

# Java Programming Notes

From Fundamentals to Intermediate Concepts

---

A Comprehensive Learning Guide Covering  
Core Java, OOP, Collections, Exception Handling,  
File Handling, and Best Practices

---

**Prepared By**

Manoj S

[manojcs6317@gmail.com](mailto:manojcs6317@gmail.com)

Beginner to Intermediate Level  
Based on Industry Coding Standards

December 24, 2025

## Contents

<b>1</b>	<b>Introduction to Java</b>	<b>3</b>
1.1	What is Java? . . . . .	3
1.2	Java Program Structure . . . . .	3
<b>2</b>	<b>Input and Output</b>	<b>4</b>
2.1	Understanding I/O Operations . . . . .	4
2.2	The Scanner Class . . . . .	4
2.3	Output Methods . . . . .	5
<b>3</b>	<b>Data Types</b>	<b>5</b>
3.1	Understanding Type Systems . . . . .	5
3.2	Primitive Data Types . . . . .	5
3.3	Type Casting . . . . .	6
3.4	The final Keyword . . . . .	7
<b>4</b>	<b>Control Structures</b>	<b>7</b>
4.1	Understanding Control Flow . . . . .	7
4.2	If-Else Statements . . . . .	7
4.3	Nested Conditions . . . . .	8
4.4	Switch Statement . . . . .	9
<b>5</b>	<b>Loops</b>	<b>9</b>
5.1	Understanding Iteration . . . . .	9
5.2	For Loop Structure . . . . .	10
5.3	Pattern Programming . . . . .	10
<b>6</b>	<b>Arrays and Strings</b>	<b>13</b>
6.1	Understanding Arrays . . . . .	13
6.2	Enhanced For Loop . . . . .	14
6.3	Understanding Strings . . . . .	14
6.4	String Methods . . . . .	14
6.5	String Iteration . . . . .	15
<b>7</b>	<b>Methods (Functions)</b>	<b>16</b>
7.1	Understanding Methods . . . . .	16
7.2	Methods Without Return Value . . . . .	16
7.3	Methods With Return Value . . . . .	16
7.4	Method Overloading . . . . .	17
7.5	Variable Scope . . . . .	18
<b>8</b>	<b>Object-Oriented Programming Fundamentals</b>	<b>18</b>
8.1	Understanding OOP . . . . .	18
8.2	Classes and Objects . . . . .	19
8.3	Constructors . . . . .	19
8.4	The 'this' Keyword . . . . .	20
8.5	Encapsulation . . . . .	21
<b>9</b>	<b>Inheritance</b>	<b>22</b>
9.1	Understanding Inheritance . . . . .	22
9.2	The super Keyword . . . . .	24
9.3	Method Overriding . . . . .	25

<b>10 Abstract Classes</b>	<b>26</b>
10.1 Understanding Abstraction . . . . .	26
<b>11 Interfaces</b>	<b>27</b>
11.1 Understanding Interfaces . . . . .	27
11.2 Multiple Interface Implementation . . . . .	28
<b>12 Collections Framework</b>	<b>29</b>
12.1 Understanding Collections . . . . .	29
12.2 ArrayList . . . . .	30
12.3 HashSet . . . . .	31
12.4 HashMap . . . . .	32
12.5 TreeMap . . . . .	33
<b>13 Exception Handling</b>	<b>34</b>
13.1 Understanding Exceptions . . . . .	34
13.2 Try-Catch-Finally . . . . .	34
13.3 Custom Exceptions . . . . .	35
<b>14 File Handling</b>	<b>36</b>
14.1 Understanding File I/O . . . . .	36
14.2 Basic File Operations . . . . .	36
14.3 Writing to Files . . . . .	36
14.4 Reading from Files . . . . .	37
14.5 Working with JSON . . . . .	37
14.6 Serialization . . . . .	39
<b>15 Best Practices and Coding Standards</b>	<b>40</b>
15.1 Naming Conventions . . . . .	40
15.2 Code Organization . . . . .	40
15.3 Memory Management . . . . .	41
15.4 Exception Handling Guidelines . . . . .	41
15.5 OOP Principles . . . . .	41
<b>16 Quick Reference Guide</b>	<b>42</b>
16.1 Common Syntax Patterns . . . . .	42
16.2 Essential Imports . . . . .	43
16.3 Common String Methods . . . . .	43
16.4 Common ArrayList Methods . . . . .	43
16.5 Access Modifier Summary . . . . .	43
<b>17 Learning Path and Next Steps</b>	<b>44</b>
17.1 Topics Covered . . . . .	44
17.2 Recommended Next Steps . . . . .	44
17.3 Practice Resources . . . . .	45
<b>18 Conclusion</b>	<b>45</b>

## 1 Introduction to Java

### 1.1 What is Java?

Java is a high-level, class-based, object-oriented programming language that was developed by Sun Microsystems (now owned by Oracle) in 1995. It was designed with the philosophy of "Write Once, Run Anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.

#### Key Characteristics:

- **Platform Independent:** Java code is compiled to bytecode that runs on the Java Virtual Machine (JVM), making it portable across different operating systems
- **Object-Oriented:** Everything in Java is associated with classes and objects, along with their attributes and methods
- **Strongly Typed:** Every variable must have a declared type
- **Automatic Memory Management:** Java uses garbage collection to automatically manage memory
- **Robust and Secure:** Strong compile-time checking, exception handling, and security features

### 1.2 Java Program Structure

Every Java application must have at least one class definition and a main method which serves as the entry point for the program. The Java compiler looks for this main method to begin execution.

#### Structure Components:

- **Package Declaration:** Optional, specifies the package the class belongs to
- **Import Statements:** Brings other Java classes into scope
- **Class Definition:** Container for all code and data
- **Main Method:** Entry point where program execution begins

**Theory:** The `public` access modifier makes the class accessible from anywhere. The `static` keyword means the method belongs to the class itself rather than instances of the class, allowing the JVM to call it without creating an object. The `void` return type indicates the method doesn't return any value. The `String[] args` parameter allows command-line arguments to be passed to the program.

#### Example:

```
1 package DAY1;
2
3 public class basics {
4     public static void main(String[] args) {
5         System.out.println("Hello World!");
6     }
7 }
```

### Understanding the Components

- package DAY1;: Organizes classes into namespaces
- public class basics: Public class definition (must match filename)
- public static void main(String[] args): Program entry point
- System.out.println(): Prints output to console with newline

## 2 Input and Output

### 2.1 Understanding I/O Operations

Input/Output (I/O) operations are fundamental to interactive programs. They allow programs to receive data from users (input) and display results back to them (output). Java provides several classes for I/O operations, with the **Scanner** class being one of the most commonly used for console input.

### 2.2 The Scanner Class

**Theory:** The **Scanner** class, part of the `java.util` package, breaks input into tokens using delimiters (whitespace by default). It provides various methods to read different data types. When you create a **Scanner** object with `System.in` as the parameter, it reads from the standard input stream (keyboard).

#### Key Concepts:

- **Token:** A unit of input separated by delimiters
- **Buffer:** Temporary storage area where input is held
- `nextLine()` reads entire line including spaces until newline
- `next()` reads only until whitespace
- `nextInt()`, `nextDouble()` read specific data types
- Always close **Scanner** with `close()` to prevent resource leaks

**Common Issue:** When using `nextInt()` followed by `nextLine()`, the newline character remains in the buffer. You need an extra `nextLine()` call to consume it.

#### Example:

```

1 package DAY1;
2 import java.util.Scanner;
3
4 public class InputOutput {
5     public static void main(String[] args) {
6         Scanner sc = new Scanner(System.in);
7
8         System.out.print("Enter the name :");
9         String name = sc.nextLine(); // Reads entire line with spaces
10
11        System.out.print("Enter the age:");
12        int age = sc.nextInt(); // Reads integer value
13

```

```

14     System.out.println("Hello! " + name + ", you are " + age + " "
15         old.);
16
17     sc.close(); // Good practice: close scanner
18 }
```

## 2.3 Output Methods

**Theory:** Java provides three main output methods through the `System.out` object:

- `print()`: Displays text without adding a newline
- `println()`: Displays text and adds a newline at the end
- `printf()`: Formatted output similar to C (not shown in examples)

The `+` operator concatenates strings and converts other types to strings automatically.

