	Name	Description	Example
&&	Logical AND	True only if both conditions are true	$(a > 5 \ \&\& \ b < 5)$ gives true
	Logical OR	True if at least one condition is true	$(a < 5 \ \ b < 5)$ gives true
!	Logical NOT	Reverses the result	$!(a > b)$ gives false

Table 4.4: Truth table

A	B	A && B	A B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Short-circuit evaluation: with &&, if the left operand is false the right operand is not evaluated; with ||, if the left operand is true the right operand is skipped. The non-short-circuit versions & and | always evaluate both sides.

1.4.4 Assignment Operators

The simple assignment operator = stores the value of the right-hand expression in the left-hand variable. Compound (shorthand) operators combine an operation with assignment.

Table 4.5: Assignment operators

Operator	Example	Equivalent to
=	a = 5	Assigns 5 to a
+=	a += 3	a = a + 3
-=	a -= 3	a = a - 3
*=	a *= 3	a = a * 3
/=	a /= 3	a = a / 3
%=	a %= 3	a = a % 3
&=, =, ^=, <<=, >>=	a <<= 2	a = a << 2 (and so on)

1.4.5 Increment and Decrement Operators

++ increases and -- decreases the value of a variable by 1. They are unary operators and have two forms: **prefix** (operator before variable: the value is changed first, then used) and **postfix** (operator after variable: the value is used first, then changed).

Table 4.6: Prefix and postfix operators

Expression	Meaning	If a = 5, result
b = ++a	Prefix: increment a, then assign	a = 6, b = 6
b = a++	Postfix: assign, then increment a	b = 5, a = 6
b = --a	Prefix: decrement a, then assign	a = 4, b = 4
b = a--	Postfix: assign, then decrement a	b = 5, a = 4

Example 1.11: Increment operator in an expression

```
int a = 5;
int y = a++ + ++a;    // a++ gives 5 (a becomes 6); ++a makes a 7 and gives 7
System.out.println(y);    // 12
System.out.println(a);    // 7
```

1.4.6 Conditional (Ternary) Operator

The **? :** operator is the only ternary operator in Java. Syntax: **condition ? value1 : value2**; If the condition is true, value1 is the result; otherwise value2. It is a compact replacement for a simple if-else.

Example 1.12: Ternary operator

```
int a = 10, b = 20;
int max = (a > b) ? a : b;    // max = 20
String r = (marks >= 40) ? "Pass" : "Fail";
```

1.4.7 Bitwise Operators

Bitwise operators work on the individual bits of integer operands (byte, short, int, long, char). Consider $a = 5$ (0101) and $b = 3$ (0011).

Table 4.7: Bitwise operators

Operator	Name	Rule	Example	Result
&	Bitwise AND	1 if both bits are 1	5 & 3 (0101 & 0011)	1 (0001)
	Bitwise OR	1 if any bit is 1	5 3 (0101 0011)	7 (0111)
^	Bitwise XOR	1 if bits are different	5 ^ 3 (0101 ^ 0011)	6 (0110)
~	Bitwise complement	Inverts all bits (= -(n+1))	~5	-6
<<	Left shift	Shifts bits left, fills 0; multiplies by 2^n	5 << 1	10
>>	Right shift (signed)	Shifts right, keeps sign; divides by 2^n	-8 >> 1	-4
>>>	Unsigned right shift	Shifts right, fills 0 on the left	8 >>> 1	4

1.4.8 Special Operators

Table 4.8: Special operators

Operator	Purpose	Example
instanceof	Tests whether an object is an instance of a given class; returns boolean	s instanceof String
Dot (.) operator	Accesses members (variables and methods) of an object or class	obj.display(); System.out.println()
new	Creates an object or array and allocates memory	Student s = new Student();
Cast (type)	Converts one type to another	(int) 5.7
Array index []	Accesses an element of an array	marks[2]
Parentheses ()	Groups expressions, forces precedence, calls methods	(a + b) * c

1.4.9 Operator Precedence and Associativity

When an expression has several operators, **precedence** decides which is evaluated first, and **associativity** decides the order among operators of equal precedence. Parentheses always override precedence.

