



Gurukul Technicals Institute

Computer education · Robotics · Science projects · Delhi

FREE EBOOK

PROGRAMMING AND WEB

Java Practice Programs

Core Java concepts with twenty-four programs you can run

INSIDE THIS BOOK

- Setting up and your first program
- Data types, operators and input
- Making decisions
- Loops and patterns
- Arrays and strings
- ...and 3 more chapters, practice questions and a glossary

Beginner to intermediate

Students learning Java for school, college or interviews

engineergurukul.com

+91 97116 99511

Java Practice Programs

Learn core Java by running twenty-four complete programs, from your first class to loops, arrays, strings, classes, inheritance, interfaces, exceptions, collections and files. Each program shows its output and explains the idea behind it, ready for practicals, exams and interviews.

WHO IT IS FOR	Students learning Java for school, college or interviews
LEVEL	Beginner to intermediate
SUBJECT AREA	Programming and web
WHAT IS INSIDE	8 chapters, practice questions with answers, and a glossary

CONTENTS

1	Setting up and your first program	3
2	Data types, operators and input	4
3	Making decisions	8
4	Loops and patterns	10
5	Arrays and strings	13
6	Methods, classes and objects	17
7	Inheritance, interfaces and packages	19
8	Exceptions, collections and files	22
	Practice questions	26
	Answers	27
	Glossary	28

Free ebook: you may read, print and share it for learning; it may not be sold or changed. © 2026 Gurukul Technicals Institute. First edition, October 2026. Contact details are on the back cover.

Setting up and your first program

Java programs run on the **JVM** (Java Virtual Machine), software that exists for Windows, Linux, macOS and more. The same compiled program runs on all of them, which is why Java is called "write once, run anywhere".

To write Java you need the **JDK** (Java Development Kit). It contains two tools you will use all the time:

Tool	Job
<code>javac</code>	the compiler: turns <code>Hello.java</code> into <code>Hello.class</code>
<code>java</code>	the launcher: starts the JVM and runs the class

The `.class` file holds **bytecode**, instructions for the JVM rather than for one kind of processor. Install a JDK of version 17 or later; every program in this book works on Java 17. Type `javac -version` in a terminal to check what you have.

Program 1: Hello

Save this as `Hello.java`. The file name must match the public class name exactly, including capital letters.

JAVA

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

Namaste, Java!

Compile with `javac Hello.java`, then run with `java Hello` (no `.class` at the end).

- `public class Hello` declares a **class**. In Java every piece of code lives inside a class.
- `public static void main(String[] args)` is where the program starts. `static` means it can run without creating an object, `void` means it returns nothing, and `args` holds any words typed after the program name.
- `System.out.println` prints a line and moves to the next. `System.out.print` prints without the new line.

DID YOU KNOW

Since Java 11 you can run a single-file program without a separate compile step: `java Hello.java`. It is handy for quick practice.

REMEMBER

Java is **case-sensitive**: `Main`, `main` and `MAIN` are three different names. Most beginner errors are a capital letter or a missing semicolon.

CHAPTER 2

Data types, operators and input

A **variable** is a named box that holds a value. In Java every variable has a fixed **data type**. There are eight basic or **primitive** types, and apart from `boolean`, each has the same size on every computer:

Type	Size	Holds	Example
<code>byte</code>	1 byte	small whole numbers	<code>byte age = 15;</code>
<code>short</code>	2 bytes	whole numbers	<code>short year = 2025;</code>
<code>int</code>	4 bytes	whole numbers (the usual choice)	<code>int marks = 87;</code>
<code>long</code>	8 bytes	very large whole numbers	<code>long km = 384_400L;</code>
<code>float</code>	4 bytes	decimals	<code>float t = 36.6f;</code>
<code>double</code>	8 bytes	decimals (the usual choice)	<code>double price = 249.50;</code>
<code>char</code>	2 bytes	one character	<code>char grade = 'A';</code>
<code>boolean</code>	not stated	<code>true</code> or <code>false</code>	<code>boolean passed = true;</code>

`String` is not primitive; it is a class for text, written in double quotes.

Program 2: Variables and type casting

JAVA

```
public class DataTypes {
    public static void main(String[] args) {
        int students = 42;
        long moonKm = 384_400L;
        double price = 249.50;
        char grade = 'A';
        boolean passed = true;
        String city = "Lucknow";
        System.out.println(city + " class has " + students + " students");
        System.out.println("Moon is about " + moonKm + " km away");
        System.out.println("Price " + price + ", grade " + grade + ", passed " +
passed);
        double a = 7 / 2;
        double b = 7 / 2.0;
        int c = (int) 9.99;
        System.out.println(a + " " + b + " " + c);
    }
}
```

```
Lucknow class has 42 students
Moon is about 384400 km away
Price 249.5, grade A, passed true
3.0 3.5 9
```

- `+` joins text and values into one string. This is called **concatenation**.
- Underscores in `384_400` only make the number easier to read. The `L` marks a `long` value.
- `7 / 2` is 3: two ints give an int, and the fraction is dropped before the answer is stored in the double. `7 / 2.0` is 3.5 because one side is a decimal.
- `(int) 9.99` is a **cast**: it converts to int by cutting off the decimal part, giving 9, not 10.