## 3 Data Types

### 3.1 Understanding Type Systems

**Theory:** Java is a statically-typed language, meaning every variable must be declared with a data type before use. This allows the compiler to check for type errors at compile-time, making programs more reliable. Java has two categories of data types: primitive types and reference types.

**Primitive Types** are basic data types built into the language. They hold simple values directly in memory and are not objects. Java has 8 primitive types.

**Reference Types** are objects created from classes. They store references (memory addresses) to the actual objects rather than the values themselves.

### 3.2 Primitive Data Types

**Theory:** Each primitive type has a specific size (measured in bits/bytes) and range of values it can store. Choosing the right type depends on the range of values needed and memory efficiency considerations.

Type	Size	Range	Usage
byte	8 bits	-128 to 127	Small integers, saving memory
short	16 bits	-32,768 to 32,767	Larger integers, still memory-conscious
int	32 bits	$-2^{31}$ to $2^{31} - 1$	Default integer type
long	64 bits	$-2^{63}$ to $2^{63} - 1$	Very large integers, suffix with L
float	32 bits	$\pm 3.4 \times 10^{38}$	Decimal numbers, suffix with f
double	64 bits	$\pm 1.7 \times 10^{308}$	Default decimal type
char	16 bits	0 to 65,535	Single Unicode characters
boolean	1 bit	true or false	Logical values

**Example:**

```

1 package DAY2;
2
3 public class datatypes {
4     public static void main(String[] args) {
```

```

5   // Primitive data types
6   int a = 10;           // Most common integer type
7   short b = 1;          // Smaller range, saves memory
8   long c = 748934678;    // For large numbers
9   float d = 3;          // Single-precision decimal
10  double e = 3.542;      // Double-precision decimal (default)
11  char f = 'f';          // Single character in single quotes
12  boolean g = true;       // true or false only
13
14  System.out.println(d);
15 }
16 }
```

### 3.3 Type Casting

**Theory:** Type casting is the process of converting a value from one data type to another. There are two types:

#### 1. Implicit Casting (Widening Conversion):

- Automatic conversion by compiler
- From smaller to larger data type
- No data loss occurs
- Order: byte → short → int → long → float → double
- Also: char → int

#### 2. Explicit Casting (Narrowing Conversion):

- Manual conversion using cast operator
- From larger to smaller data type
- Potential data loss (decimal parts truncated)
- Syntax: (targetType) value
- Programmer takes responsibility for potential data loss

#### Example:

```

1 // Implicit Casting (Widening)
2 int num = 1;
3 double y = num; // int automatically converted to double
4 System.out.println(y); // Output: 1.0
5
6 // Explicit Casting (Narrowing)
7 double x = 3.452;
8 int z = (int) x; // Manually cast double to int
9 System.out.println(z); // Output: 3 (decimal part lost)
```

### 3.4 The final Keyword

**Theory:** The `final` keyword is a modifier that makes a variable constant. Once a final variable is assigned a value, it cannot be reassigned. This is useful for defining constants that should never change throughout program execution, such as mathematical constants, configuration values, or fixed limits.

#### Benefits:

- Prevents accidental modification
- Makes code more readable and maintainable
- Compiler can optimize code better
- Convention: use UPPER\_CASE names for final variables

#### Example:

```
1 final double pi = 3.14;
2 System.out.println(pi);
3 // pi = 3.15; // COMPILE ERROR: Cannot assign value to final variable
```

## 4 Control Structures

### 4.1 Understanding Control Flow

**Theory:** Control structures determine the order in which statements are executed in a program. By default, Java executes statements sequentially from top to bottom. Control structures allow us to alter this flow based on conditions (decision-making) or repeat code blocks (iteration).

#### Types of Control Structures:

1. **Sequential:** Default top-to-bottom execution
2. **Selection:** Decision-making (if, if-else, switch)
3. **Iteration:** Repetition (for, while, do-while loops)

### 4.2 If-Else Statements

**Theory:** The if-else statement evaluates a boolean expression and executes different code blocks based on whether the condition is true or false. The condition must evaluate to a boolean value. Java supports:

- **Simple if:** Executes block only if condition is true
- **if-else:** Two-way decision
- **if-else-if ladder:** Multiple conditions checked sequentially
- **Nested if:** If statements inside other if statements

#### Comparison Operators:

- `==`: Equal to
- `!=`: Not equal to

- <: Less than
- >: Greater than
- <=: Less than or equal to
- >=: Greater than or equal to

**Example:**

```

1 package DAY3;
2
3 public class conditions {
4     public static void main(String[] args) {
5         // Basic if-else condition
6         int a = 10;
7         if (a < 5) {
8             System.out.println("True");
9         } else {
10            System.out.println("False");
11        }
12
13        // If-else-if ladder for multiple conditions
14        double temp = 28.8;
15        if (temp > 32) {
16            System.out.println("High");
17        } else if (temp == 32) {
18            System.out.println("Room temperature");
19        } else if (temp < 32) {
20            System.out.println("Low");
21        } else {
22            System.out.println("Invalid");
23        }
24    }
25 }
```

### 4.3 Nested Conditions

**Theory:** Nested conditions occur when an if statement is placed inside another if statement. This allows for more complex decision-making where multiple conditions must be evaluated in a hierarchical manner. Each inner condition is only evaluated if its outer condition is true.

**Use Cases:**

- Multiple criteria must all be true (AND logic)
- Sequential validation steps
- Complex business rules

**Best Practice:** Keep nesting to 2-3 levels maximum for readability. Consider using logical operators (&&, ——) or switch statements for deeply nested conditions.

**Example:**

```

1 // Nested conditions for age verification
2 int age = 21;
3 boolean id = false;
4
5 if (age >= 18) {
```

```

6 // First condition: Check age
7 if (id == true) {
8     // Second condition: Check ID
9     System.out.println("Approved");
10 } else {
11     System.out.println("ID Required");
12 }
13 } else {
14     System.out.println("Not Allowed");
15 }

```

## 4.4 Switch Statement

**Theory:** The switch statement provides a cleaner alternative to multiple if-else-if statements when comparing a single variable against multiple constant values. It's more readable and efficient when you have many conditions checking the same variable.

### Traditional vs Arrow Syntax:

- **Traditional:** Uses `case:` with `break` statements
- **Arrow (Java 12+):** Uses `case ->` without `break` needed

### Key Points:

- Case values must be constants (compile-time known)
- Supports int, byte, short, char, String, and enums
- Arrow syntax prevents fall-through automatically
- Default case handles unmatched values (optional but recommended)

### Example:

```

1 String day = "Wed";
2 switch (day) {
3     case "mon" -> System.out.println("Day 1");
4     case "Tue" -> System.out.println("Day 2");
5     case "Wed" -> System.out.println("Day 3");
6     default -> System.out.println("Invalid day");
7 }

```

## 5 Loops

### 5.1 Understanding Iteration

**Theory:** Loops allow us to execute a block of code repeatedly without writing the same code multiple times. This is fundamental to programming as many tasks involve repetitive operations. Java provides three types of loops:

1. **for loop:** When you know the number of iterations in advance
2. **while loop:** When iterations depend on a condition (not shown in examples)
3. **do-while loop:** When code must execute at least once (not shown in examples)
4. **enhanced for loop:** For iterating through arrays/collections

## 5.2 For Loop Structure

**Theory:** The for loop has three parts:

1. **Initialization:** Executed once at the start
2. **Condition:** Checked before each iteration
3. **Update:** Executed after each iteration

Syntax: `for (initialization; condition; update) { statements }`

**Example - Basic Square:**

```

1 package DAY4;
2
3 public class Loops {
4     public static void main(String[] args) {
5         // 5x5 Square pattern
6         for (int i = 1; i <= 5; i++) {
7             for (int j = 1; j <= 5; j++) {
8                 System.out.print("* ");
9             }
10            System.out.println(); // New line after each row
11        }
12    }
13}
```

## 5.3 Pattern Programming

**Theory:** Pattern programming helps understand nested loops and control flow. The outer loop typically controls rows, while inner loop(s) control columns or elements within each row.