Table 4.9: Precedence of operators

Level (high to low)	Operators	Associativity
1	() [] .	Left to right
2	++ -- (postfix)	Left to right
3	++ -- (prefix), unary + -, !, ~, (type) cast, new	Right to left
4	* / %	Left to right
5	+ -	Left to right
6	<< >> >>>	Left to right
7	< <= > >= instanceof	Left to right
8	== !=	Left to right
9	& then ^ then	Left to right
10	&& then	Left to right
11	? :	Right to left
12	= += -= *= /= %= etc.	Right to left

Example: $x = 10 + 5 * 2 - 8 / 4$; gives $5 * 2 = 10$, $8 / 4 = 2$, then $10 + 10 - 2 = 18$.

1.5 Decision Making and Branching

Normally statements execute one after another (sequentially). **Decision making** (selection) statements let a program choose different paths depending on a condition, and **branching** transfers control to a particular block of code. Java provides the **if** family, the **switch** statement and the **? :** operator. The condition in Java must be a *boolean* expression (unlike C, 0 and 1 are not allowed).

1.5.1 Simple if Statement

Executes a block only when the condition is true; otherwise the block is skipped.

Syntax and Example 1.13

```
if (condition) {
    // statements executed when condition is true
}

// Example
int age = 20;
if (age >= 18) {
    System.out.println("Eligible to vote");
}
```

1.5.2 The if-else Statement

Provides two paths: one block when the condition is true and another when it is false.

Syntax and Example 1.14

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

// Example
int num = 7;
if (num % 2 == 0)
    System.out.println("Even");
else
    System.out.println("Odd");
```

1.5.3 Nested if Statement

An if statement placed inside another if or else block is called a nested if. It is used when a decision depends on a previous decision.

Example 1.15: Largest of three numbers using nested if

```
if (a > b) {
    if (a > c)
        System.out.println("a is largest");
    else
        System.out.println("c is largest");
} else {
    if (b > c)
        System.out.println("b is largest");
    else
        System.out.println("c is largest");
}
```

1.5.4 The else-if Ladder

Used when there are many alternatives. Conditions are tested from the top; the first true condition's block is executed and the rest of the ladder is skipped. If none is true, the final *else* block runs.

Syntax idea and Example 1.16: Grade calculation

```
if (marks >= 75)      grade = 'A';
else if (marks >= 60) grade = 'B';
else if (marks >= 40) grade = 'C';
else                 grade = 'F';
```

1.5.5 The switch Statement

The **switch** statement selects one of many blocks by comparing the value of an expression with a list of **case** constants. It is a cleaner alternative to a long else-if ladder when a single variable is compared with fixed values.

Syntax and Example 1.17

```
switch (expression) {
    case value1:
        // statements
        break;
    case value2:
        // statements
        break;
    default:
        // statements when no case matches
}

// Example
int day = 3;
switch (day) {
    case 1: System.out.println("Mon"); break;
    case 2: System.out.println("Tue"); break;
    case 3: System.out.println("Wed"); break;
    default: System.out.println("Invalid day");
} // output: Wed
```

Rules of switch

- The expression must be of type byte, short, int, char, String (Java 7 onwards) or an enum. **long, float, double and boolean are not allowed.**

- Case values must be constants (literals or final variables) and must be unique.
- **break** ends the switch; without it execution *falls through* to the next case.
- **default** is optional, can appear anywhere, and runs when no case matches.
- Switch checks only for equality, while if can test ranges and complex conditions.

Table 5.1: if-else vs switch

if-else	switch
Can test any boolean expression, ranges and conditions	Tests a single expression for equality with constants
Works with all data types	Works with byte, short, int, char, String, enum
Slower when there are many conditions	Faster and more readable for many fixed choices
No break needed	break is needed to avoid fall-through

1.5.6 The ? : Operator

The conditional operator is a shorthand form of if-else that returns a value. Syntax: **variable = (condition) ? expression1 : expression2;** It is best for short, simple decisions, and it can be nested, although readability suffers.

Example 1.18: ? : operator

```
int a = 15, b = 9;
String msg = (a % 2 == 0) ? "Even" : "Odd";      // Odd
int small = (a < b) ? a : b;                      // 9