Program 3: Operators

JAVA

```
public class Operators {
    public static void main(String[] args) {
        int a = 17, b = 5;
        System.out.println("a / b = " + (a / b));
        System.out.println("a % b = " + (a % b));
        a += 3;
        System.out.println("a is now " + a);
        System.out.println("Joined: " + a + b);
        System.out.println("Added: " + (a + b));
        int x = 5;
        int y = x++;
        int z = ++x;
        System.out.println("x=" + x + " y=" + y + " z=" + z);
        boolean ok = a > 10 && b < 10;
        String kind = (a % 2 == 0) ? "even" : "odd";
        System.out.println(ok + ", a is " + kind);
    }
}
```

```
a / b = 3
a % b = 2
a is now 20
Joined: 205
Added: 25
x=7 y=5 z=7
true, a is even
```

Operator	Meaning
<code>+ - * / %</code>	arithmetic; % gives the remainder
<code>+= -= *= /=</code>	compound assignment: <code>a += 3</code> means <code>a = a + 3</code>
<code>++ --</code>	add or subtract 1
<code>== != < > <= >=</code>	comparison, giving true or false
<code>&&</code>	! logical AND, OR, NOT
<code>? :</code>	the conditional operator: picks one of two values

`x++` uses `x` and then adds 1, so `y` got 5. `++x` adds 1 first, so `z` got 7. "Joined: 205" happens because Java reads left to right: `text + 20` is `text`, and `text + 5` is more text. Brackets force the addition first.

Reading input with Scanner

The `Scanner` class reads what the user types. It lives in the package `java.util`, so the program starts with `import java.util.Scanner;`

Method	Reads
<code>nextInt()</code>	an int
<code>nextDouble()</code>	a double
<code>next()</code>	one word (stops at a space)
<code>nextLine()</code>	the rest of the line, spaces included
<code>nextBoolean()</code>	true or false

Program 4: Total and average marks

JAVA

```
import java.util.Scanner;
public class Average {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Your name: ");
        String name = sc.nextLine();
        System.out.print("Marks in three subjects: ");
        int m1 = sc.nextInt(), m2 = sc.nextInt(), m3 = sc.nextInt();
        int total = m1 + m2 + m3;
        double avg = total / 3.0;
        System.out.printf("%s: total %d, average %.2f%n", name, total, avg);
        sc.close();
    }
}
```

```
Your name: Neha Gupta
Marks in three subjects: 78 85 90
Neha Gupta: total 253, average 84.33
```

`new Scanner(System.in)` creates a scanner that reads from the keyboard. `printf` works like C's: `%s` is text, `%d` an int, `%.2f` a decimal with two places, and `%n` a new line. `sc.close()` releases the scanner when we are done.

REMEMBER

After `nextInt()`, the Enter key is still waiting in the input. A `nextLine()` straight after it returns an empty line. Read the line first, as here, or add an extra `sc.nextLine();` to swallow the leftover Enter.

Making decisions

Java decides with `if`, `else if`, `else` and `switch`. A condition must be a `boolean`: unlike C, Java does not treat 0 as false, so `if (n = 5)` is a compile error, which saves you from a classic bug.

Program 5: Leap year

JAVA

```
import java.util.Scanner;
public class LeapYear {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Year: ");
        int y = sc.nextInt();
        boolean leap = (y % 4 == 0 && y % 100 != 0) || y % 400 == 0;
        System.out.println(y + (leap ? " is" : " is not") + " a leap year");
        sc.close();
    }
}
```

```
Year: 2024
2024 is a leap year
```

Years divisible by 4 are leap years, except century years, which must be divisible by 400. So 2000 was a leap year and 1900 was not.

Program 6: Largest of three

JAVA

```
import java.util.Scanner;
public class Largest {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Three numbers: ");
        int a = sc.nextInt(), b = sc.nextInt(), c = sc.nextInt();
        int max = a;
        if (b > max) max = b;
        if (c > max) max = c;
        System.out.println("Largest: " + max);
        System.out.println("Using Math.max: " + Math.max(a, Math.max(b, c)));
        sc.close();
    }
}
```

```
Three numbers: 56 91 73
Largest: 91
Using Math.max: 91
```

Assume the first number is the largest, then let each other number challenge it. The last line uses the ready-made `Math.max` instead.