**Pattern 1: Right Triangle**

**Logic:** For row  $i$ , print  $i$  stars. The number of stars increases with each row.

**Example:**

```

1 System.out.println("=====");
2 for (int i = 1; i <= 5; i++) {
3     for (int j = 1; j <= i; j++) {
4         System.out.print("* ");
5     }
6     System.out.println();
7 }
// Output:
// *
// * *
// * * *
// * * * *
// * * * * *
```

**Pattern 2: Inverted Right Triangle**

**Logic:** Start with maximum stars and decrease by one each row.

**Example:**

```

1 System.out.println("=====");
2 for (int i = 5; i >= 1; i--) {
```

```

3     for (int j = 1; j <= i; j++) {
4         System.out.print("* ");
5     }
6     System.out.println();
7 }
// Output:
// * * * *
// * * *
// * *
// *
// *

```

**Pattern 3: Pyramid****Logic:** Combine spaces and stars. For row  $i$ : print  $(n - i)$  spaces, then  $i$  stars.**Example:**

```

1 System.out.println("=====");
2 for (int i = 1; i <= 7; i++) {
3     // Print leading spaces
4     for (int s = 1; s <= 7 - i; s++) {
5         System.out.print(" ");
6     }
7     // Print stars
8     for (int j = 1; j <= i; j++) {
9         System.out.print("* ");
10    }
11    System.out.println();
12 }
// Output forms a pyramid shape

```

**Pattern 4: Inverted Pyramid****Example:**

```

1 System.out.println("=====");
2 for (int i = 7; i >= 1; i--) {
3     for (int s = 1; s <= 7 - i; s++) {
4         System.out.print(" ");
5     }
6     for (int j = 1; j <= i; j++) {
7         System.out.print("* ");
8     }
9     System.out.println();
10 }

```

**Pattern 5: Number Triangle****Logic:** Instead of stars, print column numbers.**Example:**

```

1 System.out.println("=====");
2 for (int i = 1; i <= 5; i++) {
3     for (int j = 1; j <= i; j++) {
4         System.out.print(j + " ");
5     }
6     System.out.println();
7 }
// Output:

```

```

9 // 1
10 // 1 2
11 // 1 2 3
12 // 1 2 3 4
13 // 1 2 3 4 5

```

**Pattern 6: Sequential Numbers****Logic:** Use a counter variable that increments across all iterations.**Example:**

```

1 System.out.println("=====");
2 int num = 1;
3 for (int i = 1; i <= 6; i++) {
4     for (int j = 1; j <= i; j++) {
5         System.out.print(num++ + " "); // Post-increment
6     }
7     System.out.println();
8 }
9 // Output:
10 // 1
11 // 2 3
12 // 4 5 6
13 // 7 8 9 10
14 // ...

```

**Pattern 7: Hollow Rectangle****Logic:** Print stars only on borders (first/last row or first/last column).**Example:**

```

1 System.out.println("=====");
2 int n = 5;
3 for (int i = 1; i <= n; i++) {
4     for (int j = 1; j <= n; j++) {
5         if (i == 1 || i == n || j == 1 || j == n) {
6             System.out.print("* ");
7         } else {
8             System.out.print("  "); // Two spaces
9         }
10    }
11    System.out.println();
12 }
13 // Output: Rectangle with hollow center

```

**Pattern 8: Descending Number Triangle****Example:**

```

1 System.out.println("=====");
2 for (int i = 7; i >= 1; i--) {
3     for (int j = 1; j <= i; j++) {
4         System.out.print(j + " ");
5     }
6     System.out.println();
7 }

```

## 6 Arrays and Strings

### 6.1 Understanding Arrays

**Theory:** An array is a container object that holds a fixed number of values of a single type. Arrays are reference types in Java, even when they store primitive types. Once created, the array size is fixed and cannot be changed.

#### Key Characteristics:

- **Fixed Size:** Declared at creation, cannot be modified
- **Zero-Indexed:** First element at index 0, last at length-1
- **Homogeneous:** All elements must be same type
- **Contiguous Memory:** Elements stored in consecutive memory locations
- **Fast Access:** O(1) time complexity for index-based access

#### Array Declaration Methods:

1. Declare then assign: `int[] arr = new int[5];`
2. Direct initialization: `int[] arr = {1, 2, 3, 4, 5};`

#### Example:

```

1 package DAY5;
2
3 public class arrayNstrings {
4     public static void main(String[] args) {
5         // Method 1: Declare with size, then assign
6         int[] arr = new int[5];
7         arr[0] = 1;
8         arr[1] = 2;
9         arr[2] = 3;
10        arr[3] = 4;
11        arr[4] = 5;
12
13        // Enhanced for loop (for-each)
14        for (int x : arr) {
15            System.out.print(x + ", ");
16        }
17        System.out.println();
18
19        // Array length property
20        System.out.println(arr.length); // Output: 5
21
22        // Method 2: Direct initialization
23        double[] x = {1.1, 2.2, 3.3, 4.4, 5.5};
24        for (double y : x) {
25            System.out.print(y + ", ");
26        }
27        System.out.println();
28    }
29}
```

## 6.2 Enhanced For Loop

**Theory:** The enhanced for loop (for-each loop) provides a cleaner syntax for iterating through arrays and collections. It automatically handles iteration without needing index variables.

### Advantages:

- More readable and less error-prone
- No index management needed
- Cannot accidentally go out of bounds

### Limitations:

- Cannot modify array elements (read-only)
- Cannot access index during iteration
- Only forward iteration

## 6.3 Understanding Strings

**Theory:** Strings are objects that represent sequences of characters. In Java, strings are immutable, meaning once created, their content cannot be changed. Any operation that appears to modify a string actually creates a new string object.

### String Immutability:

- String objects cannot be modified after creation
- Methods like `concat()`, `substring()` return new strings
- Original string remains unchanged
- Enhances security, thread-safety, and allows string pooling

**String Pool:** Java maintains a special memory region called the string pool where string literals are stored. If two string literals have the same value, they reference the same object in the pool.

### Example:

```

1 // String immutability demonstration
2 String name = "Manoj";
3 name.concat("S"); // Creates new string but doesn't assign
4 System.out.println(name); // Output: "Manoj" (unchanged)
5
6 // Proper concatenation
7 String s = "Hello";
8 s = s + " World"; // Creates new string and assigns
9 System.out.println(s); // Output: "Hello World"

```

## 6.4 String Methods

**Theory:** The `String` class provides numerous methods for string manipulation. All methods return new strings or values without modifying the original string.

### Common String Methods:

- `length()`: Returns number of characters

- `charAt(index)`: Returns character at specified index
- `substring(start, end)`: Extracts portion of string (end exclusive)
- `split(delimiter)`: Splits string into array based on delimiter
- `concat(str)`: Joins strings together
- `toLowerCase()`, `toUpperCase()`: Case conversion
- `trim()`: Removes leading/trailing whitespace
- `equals(str)`: Compares string content

**Example:**

```

1 String test = "Java JavaScript Python";
2
3 // Split into array
4 String[] lang = test.split(" ");
5 for (String l : lang) {
6     System.out.print(l + " ");
7 }
8
9 // Character access
10 System.out.println(test.charAt(4)); // Output: 'a'
11
12 // Length
13 System.out.println(test.length()); // Output: 22
14
15 // Substring
16 System.out.println(test.substring(0, 4)); // Output: "Java"
17
18 // Accessing last character
19 String c = "Hello";
20 System.out.println(c.length()); // Output: 5
21 System.out.println(c.charAt(4)); // Output: 'o'
22 System.out.println(test.charAt(test.length() - 1)); // Last char

```

## 6.5 String Iteration

**Theory:** Since strings are sequences of characters, we can iterate through them using loops and the `charAt()` method. This is useful for character-by-character processing.