// Nested: largest of three
int big = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);
```

1.6 Decision Making and Looping

A **loop** repeatedly executes a block of statements as long as a condition holds. Every loop has four parts: **initialisation** (set the loop variable), **condition** (test), **body** (the statements to repeat) and **update** (increment or decrement). Loops are of two types: **entry-controlled** (condition tested before the body: while, for) and **exit-controlled** (condition tested after the body: do-while).

1.6.1 The while Statement

An entry-controlled loop. The condition is checked first; if true the body executes, and the process repeats. If the condition is false at the start, the body never executes.

Syntax and Example 1.19

```
initialisation;
while (condition) {
    // body
    update;
}

// Example: sum of first 5 numbers
int i = 1, sum = 0;
while (i <= 5) {
    sum += i;
    i++;
}
System.out.println(sum);    // 15
```

1.6.2 The do-while Statement

An exit-controlled loop. The body is executed first and the condition is tested afterwards, so the body runs **at least once** even if the condition is false. Note the semicolon after while(condition).

Syntax and Example 1.20

```
do {
    // body
    update;
} while (condition);

// Example
int n = 1;
do {
    System.out.println(n);
    n++;
} while (n <= 3);           // prints 1 2 3
```

1.6.3 The for Statement

The most compact entry-controlled loop, used when the number of repetitions is known. It has initialisation, condition and update in one line.

Syntax and Example 1.21

```
for (initialisation; condition; update) {  
    // body  
}  
  
// Example: multiplication table of 5  
for (int i = 1; i <= 10; i++) {  
    System.out.println("5 x " + i + " = " + (5 * i));  
}  
  
// Enhanced for loop (for-each) to traverse arrays  
int[] arr = {10, 20, 30};  
for (int x : arr) {  
    System.out.println(x);  
}
```

Notes: any of the three parts of for may be omitted; *for(;;)* is an infinite loop. Several variables can be initialised or updated using commas, for example *for(int i = 0, j = 10; i < j; i++, j--)*. Nested loops (a loop inside a loop) are used for patterns and matrices.

Example 1.22: Nested for loop (pattern)

```
for (int i = 1; i <= 3; i++) {           // outer loop: rows  
    for (int j = 1; j <= i; j++) {       // inner loop: columns  
        System.out.print("* ");  
    }  
    System.out.println();  
}  
// Output:  
// *  
// * *  
// * * *
```

Table 6.1: Comparison of loop statements

Feature	while	do-while	for
Type	Entry-controlled	Exit-controlled	Entry-controlled
Condition checked	Before body	After body	Before body
Minimum executions	0	1	0
Semicolon at end	No	Yes	No
Best used when	Repetitions unknown	Body must run at least once (menus)	Repetitions known

1.6.4 Jumps in Loops

Jump statements transfer control from one part of the program to another and change the normal flow of a loop.

Table 6.2: Jump statements

Statement	Purpose	Effect in loop
break	Terminates the nearest enclosing loop or switch immediately	Control goes to the first statement after the loop
continue	Skips the remaining statements of the current iteration	Control goes to the next iteration (to the update part in for)
return	Exits from the current method (and so from any loop inside it)	Control returns to the caller

Example 1.23: break and continue

```

for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;      // skips 3
    if (i == 5) break;        // stops loop at 5
    System.out.print(i + " ");
}
// Output: 1 2 4

```

1.6.5 Labeled Loops

By default *break* and *continue* affect only the innermost loop. A **label** is an identifier followed by a colon placed before a loop. Using **break label;** exits the labeled (outer) loop completely, and **continue label;** jumps to the next iteration of the labeled loop. This is useful in nested loops.

Example 1.24: Labeled break and continue

```

outer:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (i == 2 && j == 2)
            break outer;          // leaves both loops
        System.out.println(i + " " + j);
    }
}
// Output: 1 1 / 1 2 / 1 3 / 2 1