Program 7: Grade from marks

This uses a sample grading scheme; your school's may differ.

JAVA

```
import java.util.Scanner;
public class Grade {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Marks (0-100): ");
        int m = sc.nextInt();
        String grade;
        if (m < 0 || m > 100) grade = "Invalid";
        else if (m >= 90) grade = "A";
        else if (m >= 75) grade = "B";
        else if (m >= 60) grade = "C";
        else if (m >= 40) grade = "D";
        else grade = "E";
        System.out.println("Grade: " + grade);
        sc.close();
    }
}
```

```
Marks (0-100): 82
Grade: B
```

An **else-if ladder** is checked from the top and stops at the first true condition, so `m >= 75` only runs for marks under 90.

Program 8: Calculator with switch

JAVA

```
import java.util.Scanner;
public class Calculator {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a sum like 12 * 4: ");
        double a = sc.nextDouble();
        String op = sc.next();
        double b = sc.nextDouble();
        double result = switch (op) {
            case "+" -> a + b;
            case "-" -> a - b;
            case "*" -> a * b;
            case "/" -> (b == 0) ? Double.NaN : a / b;
            default -> Double.NaN;
        };
        System.out.println(Double.isNaN(result) ? "Cannot calculate that" : "=" +
result);
        sc.close();
    }
}
```

```
Enter a sum like 12 * 4: 12 * 4
= 48.0
```

Type spaces between the parts, because `next()` reads one word at a time. This is a **switch expression** (standard since Java 14): each `case "+" ->` gives a value, and there is no fall-through, so no `break` is needed. `Double.NaN` means "not a number" and marks an impossible answer such as division by zero.

TIP

Older code uses `case "+": result = a + b; break;`. In that colon style, a forgotten `break` lets the next case run too.

CHAPTER 4

Loops and patterns

Loop	Best when
<code>for</code>	you know the number of repetitions
<code>while</code>	you repeat until a condition changes
<code>do-while</code>	the body must run at least once

Loop	Best when
for-each	you visit every item of an array or list

`break` leaves a loop at once; `continue` skips to the next round.

Program 9: Multiplication tables with do-while

JAVA

```
import java.util.Scanner;
public class Table {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String again;
        do {
            System.out.print("Table of: ");
            int n = sc.nextInt();
            for (int i = 1; i <= 10; i++)
                System.out.println(n + " x " + i + " = " + n * i);
            System.out.print("Another? (y/n): ");
            again = sc.next();
        } while (again.equalsIgnoreCase("y"));
        sc.close();
    }
}
```

```
Table of: 9
9 x 1 = 9
...
9 x 10 = 90
Another? (y/n): n
```

The do-while body runs once before the question is asked, which suits a menu.

`equalsIgnoreCase` accepts both "y" and "Y".

Program 10: Factorial with while, Fibonacci with for

JAVA

```
public class FactFib {
    public static void main(String[] args) {
        int n = 15, i = 1;
        long fact = 1;
        while (i <= n) {
            fact *= i;
            i++;
        }
        System.out.println(n + "! = " + fact);
        int a = 0, b = 1;
        for (int k = 1; k <= 12; k++) {
            System.out.print(a + " ");
            int next = a + b;
            a = b;
            b = next;
        }
        System.out.println();
    }
}
```

```
15! = 1307674368000
0 1 1 2 3 5 8 13 21 34 55 89
```

- 15! is far beyond an int's limit (about 2.1 billion), so `fact` is a `Long`, which holds factorials up to 20!.
- In the Fibonacci series each term is the sum of the two before it. `a` and `b` hold the last two, and both slide forward after each print.

Program 11: Reverse and palindrome number

JAVA

```
public class Palindrome {
    public static void main(String[] args) {
        int num = 12321, n = num, rev = 0;
        while (n > 0) {
            rev = rev * 10 + n % 10;
            n /= 10;
        }
        System.out.println("Reverse: " + rev);
        System.out.println(num + (rev == num ? " is" : " is not") + " a palindrome");
    }
}
```

```
Reverse: 12321
12321 is a palindrome
```

`n % 10` takes the last digit and `n /= 10` removes it. We work on a copy, `n`, so `num` is still there for the final comparison.

Program 12: Two patterns

JAVA

```
public class Patterns {
    public static void main(String[] args) {
        int rows = 4;
        for (int i = 1; i <= rows; i++) {
            for (int j = 1; j <= i; j++)
                System.out.print("* ");
            System.out.println();
        }
        for (int i = 1; i <= rows; i++) {
            System.out.print(" ".repeat(rows - i));
            for (int j = 1; j <= i; j++)
                System.out.print(j + " ");
            System.out.println();
        }
    }
}
```

```
*
* *
* * *
* * * *
  1
 1 2
1 2 3
1 2 3 4
```

The outer loop picks the row and the inner loop prints what is in it. In the second pattern, row `i` first gets `rows - i` spaces. `" ".repeat(3)` makes a string of three spaces.