**Example:**

```

1 String ss = "Code";
2 for (int i = 0; i < ss.length(); i++) {
3     System.out.println(ss.charAt(i));
4 }
5 // Output:
6 // C
7 // o
8 // d
9 // e

```

## 7 Methods (Functions)

### 7.1 Understanding Methods

**Theory:** Methods (also called functions) are reusable blocks of code that perform specific tasks. They help organize code, reduce redundancy, and make programs more modular and maintainable. Methods follow the DRY (Don't Repeat Yourself) principle.

#### Method Structure:

```
accessModifier returnType methodName(parameters) {
    // method body
    return value; // if returnType is not void
}
```

#### Components:

- **Access Modifier:** Controls visibility (public, private, protected, default)
- **Return Type:** Data type of returned value, or void if nothing returned
- **Method Name:** Identifier following camelCase convention
- **Parameters:** Input values (optional)
- **Method Body:** Code to be executed
- **Return Statement:** Returns value to caller (required for non-void)

### 7.2 Methods Without Return Value

**Theory:** Methods with `void` return type perform actions but don't return any value to the caller. They're used for operations like printing, updating state, or performing calculations where the result isn't needed by the caller.

#### Example:

```
1 package DAY6;
2
3 public class MethodsNScope {
4     // Method with no return value
5     static void Sum(int a, int b) {
6         System.out.println("Sum: " + (a + b));
7     }
8
9     public static void main(String[] args) {
10        Sum(8, 4); // Output: Sum: 12
11    }
12 }
```

### 7.3 Methods With Return Value

**Theory:** Methods that return values allow the result of computation to be used elsewhere in the program. The return type must match the type of value being returned. The `return` statement immediately exits the method and sends the value back to the caller.

#### Example:

```

1 static int square(int num) {
2     return num * num;
3 }
4
5 // Usage
6 int ans = square(4);
7 System.out.println("Square: " + ans); // Output: Square: 16

```

## 7.4 Method Overloading

**Theory:** Method overloading is a feature that allows multiple methods in the same class to have the same name but different parameters. This is a form of compile-time polymorphism. The compiler determines which method to call based on the method signature.

**Method Signature:** Consists of method name and parameter list (number, type, and order of parameters).

### Overloading Rules:

- Methods must have different parameter lists
- Can differ in: number of parameters, type of parameters, or order of parameters
- Return type alone is NOT sufficient for overloading
- Access modifiers can be different

### Benefits:

- Improves code readability
- Reduces complexity by using same name for similar operations
- Provides flexibility for different input types

### Example:

```

1 // Overloaded add methods for different types
2 static int add(int a, int b) {
3     return a + b;
4 }
5
6 static double add(double a, double b) {
7     return a + b;
8 }
9
10 static String add(String a, String b) {
11     return a + b;
12 }
13
14 // Overloaded multiply with different parameter counts
15 static void multiply(int a, int b) {
16     System.out.println("Product: " + (a * b));
17 }
18
19 static void multiply(int a, int b, int c) {
20     System.out.println("Product: " + (a * b * c));
21 }
22

```

```

23 // Usage
24 System.out.println("Int Add: " + add(5, 3));           // Calls int version
25 System.out.println("Double Add: " + add(3.14, 71.56)); // Calls double
26 System.out.println("String Add: " + add("Manoj", " S")); // Calls
27     String
28 multiply(5, 8);           // Calls two-parameter version
29 multiply(6, 5, 4);       // Calls three-parameter version

```

## 7.5 Variable Scope

**Theory:** Scope refers to the region of code where a variable is accessible. Java has several types of scope:

- **Class/Static Scope:** Variables declared at class level with `static` keyword. Accessible throughout the class.
- **Instance Scope:** Non-static variables at class level. Tied to object instances.
- **Method/Local Scope:** Variables declared inside methods. Only accessible within that method.
- **Block Scope:** Variables declared inside blocks (if, for, etc.). Only accessible within that block.

**Shadowing:** When a local variable has the same name as a class variable, the local variable shadows (hides) the class variable within its scope.

**Example:**

```

1 static int count = 8; // Class-level scope
2
3 static void check() {
4     int count = 9;      // Method-level scope (shadows class variable)
5     System.out.println(count); // Prints 9 (local variable)
6 }
7
8 public static void main(String[] args) {
9     check();           // Output: 9
10    System.out.println(count); // Output: 8 (class variable)
11 }

```

## 8 Object-Oriented Programming Fundamentals

### 8.1 Understanding OOP

**Theory:** Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects" which contain data (attributes/fields) and code (methods). OOP organizes software design around data and objects rather than functions and logic.

**Four Pillars of OOP:**

1. **Encapsulation:** Bundling data and methods together, hiding internal details
2. **Inheritance:** Creating new classes from existing ones, promoting code reuse
3. **Polymorphism:** Objects taking multiple forms, same interface for different types
4. **Abstraction:** Hiding complex implementation details, showing only essential features

## 8.2 Classes and Objects

### Theory:

- **Class:** A blueprint or template that defines the structure and behavior of objects. It's a user-defined data type.
- **Object:** An instance of a class. It's a real entity that exists in memory and has state and behavior.

**Relationship:** Class is to Object as Blueprint is to House. One class can create many objects.

### Class Components:

- **Fields/Attributes:** Variables that hold object state
- **Methods:** Functions that define object behavior
- **Constructors:** Special methods for object initialization

### Example:

```

1 package DAY7;
2
3 // Class definition
4 class Student {
5     int id;          // Instance variable (attribute)
6     String name;    // Instance variable (attribute)
7
8     // Method to display student information
9     void display() {
10         System.out.println("ID: " + id + " | Name: " + name);
11     }
12 }
13
14 public class OOPPart1 {
15     public static void main(String[] args) {
16         // Creating objects (instances of Student class)
17         Student s1 = new Student();
18         s1.id = 17;
19         s1.name = "Manoj S";
20         s1.display(); // Output: ID: 17 | Name: Manoj S
21
22         Student s2 = new Student();
23         s2.id = 07;
24         s2.name = "Speed";
25         s2.display(); // Output: ID: 7 | Name: Speed
26     }
27 }
```

## 8.3 Constructors

**Theory:** A constructor is a special method that is automatically called when an object is created. Its purpose is to initialize the object's state.

### Constructor Characteristics:

- Same name as the class

- No return type (not even void)
- Called automatically when object is created using `new`
- Can be overloaded (multiple constructors with different parameters)
- If no constructor is defined, Java provides a default no-argument constructor

### Types of Constructors:

1. **Default Constructor:** No parameters, provided by Java if you don't write any
2. **Parameterized Constructor:** Takes parameters to initialize object with specific values
3. **Copy Constructor:** Creates object using another object of same class

### Example:

```

1 class Car {
2     int num;
3     String brand;
4
5     // Parameterized constructor
6     Car(int num, String brand) {
7         this.num = num;           // 'this' refers to current object
8         this.brand = brand;
9     }
10
11    void show() {
12        System.out.println("Number: " + num + " | Brand: " + brand);
13    }
14 }
15
16 // Usage
17 Car c1 = new Car(54, "Porsche"); // Constructor called automatically
18 Car c2 = new Car(87, "BMW");
19 c1.show(); // Output: Number: 54 | Brand: Porsche
20 c2.show(); // Output: Number: 87 | Brand: BMW

```

## 8.4 The 'this' Keyword

**Theory:** The `this` keyword is a reference to the current object instance. It's used to distinguish between instance variables and parameters/local variables when they have the same name.

### Uses of 'this':

- Refer to current class instance variables
- Pass current object as parameter to other methods
- Call other constructors from a constructor (constructor chaining)
- Return current object from a method

## 8.5 Encapsulation

**Theory:** Encapsulation is the bundling of data (fields) and methods that operate on that data within a single unit (class), and restricting direct access to some of the object's components. This is achieved using access modifiers.