// With continue outer: when i==2 && j==2, control jumps to i=3,
// so the output would be: 1 1, 1 2, 1 3, 2 1, 3 1, 3 2, 3 3

```

1.6.6 Points to Remember

- Forgetting the update statement creates an infinite loop.
- Using = instead of == in a condition is a common error (Java reports a type error for ints).
- A semicolon directly after for(...) or while(...) creates an empty loop body.
- A variable declared in the for header is not available outside the loop.

Practice Section

A. Multiple Choice Questions (15 Marks)

Choose the correct option for each question.

Q1. Who is known as the father of Java?

- (a) Dennis Ritchie (b) James Gosling (c) Bjarne Stroustrup (d) Guido van Rossum

Q2. The output of the Java compiler (javac) is called:

- (a) Source code (b) Machine code (c) Bytecode (d) Assembly code

Q3. Which of the following is NOT a primitive data type in Java?

- (a) int (b) boolean (c) String (d) char

Q4. Wrapping data and methods together into a single unit is called:

- (a) Inheritance (b) Encapsulation (c) Polymorphism (d) Compilation

Q5. Which keyword is used to declare a symbolic constant in Java?

- (a) const (b) static (c) final (d) constant

Q6. Run-time polymorphism in Java is achieved through:

- (a) Method overloading (b) Method overriding (c) Constructors (d) Static methods

Q7. What is the default value of a boolean instance variable?

- (a) true (b) 0 (c) null (d) false

Q8. What is the size of the int data type in Java?

- (a) 2 bytes (b) 4 bytes (c) 8 bytes (d) Depends on platform

Q9. What is the value of the expression $10 / 4$ when both operands are int?

- (a) 2.5 (b) 2 (c) 3 (d) 2.0

Q10. If `int a = 5`; then the value of `y` in `y = a++ + ++a`; is:

- (a) 10 (b) 11 (c) 12 (d) 13

Q11. What is the result of $5 \wedge 3$ in Java?

- (a) 1 (b) 7 (c) 6 (d) 8

Q12. Which type conversion is performed automatically by Java?

- (a) Narrowing (b) Widening (c) Explicit casting (d) None of these

Q13. Which of the following cannot be used as a switch expression?

- (a) int (b) char (c) String (d) double

Q14. Which loop executes its body at least once?

- (a) for (b) while (c) do-while (d) for-each

Q15. Which statement skips the rest of the current iteration and moves to the next iteration?

- (a) break (b) continue (c) return (d) exit

B. Answer Key for MCQs

Table 6.3: Answer key

Q.No.	Answer	Explanation
1	(b)	James Gosling (Sun Microsystems) developed Java
2	(c)	javac converts source code into bytecode (.class)
3	(c)	String is a class (reference type)
4	(b)	Encapsulation binds data and methods in a class
5	(c)	final makes a variable constant
6	(b)	Overriding is resolved at run time by dynamic binding
7	(d)	Boolean instance variables default to false
8	(b)	int is always 32 bits = 4 bytes
9	(b)	Integer division drops the fraction
10	(c)	5 (a becomes 6) + 7 (a becomes 7) = 12
11	(c)	$0101 \wedge 0011 = 0110 = 6$
12	(b)	Widening (small to large) is implicit
13	(d)	switch does not allow double (or long, float, boolean)
14	(c)	do-while tests the condition after the body
15	(b)	continue jumps to the next iteration

C. Broad Questions (10 Questions)

Answer the following questions in detail.

- 1 Define object and class. Explain the difference between them with a suitable Java example.
- 2 Explain the four pillars of OOP: abstraction, encapsulation, inheritance and polymorphism, with examples.
- 3 What is dynamic binding? Distinguish between static binding and dynamic binding and explain with a program how run-time polymorphism is achieved.
- 4 Explain the features of Java (compiled and interpreted, platform independent, distributed, multithreaded, high performance) in detail. Why is Java called 'Write Once, Run Anywhere'?
- 5 Describe the compilation and execution process of a Java program. Differentiate between JDK, JRE and JVM.
- 6 List all primitive data types of Java with their size, range and default values. Explain the scope of local, instance and static variables with example.
- 7 What is type casting? Explain widening and narrowing conversions with examples. What is a symbolic constant and how is it declared?
- 8 Explain arithmetic, relational, logical and bitwise operators of Java with suitable examples. Also explain the difference between prefix and postfix increment.

- 9 Explain different forms of the if statement (simple if, if-else, else-if ladder) and the switch statement with syntax and examples. Write a program to find the largest of three numbers using the ? : operator.
- 10 Explain while, do-while and for loops with syntax and flow. What are jump statements and labeled loops? Write a program to print a star pyramid pattern using nested loops.

— End of Chapter 1 —

Poly Notes Hub