CHAPTER 5

Arrays and strings

An **array** holds a fixed number of values of one type. `int[] marks = new int[5];` makes five int slots, filled with 0. The **index** runs from 0 to `length - 1`, and `marks.length` gives the size. Using index 5 here throws an `ArrayIndexOutOfBoundsException`, so Java stops the mistake instead of damaging memory.

Program 13: Array statistics, sorting and searching

JAVA

```
import java.util.Arrays;
public class ArrayWork {
    public static void main(String[] args) {
        int[] a = {67, 89, 45, 92, 78, 56};
        int max = a[0], min = a[0], sum = 0;
        for (int m : a) {
            if (m > max) max = m;
            if (m < min) min = m;
            sum += m;
        }
        System.out.printf("Max %d, Min %d, Average %.2f%n",
            max, min, (double) sum / a.length);
        for (int i = 0; i < a.length - 1; i++)
            for (int j = 0; j < a.length - 1 - i; j++)
                if (a[j] > a[j + 1]) {
                    int t = a[j];
                    a[j] = a[j + 1];
                    a[j + 1] = t;
                }
        System.out.println(Arrays.toString(a));
        System.out.println("78 is at index " + Arrays.binarySearch(a, 78));
        String[] cities = {"Pune", "Agra", "Kochi", "Bhopal"};
        Arrays.sort(cities);
        System.out.println(Arrays.toString(cities));
    }
}
```

```
Max 92, Min 45, Average 71.17
[45, 56, 67, 78, 89, 92]
78 is at index 3
[Agra, Bhopal, Kochi, Pune]
```

- `for (int m : a)` is the **for-each loop**: it hands you each element in turn, with no index to get wrong. `(double) sum` makes the division keep its decimals.
- The nested loops are **bubble sort**: neighbours are swapped when out of order, and each pass pushes the largest remaining value to the end. Examiners like to see it written by hand.
- `Arrays.sort` sorts in one line, for numbers and text. `Arrays.binarySearch` works only on a sorted array.

Program 14: Adding two matrices

JAVA

```
public class Matrix {
    public static void main(String[] args) {
        int[][] a = {{1, 2, 3}, {4, 5, 6}};
        int[][] b = {{9, 8, 7}, {6, 5, 4}};
        int[][] sum = new int[a.length][a[0].length];
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a[0].length; j++) {
                sum[i][j] = a[i][j] + b[i][j];
                System.out.print(sum[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

```
10 10 10
10 10 10
```

A **two-dimensional array** is an array of rows. `a.length` is the number of rows (2) and `a[0].length` the number of columns (3), so the loops fit any size.

Strings and StringBuilder

A **String** holds text. Strings are **immutable**: once made, they never change. Methods such as `toUpperCase()` return a new string and leave the old one as it was.

REMEMBER

Compare strings with `equals()`, never `==`. The `==` operator only checks whether two variables refer to the very same object, so two equal words can still give false.

Program 15: Useful String methods

JAVA

```
public class StringMethods {
    public static void main(String[] args) {
        String s = " Gateway of India ";
        String t = s.trim();
        System.out.println("[ " + t + " ] has " + t.length() + " characters");
        System.out.println(t.toUpperCase());
        System.out.println("First letter: " + t.charAt(0));
        System.out.println("India starts at " + t.indexOf("India"));
        System.out.println(t.substring(0, 7));
        System.out.println(t.replace("of", "to"));
        String a = "delhi", b = "DELHI";
        System.out.println(a.equals(b) + " " + a.equalsIgnoreCase(b));
    }
}
```

```
[Gateway of India] has 16 characters
GATEWAY OF INDIA
First letter: G
India starts at 11
Gateway
Gateway to India
false true
```

`trim()` removes spaces at both ends. Positions start at 0, so "India" begins at index 11.

`substring(0, 7)` gives the characters from index 0 up to, but not including, index 7.

Program 16: Vowels, palindrome word and StringBuilder

Because strings are immutable, adding to a string in a loop creates a new string every time. A **StringBuilder** is a changeable string, much faster for building text piece by piece.

JAVA