### Benefits:

- **Data Hiding:** Internal representation hidden from outside
- **Increased Flexibility:** Internal implementation can change without affecting users
- **Reusability:** Encapsulated code can be reused
- **Testing:** Easier to test individual components
- **Validation:** Setters can validate data before setting

### Access Modifiers:

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

### Example:

```

1 class Student {
2     private int id;           // Private: can't be accessed directly
3     private String name;
4
5     // Constructor
6     Student(int id, String name) {
7         this.id = id;
8         this.name = name;
9     }
10
11    // Getter methods (read access)
12    public int getId() {
13        return id;
14    }
15
16    public String getName() {
17        return name;
18    }
19
20    // Setter method (write access with potential validation)
21    public void setName(String name) {
22        this.name = name;
23    }
24
25    public void display() {
26        System.out.println("Student ID: " + id + " | Student Name: " +
27            name);
28    }
29}
30 public class FStudent {

```

```

31  public static void main(String[] args) {
32      Student s1 = new Student(7, "Manoj");
33
34      // Accessing private fields through getters
35      String name = s1.getName();
36      int id = s1.getId();
37      System.out.println("id: " + id + " | name: " + name);
38
39      // Modifying through setter
40      s1.setName("Manu");
41      s1.display(); // Output: Student ID: 7 | Student Name: Manu
42  }
43 }
```

### Another Encapsulation Example:

```

1  class Stat {
2      private int num;
3      private String val;
4
5      // Setter method
6      public void setvals(int num, String val) {
7          this.num = num;
8          this.val = val;
9      }
10
11     // Getter method
12     public int getnum() {
13         return num;
14     }
15 }
16
17 public class AccessSpec {
18     public static void main(String[] args) {
19         Stat s1 = new Stat();
20         s1.setvals(07, "Manoj");
21         int num = s1.getnum();
22         System.out.println(num); // Output: 7
23     }
24 }
```

## 9 Inheritance

### 9.1 Understanding Inheritance

**Theory:** Inheritance is a mechanism where a new class (child/subclass/derived class) is created from an existing class (parent/superclass/base class). The child class inherits fields and methods from the parent class and can add its own unique features.

#### Benefits:

- **Code Reusability:** Avoid writing duplicate code
- **Method Overriding:** Child can provide specific implementation
- **Polymorphism:** Objects can be treated as instances of parent class
- **Hierarchical Classification:** Models real-world relationships

### Types of Inheritance:

1. **Single:** One parent, one child
2. **Multilevel:** Chain of inheritance ( $A \rightarrow B \rightarrow C$ )
3. **Hierarchical:** One parent, multiple children
4. **Multiple:** One child, multiple parents (NOT supported in Java through classes, use interfaces)
5. **Hybrid:** Combination of above (NOT supported directly)

**Syntax:** class ChildClass extends ParentClass { }

### Example:

```

1 package DAY8;
2
3 // Parent class (Superclass)
4 class Animal {
5     void sounds() {
6         System.out.println("Animal makes sound");
7     }
8 }
9
10 // Child class (Subclass) - inherits from Animal
11 class Dog extends Animal {
12     void bark() {
13         System.out.println("Dog barks");
14     }
15 }
16
17 // Another inheritance example
18 class Vehicle {
19     void start() {
20         System.out.println("Starting...");
21     }
22 }
23
24 class Car extends Vehicle {
25     void drive() {
26         System.out.println("Car drives...");
27     }
28 }
29
30 public class Classes {
31     public static void main(String[] args) {
32         Dog a1 = new Dog();
33         a1.sounds(); // Inherited method from Animal
34         a1.bark(); // Own method of Dog
35
36         Car bmw = new Car();
37         bmw.start(); // Inherited method from Vehicle
38         bmw.drive(); // Own method of Car
39     }
40 }
```

## 9.2 The super Keyword

**Theory:** The `super` keyword is a reference to the immediate parent class object. It's used to access parent class members (fields and methods) that are hidden by the child class.

**Uses of super:**

- `super.variable`: Access parent class field when child has same-named field
- `super.method()`: Call parent class method when child has overridden it
- `super()`: Call parent class constructor from child constructor (must be first statement)

**Constructor Chaining:** When a child class object is created, the parent class constructor is called first (implicitly or explicitly), then the child constructor executes. This ensures proper initialization of inherited members.

**Example - super with Constructor:**

```

1  class Person {
2      Person(String name) {
3          System.out.println("Name: " + name);
4      }
5  }
6
7  class Employee extends Person {
8      Employee() {
9          super("Manoj S"); // Calls parent constructor
10         // Must be first statement in constructor
11     }
12 }
13
14 // Usage
15 Employee e1 = new Employee(); // Output: Name: Manoj S

```

**Example - super with Fields and Methods:**

```

1  class Check {
2      int num = 10; // Parent class field
3
4      void add(int a, int b) {
5          System.out.println("Sum: " + (a + b));
6      }
7  }
8
9  class Test extends Check {
10     int num = 19; // Child class field (hides parent's num)
11
12     void dis() {
13         System.out.println(num);           // Prints 19 (child's num)
14         System.out.println(super.num);    // Prints 10 (parent's num)
15         super.add(5, 8);                // Calls parent's method
16     }
17 }
18
19 // Usage
20 Test c1 = new Test();
21 c1.dis();
22 // Output:
23 // 19

```

```

24 // 10
25 // Sum: 13

```

### 9.3 Method Overriding

**Theory:** Method overriding occurs when a child class provides a specific implementation of a method that is already defined in its parent class. The overriding method in the child class must have the same name, return type (or covariant), and parameters as the parent method.

#### Key Rules:

- Method signature must be identical to parent method
- Cannot reduce access level (can increase or keep same)
- Return type must be same or subtype (covariant return)
- Cannot override final, static, or private methods
- Use `@Override` annotation for compile-time checking (optional but recommended)

#### Override vs Overload:

- **Overriding:** Same method signature, different classes (parent-child), runtime polymorphism
- **Overloading:** Same method name, different parameters, same class, compile-time polymorphism

#### Example:

```

1 class Shape {
2     void draw() {
3         System.out.println("Drawing any shape");
4     }
5 }
6
7 class Circle extends Shape {
8     @Override // Annotation for clarity and error-checking
9     void draw() {
10         System.out.println("Drawing a circle");
11     }
12 }
13
14 public class SuperKey {
15     public static void main(String[] args) {
16         // Polymorphism: Parent reference, child object
17         Shape s1 = new Circle();
18         s1.draw(); // Calls Circle's draw() method
19                     // Output: Drawing a circle
20     }
21 }

```

## 10 Abstract Classes

### 10.1 Understanding Abstraction

**Theory:** Abstraction is the process of hiding implementation details and showing only essential features to the user. It focuses on what an object does rather than how it does it. Abstract classes are one way to achieve abstraction in Java.

**Abstract Class:** A class declared with the `abstract` keyword that cannot be instantiated directly. It may contain both abstract methods (without implementation) and concrete methods (with implementation).

**Abstract Method:** A method declared without implementation (no body). Child classes must provide implementation for all abstract methods unless they are also abstract.

#### Key Points:

- Cannot create objects of abstract class directly
- Can have constructors (called by child class)
- Can have both abstract and concrete methods
- Can have instance variables and static methods
- Child class must implement all abstract methods or be declared abstract itself
- Provides partial implementation (0-100% abstraction)

#### When to Use Abstract Classes:

- When classes share common code
- When you want to provide default behavior
- When you need constructors or instance variables
- When abstraction level is not 100%

#### Example:

```

1 package DAY8;
2
3 abstract class vech { // Abstract class
4     // Abstract method (no implementation)
5     abstract void hello();
6
7     // Concrete method (with implementation)
8     void fuel() {
9         System.out.println("Fueling... ");
10    }
11 }
12
13 class car extends vech {
14     // Must override abstract method
15     @Override
16     void hello() {
17         System.out.println("Hello world!");
18     }
19 }
```

```

21 public class Abstract {
22     public static void main(String[] args) {
23         // vech v = new vech(); // ERROR: Cannot instantiate abstract
24         // class
25
26         vech v1 = new car(); // Parent reference, child object
27         v1.fuel();           // Calls concrete method
28         v1.hello();          // Calls overridden method
29     }
}

```

## 11 Interfaces

### 11.1 Understanding Interfaces

**Theory:** An interface is a reference type in Java that contains only abstract methods (before Java 8) and constants. It's a contract that specifies what a class must do, but not how it does it. Interfaces achieve 100% abstraction.