```
import java.util.Scanner;
public class WordCheck {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a word: ");
        String w = sc.next().toLowerCase();
        int vowels = 0;
        for (char c : w.toCharArray())
            if ("aeiou".indexOf(c) >= 0) vowels++;
        String rev = new StringBuilder(w).reverse().toString();
        System.out.println("Vowels: " + vowels);
        System.out.println(w.equals(rev) ? "Palindrome" : "Not a palindrome");
        StringBuilder sb = new StringBuilder();
        for (int i = 1; i <= 5; i++) {
            sb.append(i);
            if (i < 5) sb.append(", ");
        }
        sb.insert(0, "Roll numbers: ");
        System.out.println(sb + " (" + sb.length() + " characters)");
        sc.close();
    }
}
```

```
Enter a word: Malayalam
Vowels: 4
Palindrome
Roll numbers: 1, 2, 3, 4, 5 (27 characters)
```

- `toLowerCase()` makes "M" and "m" equal. `toCharArray()` turns the word into an array of characters for the for-each loop, and `"aeiou".indexOf(c)` is -1 when `c` is not a vowel.
- `reverse` turns a `StringBuilder`'s text around, and `toString()` gives an ordinary `String` back to compare with `equals`.
- `append` adds to the end and `insert` adds at a position, all inside one object.

Methods, classes and objects

A **method** is a named block of code that does one job. It may take **parameters** and may **return** a value.

Program 17: Methods, overloading and primes

JAVA

```
public class MathTools {
    static int square(int n) { return n * n; }
    static double square(double n) { return n * n; }
    static boolean isPrime(int n) {
        if (n < 2) return false;
        for (int d = 2; d * d <= n; d++)
            if (n % d == 0) return false;
        return true;
    }
    static void line(int count) { System.out.println("=".repeat(count)); }
    public static void main(String[] args) {
        line(20);
        System.out.println(square(9) + " " + square(2.5));
        for (int n = 2; n <= 50; n++)
            if (isPrime(n)) System.out.print(n + " ");
        System.out.println();
        line(20);
    }
}
```

```
=====
81 6.25
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
=====
```

- There are two methods named `square`. Java picks one by the argument type: `square(9)` calls the int version, `square(2.5)` the double version. This is **method overloading**.
- `isPrime` tests divisors only up to the square root (`d * d <= n`), which is enough, and `return false` ends it as soon as one divides `n`.
- `void` means `line` returns nothing. These methods are `static`, so `main` can call them without an object. A method whose body is one short statement can sit on one line.

Classes and objects

A **class** is a design; an **object** is a real thing made from it with `new`. A class holds **fields** (its data) and methods (what it can do). A **constructor** has the class's name and no return type, and runs when `new` creates an object.

Marking fields `private` and giving public methods to use them is called **encapsulation**: nobody can set a balance to any number they like, because every change goes through a method that checks it.

Program 18: Bank account

JAVA

```
public class BankAccount {
    private final String owner;
    private double balance;

    public BankAccount(String owner) { this(owner, 0); }
    public BankAccount(String owner, double balance) {
        this.owner = owner;
        this.balance = balance;
    }
    public void deposit(double amount) {
        if (amount > 0) balance += amount;
    }
    public boolean withdraw(double amount) {
        if (amount <= 0 || amount > balance) return false;
        balance -= amount;
        return true;
    }
    @Override
    public String toString() { return owner + ": Rs " + balance; }

    public static void main(String[] args) {
        BankAccount a = new BankAccount("Rohan Das", 2000);
        BankAccount b = new BankAccount("Lata Menon");
        a.deposit(1500);
        if (!a.withdraw(5000)) System.out.println("Withdrawal refused");
        b.deposit(800);
        System.out.println(a);
        System.out.println(b);
    }
}
```

```
Withdrawal refused
Rohan Das: Rs 3500.0
Lata Menon: Rs 800.0
```

- There are two constructors. The one-argument version calls the other with `this(owner, 0)`, so a new account starts at zero.
- `this.owner = owner` copies the parameter into the field of the same name; `this` means "this object".
- `final` means `owner` can be set once and never changed.
- `toString()` decides how an object prints. `@Override` asks the compiler to check that we really are replacing an inherited method.

DID YOU KNOW

Software that handles real money avoids `double`, because binary decimals cannot hold some values exactly, such as 0.1. Programmers use `BigDecimal` or count whole paise in a `long`.

CHAPTER 7

Inheritance, interfaces and packages

Inheritance lets a class reuse another class's fields and methods with `extends`. The existing class is the **superclass** (parent) and the new one is the **subclass** (child). Java allows only one superclass per class. `super(...)` calls the parent's constructor; in Java 17 it must be the first statement of the child's constructor.

Program 19: Shapes: inheritance and an abstract class

An **abstract class** cannot be used to create objects. It is a partly built parent: some methods are written, others are left `abstract` for each child to supply.

```

abstract class Shape {
    private final String name;
    Shape(String name) { this.name = name; }
    abstract double area();
    void describe() { System.out.printf("%s: area %.2f%n", name, area()); }
}

class Circle extends Shape {
    private final double r;
    Circle(double r) {
        super("Circle");
        this.r = r;
    }
    @Override
    double area() { return Math.PI * r * r; }
}

class Rectangle extends Shape {
    private final double l, w;
    Rectangle(double l, double w) {
        super("Rectangle");
        this.l = l;
        this.w = w;
    }
    @Override
    double area() { return l * w; }
}

public class Shapes {
    public static void main(String[] args) {
        Shape[] list = {new Circle(3), new Rectangle(4, 2.5)};
        for (Shape s : list) s.describe();
    }
}

```

```

Circle: area 28.27
Rectangle: area 10.00

```

Save it as `Shapes.java`: a file may hold several classes, but only one public class, which names the file. `Circle` and `Rectangle` inherit `describe()` from `Shape` and **override** `area()`. `describe()` is written once, yet when it runs for the circle, Java calls the circle's `area()`, because the object really is a `Circle`. One call behaving differently for different objects is **polymorphism**. `new Shape("x")` would not compile, because the class is abstract.

Program 20: An interface

An **interface** is a pure contract: a list of methods a class promises to provide. A class **implements** an interface, and it may implement several, which gives Java much of the benefit of multiple inheritance.

```

interface Payment {
    String method();
    void pay(double amount);
}

class Card implements Payment {
    private final String last4;
    Card(String last4) { this.last4 = last4; }
    public String method() { return "Card"; }
    public void pay(double amount) {
        System.out.println("Rs " + amount + " charged to card ending " + last4);
    }
}

class Cash implements Payment {
    public String method() { return "Cash"; }
    public void pay(double amount) {
        System.out.println("Rs " + amount + " received in cash");
    }
}

public class Checkout {
    public static void main(String[] args) {
        Payment[] options = {new Card("4821"), new Cash()};
        for (Payment p : options) {
            System.out.print(p.method() + ": ");
            p.pay(450);
        }
    }
}

```

Card: Rs 450.0 charged to card ending 4821
 Cash: Rs 450.0 received in cash

Interface methods are public, so the classes must mark their versions `public`.

	Abstract class	Interface
Keyword to use it	<code>extends</code>	<code>implements</code>
How many per class	one	many
Fields	any	constants only
Constructors	yes	no
Best for	related classes sharing code	unrelated classes sharing a promise

Program 21: Packages

A **package** is a folder of related classes with a shared name, such as `java.util`. Packages keep large programs organised and stop name clashes. Make a folder named `shop`, and save this inside it as `shop/Product.java`:

JAVA

```
package shop;

public class Product {
    private final String name;
    private final double price;
    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }
    public String label() { return name + " - Rs " + price; }
}
```

Save this as `Store.java` in the folder that contains `shop`:

JAVA

```
import shop.Product;
public class Store {
    public static void main(String[] args) {
        Product p = new Product("Geometry box", 120);
        System.out.println(p.label());
    }
}
```

```
Geometry box - Rs 120.0
```

Compile both with `javac Store.java shop/Product.java`, then run `java Store`. The first line of `Product.java`, `package shop;`, must match its folder name. `import shop.Product;` lets `Store` use the class by its short name. Only `public` classes and members can be used freely from another package.

CHAPTER 8

Exceptions, collections and files

An **exception** is an error that happens while the program runs, such as dividing by zero or opening a missing file. If nothing handles it, the program stops with a long error message. You handle it with `try` and `catch`.

Kind	Must you handle it?	Examples
Checked	yes: catch it or declare it with <code>throws</code>	<code>IOException</code>
Unchecked	no, but you may	<code>ArithmeticException</code> , <code>ArrayIndexOutOfBoundsException</code>

Program 22: try, catch, finally and your own exception

JAVA

```

class InsufficientBalanceException extends Exception {
    InsufficientBalanceException(String message) { super(message); }
}

public class Exceptions {
    static double balance = 1000;
    static void withdraw(double amount) throws InsufficientBalanceException {
        if (amount > balance)
            throw new InsufficientBalanceException("Balance is only Rs " +
            balance);
        balance -= amount;
    }
    public static void main(String[] args) {
        try {
            System.out.println(10 / 0);
        } catch (ArithmeticException e) {
            System.out.println("Maths error: " + e.getMessage());
        } finally {
            System.out.println("finally always runs");
        }
        try {
            withdraw(400);
            withdraw(900);
        } catch (InsufficientBalanceException e) {
            System.out.println("Refused. " + e.getMessage());
        }
    }
}

```