#### Interface Characteristics:

- All methods are implicitly `public abstract` (before Java 8)
- All variables are implicitly `public static final` (constants)
- Cannot have constructors
- Cannot be instantiated
- Supports multiple inheritance (a class can implement multiple interfaces)
- From Java 8: Can have default and static methods
- From Java 9: Can have private methods

#### Interface vs Abstract Class:

Interface	Abstract Class
100% abstraction (traditionally) Multiple inheritance supported Only constants (public static final) No constructors implements keyword Methods public abstract by default	0-100% abstraction Single inheritance only Can have instance variables Can have constructors extends keyword Can have any access modifier

#### When to Use Interfaces:

- When you need multiple inheritance
- When unrelated classes need common behavior
- To specify a contract without implementation
- For loose coupling between components

#### Example - Basic Interface:

```

1 package DAY8;
2
3 interface animal {
4     void sound(); // Implicitly public abstract
5 }
6
7 class dog implements animal {
8     // Must implement all interface methods
9     public void sound() {
10         System.out.println("Dog barks");
11     }
12 }
13
14 public class Interfaces {
15     public static void main(String[] args) {
16         dog d = new dog();
17         d.sound(); // Output: Dog barks
18     }
19 }
```

## 11.2 Multiple Interface Implementation

**Theory:** Unlike class inheritance where a class can extend only one parent class, a class can implement multiple interfaces. This provides a way to achieve multiple inheritance in Java, allowing a class to inherit behavior from multiple sources.

**Syntax:** class MyClass implements Interface1, Interface2, Interface3 { }

**Example:**

```

1 interface alpha {
2     void show();
3 }
4
5 interface beta {
6     void display();
7 }
8
9 interface gama {
10     int num = 7; // Public static final by default
11 }
12
13 // Implementing multiple interfaces
14 class a implements alpha, beta, gama {
15     public void show() {
16         System.out.println("Alpha");
17     }
18
19     public void display() {
20         System.out.println("Beta");
21     }
22
23     void prints() {
24         System.out.println("Ronaldo: " + num); // Accessing interface
25             constant
26     }
27 }
```

```

28 public class Interfaces {
29     public static void main(String[] args) {
30         a b = new a();
31         b.show();          // Output: Alpha
32         b.display();     // Output: Beta
33         b.prints();      // Output: Ronaldo: 7
34     }
35 }
```

### Real-World Example:

```

1 interface Payment {
2     void pay();
3 }
4
5 class UPI implements Payment {
6     @Override
7     public void pay() {
8         System.out.println("Can proceed the UPI Payment...");
```

## 12 Collections Framework

### 12.1 Understanding Collections

**Theory:** The Java Collections Framework provides a set of classes and interfaces for storing and manipulating groups of objects. It includes implementations of common data structures like lists, sets, and maps.

#### Benefits:

- Dynamic sizing (grow/shrink automatically)
- Provides ready-to-use data structures
- Algorithms for searching, sorting, etc.
- Better performance through optimized implementations
- Interoperability between different collection types

#### Collection Hierarchy:

```

Collection (Interface)
+-- List (Interface) - Ordered, allows duplicates
|   +-- ArrayList - Fast random access
|   +-- LinkedList - Fast insertion/deletion
|   +-- Vector - Synchronized ArrayList
+-- Set (Interface) - No duplicates
```

```

|   +-- HashSet - Fast, unordered
|   +-- TreeSet - Sorted
+-- Queue (Interface) - FIFO operations

```

## 12.2 ArrayList

**Theory:** ArrayList is a resizable array implementation of the List interface. Unlike arrays, ArrayLists can grow and shrink dynamically. Internally, it uses an array that is resized when needed (typically 1.5x capacity when full).

### Characteristics:

- Dynamic size (no fixed capacity)
- Allows duplicate elements
- Maintains insertion order
- Fast random access: O(1) time for get/set
- Slower insertion/deletion in middle: O(n) time
- Not synchronized (not thread-safe)
- Uses generics for type safety

### Common Methods:

- `add(element)`: Add element at end
- `add(index, element)`: Insert at specific position
- `get(index)`: Retrieve element
- `set(index, element)`: Replace element
- `remove(index)`: Remove by index
- `size()`: Get number of elements
- `clear()`: Remove all elements
- `contains(element)`: Check if element exists

### Example:

```

1 package DAY9;
2 import java.util.ArrayList;
3
4 public class Arraylist {
5     public static void main(String[] args) {
6         // Creating ArrayList with generic type Integer
7         ArrayList<Integer> nums = new ArrayList<>();
8
9         // Adding elements
10        nums.add(10);
11        nums.add(20);
12        nums.add(30);
13
14        // Iterating using enhanced for loop
15        for (int i : nums) {

```

```

16         System.out.println(i);
17     }
18
19     // Accessing element by index
20     System.out.println(nums.get(0)); // Output: 10
21 }
22 }
```

## 12.3 HashSet

**Theory:** HashSet is a collection that uses a hash table for storage. It implements the Set interface and does not allow duplicate elements. Elements are stored in no particular order (unordered).

### Characteristics:

- No duplicate elements (automatically ignored)
- Unordered (no guaranteed iteration order)
- Allows one null element
- Fast operations: O(1) average time for add, remove, contains
- Uses hashing mechanism internally
- Not synchronized (not thread-safe)

### How it Works:

1. When adding element, its `hashCode()` is calculated
2. Hash code determines storage location (bucket)
3. If two objects have same hash code, `equals()` is used to check equality
4. If equal, duplicate is not added

### When to Use:

- When you need unique elements
- When order doesn't matter
- When fast lookup is important
- For removing duplicates from a collection

### Example:

```

1 package DAY9;
2 import java.util.HashSet;
3
4 public class HashSet {
5     public static void main(String[] args) {
6         HashSet<Integer> nums = new HashSet<>();
7
8         nums.add(10);
9         nums.add(20);
10        nums.add(10); // Duplicate - will not be added
```

```

11     nums.add(30);
12     nums.add(10); // Duplicate - will not be added
13
14     // Only unique elements are stored
15     for (int i : nums) {
16         System.out.println(i);
17     }
18     // Output: 10, 20, 30 (order may vary)
19 }
20 }
```

## 12.4 HashMap

**Theory:** HashMap is a collection that stores key-value pairs. It implements the Map interface and uses hashing to store elements. Each key is unique, but values can be duplicate.

### Characteristics:

- Stores key-value pairs
- Keys are unique (duplicates override previous value)
- Values can be duplicate
- Allows one null key and multiple null values
- Unordered (no guaranteed iteration order)
- Fast operations: O(1) average time for get/put
- Not synchronized (not thread-safe)

### Common Methods:

- `put(key, value)`: Insert/update entry
- `get(key)`: Retrieve value for key
- `remove(key)`: Delete entry
- `containsKey(key)`: Check if key exists
- `containsValue(value)`: Check if value exists
- `keySet()`: Get all keys
- `values()`: Get all values
- `entrySet()`: Get all key-value pairs

### Example:

```

1 package DAY9;
2 import java.util.HashMap;
3
4 public class hashmap {
5     public static void main(String[] args) {
6         HashMap<String, Integer> emps = new HashMap<>();
7
8         // Adding key-value pairs
```

```

9     emps.put("Java", 90);
10    emps.put("DSA", 85);
11
12    // Retrieving value by key
13    int val = emps.get("Java");
14    System.out.println(val); // Output: 90
15
16 }
}

```

## 12.5 TreeMap

**Theory:** TreeMap is a sorted map implementation based on a Red-Black tree. Keys are automatically sorted in natural order (or by a provided comparator). It provides guaranteed  $\log(n)$  time cost for basic operations.