```

Maths error: / by zero
finally always runs
Refused. Balance is only Rs 600.0

```

- When an exception is thrown inside `try`, Java jumps to the matching `catch`, skipping the rest of the `try` block.
- `finally` runs whether or not an exception happened. It is the place to close files and connections.

- `InsufficientBalanceException` is our own **checked exception**, made by extending `Exception`. Because it is checked, `withdraw` must declare it with `throws`, and the caller must catch it or declare it too.

TIP

Never write an empty `catch` block. At the very least print the message, or the error disappears silently and is hard to find later.

Collections

An array cannot grow. The **collections** in `java.util` can. Two you will use constantly:

Class	Holds	Key methods
<code>ArrayList</code>	a list in order, which grows as needed	<code>add</code> , <code>get</code> , <code>set</code> , <code>remove</code> , <code>size</code> , <code>contains</code>
<code>HashMap</code>	pairs of a key and a value	<code>put</code> , <code>get</code> , <code>getOrDefault</code> , <code>containsKey</code> , <code>remove</code>

The type inside angle brackets, as in `ArrayList<String>`, says what the collection holds. This is called **generics**. Collections hold objects only, so numbers use wrapper classes such as `Integer` instead of `int`; Java converts between them automatically.

Program 23: A student list with ArrayList

JAVA

```
import java.util.ArrayList;
class Student {
    String name;
    int marks;
    Student(String name, int marks) { this.name = name; this.marks = marks; }
}

public class StudentList {
    public static void main(String[] args) {
        ArrayList<Student> list = new ArrayList<>();
        list.add(new Student("Zoya", 81));
        list.add(new Student("Amit", 67));
        list.add(new Student("Harpreet", 93));
        list.add(new Student("Deepa", 74));
        list.removeIf(s -> s.marks < 70);
        list.sort((x, y) -> y.marks - x.marks);
        System.out.println(list.size() + " students scored 70 or more:");
        for (Student s : list)
            System.out.println(s.name + " " + s.marks);
        System.out.println("Topper: " + list.get(0).name);
    }
}
```

```
3 students scored 70 or more:  
Harpreet 93  
Zoya 81  
Deepa 74  
Topper: Harpreet
```

In `new ArrayList<>()`, the empty `<>` lets Java work out the type. `s -> s.marks < 70` is a **lambda**, a tiny unnamed function: "given student `s`, is the mark below 70?" `removeIf` deletes every student for whom it is true. The second lambda tells `sort` how to compare two students; `y.marks - x.marks` puts higher marks first.

Reading a text file

To read a file line by line, wrap a `FileReader` in a `BufferedReader`, which reads efficiently and offers `readLine()`. `readLine()` returns `null` at the end of the file. A **try-with-resources** block, `try (...)`, closes the file automatically, even if an error happens.

Program 24: Word frequency from a file

First make a file named `story.txt` in the same folder, using Notepad, with these three lines:

```
The crow was thirsty.  
The crow found a pot.  
The pot had a little water.
```

JAVA

```
import java.io.*;  
import java.util.*;  
public class WordFrequency {  
    public static void main(String[] args) {  
        Map<String, Integer> freq = new HashMap<>();  
        int lines = 0;  
        try (BufferedReader br = new BufferedReader(new FileReader("story.txt")))  
        {  
            String line;  
            while ((line = br.readLine()) != null) {  
                lines++;  
                for (String w : line.toLowerCase().split("[^a-z]+"))  
                    if (!w.isEmpty())  
                        freq.put(w, freq.getOrDefault(w, 0) + 1);  
            }  
        } catch (IOException e) {  
            System.out.println("Cannot read file: " + e.getMessage());  
            return;  
        }  
        System.out.println("Lines: " + lines);  
        System.out.println(new TreeMap<>(freq));  
    }  
}
```

Lines: 3