### Characteristics:

- Keys are sorted (natural order or custom comparator)
- No null keys allowed (throws NullPointerException)
- Null values are allowed
- Slower than HashMap:  $O(\log n)$  time for get/put
- Not synchronized (not thread-safe)
- Maintains sorted order during iteration

### When to Use:

- When you need sorted keys
- When you need range operations (subMap, headMap, tailMap)
- When you need first/last key access

### Example:

```

1 package DAY9;
2 import java.util.TreeMap;
3
4 public class Treemap {
5     public static void main(String[] args) {
6         TreeMap<String, Integer> emps = new TreeMap<>();
7
8         emps.put("Manoj", 19);
9         emps.put("Speed", 21);
10        emps.put("Ronaldo", 41);
11
12        // Keys are automatically sorted alphabetically
13        System.out.println(emps);
14        // Output: {Manoj=19, Ronaldo=41, Speed=21}
15    }
16 }

```

## 13 Exception Handling

### 13.1 Understanding Exceptions

**Theory:** An exception is an event that disrupts the normal flow of program execution. Exception handling is a mechanism to handle runtime errors, maintaining the normal flow of the application.

#### Exception Hierarchy:

```
Throwable
+-- Error (System errors, not handled by programs)
|   +-- OutOfMemoryError, StackOverflowError, etc.
+-- Exception (Can be handled)
    +-- Checked Exceptions (Compile-time checking)
        |   +-- IOException, SQLException, etc.
    +-- RuntimeException (Unchecked Exceptions)
        +-- NullPointerException, ArithmeticException, etc.
```

#### Checked vs Unchecked Exceptions:

- **Checked:** Checked at compile-time, must be handled or declared
- **Unchecked:** Not checked at compile-time, occur during runtime

#### Why Handle Exceptions:

- Prevent program crash
- Provide meaningful error messages
- Separate error handling from normal code
- Enable graceful recovery from errors

### 13.2 Try-Catch-Finally

**Theory:** The try-catch-finally block is used to handle exceptions gracefully.

#### Structure:

- **try:** Contains code that might throw an exception
- **catch:** Handles specific exception types
- **finally:** Always executes, regardless of exception (cleanup code)

**Multiple Catch Blocks:** You can have multiple catch blocks for different exception types. They are checked in order, so put more specific exceptions before general ones.

#### Example:

```
1 package DAY9;
2 import java.util.Scanner;
3
4 public class Exceptions {
5     public static void main(String[] args) {
6         Scanner sc = new Scanner(System.in);
7
8         try {
9             int ans = 5 / 1; // No exception here
```

```

10         System.out.println("Division: " + ans);
11     } catch (ArithmecticException e) {
12         System.out.println("Error Occurred");
13     } finally {
14         System.out.println("Program Ended");
15         // Always executes, even if exception occurs
16     }
17
18     sc.close();
19 }
20 }
```

### 13.3 Custom Exceptions

**Theory:** Java allows you to create your own exception classes by extending the `Exception` class (for checked exceptions) or `RuntimeException` (for unchecked exceptions). Custom exceptions make code more readable and provide domain-specific error handling.

#### Steps to Create Custom Exception:

1. Create a class that extends `Exception` (or `RuntimeException`)
2. Create constructors (typically one with `String` message parameter)
3. Call `super(message)` to set exception message

#### When to Use:

- When built-in exceptions don't adequately describe the error
- For business logic violations
- To add additional information or behavior to exceptions

#### Example:

```

1 // Custom exception class
2 class AgeException extends Exception {
3     AgeException(String msg) {
4         super(msg); // Call parent constructor
5     }
6 }
7
8 // Using custom exception
9 try {
10     System.out.println("Enter the age: ");
11     int age = sc.nextInt();
12
13     if (age < 18) {
14         throw new AgeException("Not Eligible"); // Throw custom
15             exception
16     }
17 } catch (AgeException e) {
18     System.out.println(e.getMessage()); // Output: Not Eligible
19 }
```

## 14 File Handling

### 14.1 Understanding File I/O

**Theory:** File handling involves reading from and writing to files. Java provides several classes in the `java.io` package for file operations. Files can store data permanently, unlike variables which exist only during program execution.

#### File Operation Categories:

1. **File Operations:** Create, delete, check existence
2. **Writing:** Store data to files
3. **Reading:** Retrieve data from files

#### Common Classes:

- **File:** Represents file/directory pathname
- **FileWriter:** For writing character files
- **FileReader:** For reading character files
- **BufferedWriter:** Efficient writing with buffering
- **BufferedReader:** Efficient reading with buffering

### 14.2 Basic File Operations

**Theory:** The `File` class represents file or directory paths. It provides methods to check file properties and perform operations like create, delete, etc.

#### Example:

```

1 package DAY10;
2 import java.io.*;
3
4 public class Files {
5     public static void main(String[] args) throws Exception {
6         // Create File object
7         File fc = new File("data.txt");
8
9         // Create new file
10        fc.createNewFile();
11
12        // File properties
13        System.out.println("Name: " + fc.getName());
14        System.out.println("Exists: " + fc.exists());
15        System.out.println("Path: " + fc.getAbsolutePath());
16    }
17}
```

### 14.3 Writing to Files

**Theory:** `FileWriter` writes characters to files. `BufferedWriter` provides buffering for efficient writing. The second parameter in constructor determines append (true) or overwrite (false) mode.

#### FileWriter vs BufferedWriter:

- **FileWriter:** Direct writing, slower for large data
- **BufferedWriter:** Uses buffer, faster, provides `newLine()` method

**Example:**

```

1 // FileWriter - basic writing
2 FileWriter fw = new FileWriter("data.txt", true); // true = append
   mode
3 fw.write("Hello World!!");
4 fw.close(); // Important: close to flush and release resources
5
6 // BufferedWriter - efficient writing
7 BufferedWriter bw = new BufferedWriter(new FileWriter("data.txt", true))
   );
8 bw.newLine(); // Add newline
9 bw.write("Manoj S");
10 bw.newLine();
11 bw.write("Wassup");
12 bw.close();

```

## 14.4 Reading from Files

**Theory:** BufferedReader reads text from input stream efficiently. The `readLine()` method reads one line at a time and returns null when end of file is reached.

**Example:**

```

1 BufferedReader bb = new BufferedReader(new FileReader("data.txt"));
2 String l;
3
4 // Read line by line until end of file
5 while ((l = bb.readLine()) != null) {
6     System.out.println(l);
7 }
8 bb.close();

```

## 14.5 Working with JSON

**Theory:** JSON (JavaScript Object Notation) is a lightweight data-interchange format. Java doesn't have built-in JSON support, so we use external libraries like org.json.

**JSON Structure:**

- **JSONObject:** Represents key-value pairs {}
- **JSONArray:** Represents ordered list []

**Writing JSON - Example:**

```

1 package DAY10;
2 import org.json.*;
3 import java.io.*;
4
5 public class jsonD {
6     public static void main(String[] args) {
7         try {
8             // Create JSON array

```

```

9         JSONArray arr = new JSONArray();
10
11     // Create JSON objects
12     JSONObject js = new JSONObject();
13     js.put("id", 1);
14     js.put("name", "Manoj S");
15     js.put("Course", "Java");
16
17     JSONObject js1 = new JSONObject();
18     js1.put("id", 2);
19     js1.put("name", "Manu");
20     js1.put("Course", "python");
21
22     JSONObject js2 = new JSONObject();
23     js2.put("id", 3);
24     js2.put("name", "speed");
25     js2.put("Course", "go");
26
27     // Add objects to array
28     arr.put(js);
29     arr.put(js1);
30     arr.put(js2);
31
32     // Write to file with indentation
33     BufferedWriter os = new BufferedWriter(
34         new FileWriter("data.json", true)
35     );
36     os.write(arr.toString(4)); // 4 = indentation spaces
37     System.out.println(js1.toString(4));
38     os.close();
39 } catch (Exception e) {
40     e.printStackTrace();
41 }
42 }
43 }
```

### Reading JSON - Example:

```

1 package DAY10;
2 import java.nio.file.*;
3 import org.json.JSONArray;
4 import org.json.JSONObject;
5
6 public class Rjson {
7     public static void main(String[] args) {
8         try {
9             // Read entire file content
10            String content = new String(
11                java.nio.file.Files.readAllBytes(
12                    java.nio.file.Paths.get("data.json")
13                )
14            );
15
16            // Parse JSON
17            JSONArray arr = new JSONArray(content);
18
19            // Iterate through array
20            for (int i = 0; i < arr.length(); i++) {
```

```

21         JSONObject obj = arr.getJSONObject(i);
22         System.out.println(
23             obj.getInt("id") + " " +
24             obj.getString("name")
25         );
26     }
27 } catch (Exception e) {
28     System.out.println("Error");
29 }
30 }
31 }
```

## 14.6 Serialization

**Theory:** Serialization is the process of converting an object into a byte stream for storage or transmission. Deserialization is the reverse process of reconstructing the object from the byte stream.

### Key Concepts:

- Class must implement `Serializable` interface (marker interface)
- `ObjectOutputStream`: Writes objects to stream
- `ObjectInputStream`: Reads objects from stream
- `transient` keyword: Marks fields to skip during serialization
- `serialVersionUID`: Version control for serialized classes

### Use Cases:

- Saving object state to files
- Sending objects over network
- Caching objects
- Deep copying objects

### Example:

```

1 package DAY10;
2 import java.io.*;
3
4 // Class must implement Serializable
5 class Student implements Serializable {
6     String name;
7     int id;
8
9     Student(String name, int id) {
10         this.name = name;
11         this.id = id;
12     }
13 }
14
15 public class serialziation {
16     public static void main(String[] args) {
17         try {
```

```

18     // Serialization - Writing object to file
19     ObjectOutputStream oos = new ObjectOutputStream(
20         new FileOutputStream("student.dat")
21     );
22     oos.writeObject(new Student("Manoj S", 7));
23     oos.close();
24
25     // Deserialization - Reading object from file
26     ObjectInputStream ois = new ObjectInputStream(
27         new FileInputStream("student.dat")
28     );
29     Student s = (Student) ois.readObject(); // Cast required
30     ois.close();
31
32     System.out.println(s.id + " " + s.name);
33     // Output: 7 Manoj S
34 } catch (Exception e) {
35     System.out.println("Error");
36 }
37 }
38 }
```

## 15 Best Practices and Coding Standards

### 15.1 Naming Conventions

**Theory:** Consistent naming makes code more readable and maintainable. Java has established conventions that developers should follow.

Element	Convention	Example
Classes	PascalCase	StudentRecord, FileHandler
Methods	camelCase	getName(), calculateTotal()
Variables	camelCase	firstName, totalAmount
Constants	UPPER_SNAKE_CASE	MAX_SIZE, PI_VALUE
Packages	lowercase	com.company.project
Interfaces	PascalCase (often -able)	Runnable, Serializable

### 15.2 Code Organization

#### Best Practices:

- One public class per .java file
- Class name must match filename
- Group related classes in packages
- Order class members: fields, constructors, methods
- Keep methods short and focused (single responsibility)
- Use meaningful names that describe purpose
- Add comments for complex logic, not obvious code

### 15.3 Memory Management

#### Best Practices:

- Close resources (Scanner, FileWriter, etc.) after use
- Use try-with-resources for automatic closing
- Avoid memory leaks by removing unused object references
- Set large objects to null when no longer needed
- Be careful with static variables (live for entire program)

#### Try-with-Resources Example:

```
1 // Automatic resource management
2 try (Scanner sc = new Scanner(System.in)) {
3     String input = sc.nextLine();
4     // Scanner automatically closed
5 }
```

### 15.4 Exception Handling Guidelines

- Catch specific exceptions before general ones
- Don't catch Exception or Throwable unless necessary
- Always provide meaningful error messages
- Log exceptions for debugging
- Clean up resources in finally or use try-with-resources
- Don't use exceptions for control flow

### 15.5 OOP Principles

#### SOLID Principles:

- Single Responsibility: Class should have one reason to change
- Open/Closed: Open for extension, closed for modification
- Liskov Substitution: Subclass objects should be substitutable
- Interface Segregation: Many specific interfaces better than one general
- Dependency Inversion: Depend on abstractions, not concretions

## 16 Quick Reference Guide

### 16.1 Common Syntax Patterns

```
1 // Class Definition
2 public class ClassName {
3     // Fields
4     private int field;
5
6     // Constructor
7     public ClassName(int field) {
8         this.field = field;
9     }
10
11    // Method
12    public int getField() {
13        return field;
14    }
15 }
16
17 // Loops
18 for (int i = 0; i < n; i++) { }
19 for (Type item : collection) { }
20 while (condition) { }
21 do { } while (condition);
22
23 // Conditionals
24 if (condition) { } else { }
25 switch (value) {
26     case 1 -> action1;
27     case 2 -> action2;
28     default -> defaultAction;
29 }
30
31 // Exception Handling
32 try {
33     // risky code
34 } catch (ExceptionType e) {
35     // handle exception
36 } finally {
37     // cleanup
38 }
39
40 // Array Declaration
41 int[] arr = new int[size];
42 int[] arr = {1, 2, 3};
43
44 // ArrayList
45 ArrayList<Type> list = new ArrayList<>();
46 list.add(element);
47 Type item = list.get(index);
48
49 // HashMap
50 HashMap<KeyType, ValueType> map = new HashMap<>();
51 map.put(key, value);
52 ValueType val = map.get(key);
```

## 16.2 Essential Imports

```

1 // Input/Output
2 import java.util.Scanner;
3 import java.io.*;
4
5 // Collections
6 import java.util.ArrayList;
7 import java.util.HashMap;
8 import java.util.HashSet;
9 import java.util.TreeMap;
10
11 // JSON (external library)
12 import org.json.*;
13
14 // File Operations
15 import java.nio.file.*;

```

## 16.3 Common String Methods

Method	Description
length()	Returns number of characters
charAt(index)	Returns character at index
substring(start, end)	Extracts portion (end exclusive)
split(delimiter)	Splits into array
toLowerCase()	Converts to lowercase
toUpperCase()	Converts to uppercase
trim()	Removes whitespace from ends
equals(str)	Compares content
contains(str)	Checks if contains substring
indexOf(str)	Finds first occurrence index
replace(old, new)	Replaces all occurrences

## 16.4 Common ArrayList Methods

Method	Description
add(element)	Adds element at end
add(index, element)	Inserts at position
get(index)	Returns element at index
set(index, element)	Replaces element
remove(index)	Removes element
size()	Returns number of elements
clear()	Removes all elements
contains(element)	Checks if contains
isEmpty()	Checks if empty
indexOf(element)	Finds index of element

## 16.5 Access Modifier Summary

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

## 17 Learning Path and Next Steps

### 17.1 Topics Covered

You have now learned:

1. Java basics: syntax, I/O, data types
2. Control structures: conditionals, loops
3. Arrays and Strings
4. Methods and scope
5. Object-Oriented Programming: classes, objects, inheritance
6. Advanced OOP: abstraction, interfaces
7. Collections Framework
8. Exception handling
9. File I/O and JSON
10. Serialization

### 17.2 Recommended Next Steps

#### Intermediate Topics to Explore:

- Lambda expressions and functional programming
- Stream API for collection processing
- Multithreading and concurrency
- Java 8+ features (Optional, CompletableFuture)
- Design patterns (Singleton, Factory, Observer, etc.)
- JDBC for database connectivity
- Unit testing with JUnit

#### Advanced Topics:

- Spring Framework for enterprise applications
- Java EE / Jakarta EE
- Microservices architecture
- Performance optimization and JVM tuning
- Reactive programming

### 17.3 Practice Resources

- **Coding Practice:** LeetCode, HackerRank, CodeChef
- **Projects:** Build real applications (TODO app, calculator, file manager)
- **Open Source:** Contribute to GitHub projects
- **Documentation:** Read official Java docs regularly
- **Community:** Join Java forums and communities

## 18 Conclusion

This document covered the fundamental and intermediate concepts of Java programming, from basic syntax to object-oriented programming and file handling. Understanding these concepts provides a solid foundation for Java development.

### Key Takeaways:

- Java is object-oriented, platform-independent, and strongly typed
- OOP principles (encapsulation, inheritance, polymorphism, abstraction) are central to Java
- Collections Framework provides powerful data structures
- Exception handling ensures robust error management
- Following best practices and conventions makes code maintainable

**Remember:** Programming is a skill developed through practice. Keep coding, building projects, and learning new concepts. The more you code, the better you become!

*Happy Coding!*