```
{a=2, crow=2, found=1, had=1, little=1, pot=2, the=3, thirsty=1, was=1, water=1}
```

- `split("[^a-z]+")` cuts the line wherever there are characters that are not small letters, such as spaces and full stops, leaving only words.
- `freq.getOrDefault(w, 0) + 1` reads the old count (0 for a new word) and `put` stores the new one.
- A `HashMap` keeps no particular order, so we copy it into a `TreeMap`, which sorts by key, before printing.
- `IOException` is a checked exception, so the compiler insists we catch it. Delete `story.txt` and run again to see the message.

TRY THIS

Change Program 24 to print only words that appear more than once. Loop over `freq.entrySet()` and test `entry.getValue() > 1`.

CHECK YOURSELF

Practice questions

1. What do `javac` and `java` each do, and what is in a `.class` file?
2. Why must `Hello.java` contain a public class named exactly `Hello`?
3. What does `7 / 2` give in Java, and how do you get 3.5?
4. Why does `"Total: " + 4 + 6` print "Total: 46"? How do you print 10?
5. What is the difference between `x++` and `++x`?
6. Why can `nextLine()` return an empty string straight after `nextInt()`?
7. Why is no `break` needed in a switch that uses `case "+" ->?`
8. What happens if you use an array index equal to the array's length?
9. Why should strings be compared with `equals()` rather than `==`?
10. What does "strings are immutable" mean, and why does `StringBuilder` exist?
11. What is method overloading? Give an example from this book.
12. What is a constructor, and what does `this(owner, 0)` do in Program 18?
13. What is encapsulation?
14. What does `super("Circle")` do in Program 19?
15. Give two differences between an abstract class and an interface.
16. What statement must come first in `Product.java` to put the class in the package `shop`?
17. What is the difference between checked and unchecked exceptions?

18. When does a `finally` block run?
19. Why can an `ArrayList` not hold `int` directly, and what does it use instead?
20. What does try-with-resources do for you?

ANSWER KEY

Answers

1. `javac` compiles source code into bytecode, which the `.class` file holds; `java` starts the JVM to run it.
2. Java requires a public class to be saved in a file with the same name, letter for letter.
3. It gives 3, because both are ints. Write `7 / 2.0` or `(double) 7 / 2` to get 3.5.
4. Java works left to right: `text + 4` is `text`, then `+ 6` adds more text. Write `"Total: " + (4 + 6)`.
5. `x++` uses the old value and then adds 1; `++x` adds 1 first and uses the new value.
6. `nextInt()` leaves the Enter key in the input, and `nextLine()` reads up to it at once. Call an extra `nextLine()` to clear it.
7. The arrow form never falls through: only the matching case runs.
8. Java throws an `ArrayIndexOutOfBoundsException`, because the last valid index is `length - 1`.
9. `==` checks whether two variables refer to the same object; `equals()` compares the characters.
10. A `String` never changes once made; each change makes a new string. `StringBuilder` can change, so building text in a loop is faster.
11. Several methods with the same name but different parameter lists, such as `square(int)` and `square(double)` in Program 17.
12. It sets up a new object when `new` runs. `this(owner, 0)` calls the other constructor with a starting balance of 0.
13. Keeping fields private and allowing changes only through public methods that check the values.
14. It calls the `Shape` constructor to set `name` before `Circle` sets its own field `r`.
15. A class can extend only one abstract class but implement many interfaces; an abstract class can have constructors and ordinary fields, an interface cannot.
16. `package shop;`
17. Checked exceptions must be caught or declared with `throws`, or the code will not compile; unchecked exceptions need not be.
18. Always, after the `try` and any `catch`, whether or not an exception happened.
19. Collections hold objects only. It uses the wrapper class `Integer`, and Java converts automatically.
20. It closes the file or other resource automatically when the block ends, even after an exception.

Glossary

Abstract class

a class that cannot make objects and may leave methods for subclasses.

ArrayList

a list from `java.util` that grows as items are added.

Bytecode

the instructions in a `.class` file, run by the JVM.

Class

a design describing the fields and methods of objects.

Constructor

a special method that sets up a new object.

Encapsulation

keeping fields private, reached only through methods.

Exception

an error event that happens while a program runs.

HashMap

a collection of key-value pairs for fast lookup.

Inheritance

a class reusing another's fields and methods with `extends`.

Interface

a set of methods that a class promises to provide.

JDK

the Java Development Kit: the compiler, the launcher and more.

JVM

the Java Virtual Machine, which runs bytecode.

Lambda

a short unnamed function, such as
`s -> s.marks < 70.`

Overloading

methods sharing a name but taking different parameters.

Overriding

a subclass's own version of an inherited method.

Package

a named folder of related classes.

Polymorphism

one call acting differently for different objects.

StringBuilder

a changeable string used to build text efficiently.



Gurukul Technicals Institute

Computer education institute in Delhi, with authorised member centres, robotics classes and science projects, working models and DIY kits.

COURSES

Computer & Office Skills · Accounting & Business Software · Programming & Web Development · Design & Digital Skills · Hardware & Networking · Robotics & Coding · Other Programmes

PROJECTS AND KITS

Ready-made school projects · DIY and STEM kits · Robotics and coding · Components and spares · Engineering projects

CONTACT

Head office: A-10, Chief Commissioner Colony, opposite Rana Pratap Bagh, Delhi 110007

Phone: 011 2721 1512 · **Mobile and WhatsApp:** +91 97116 99511

Email: engineergurukul@yahoo.co.in

Website: engineergurukul.com · **Instagram:** [@science_projects_club](https://www.instagram.com/science_projects_club)

More free ebooks for computer courses, robotics and science projects: engineergurukul.com/resources/ebooks/

Free to read, print and share. Not for sale.