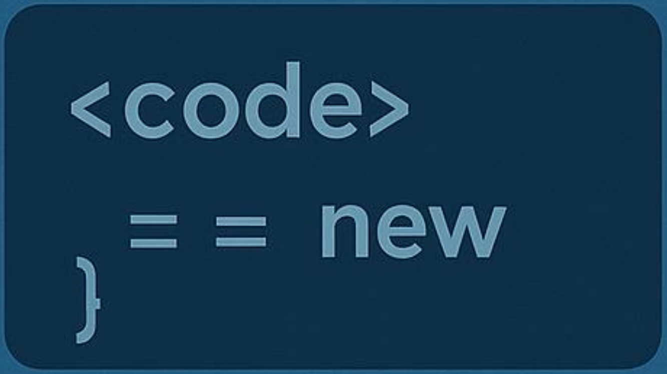
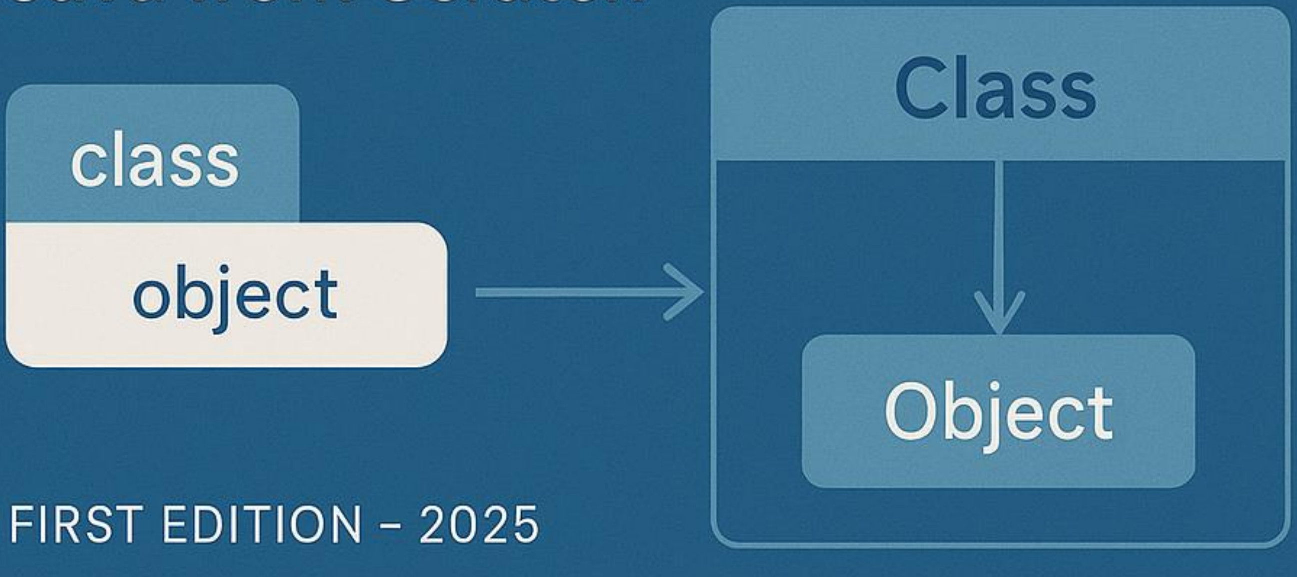


JAVA PROGRAMMING & OBJECT-ORIENTED FUNDAMENTALS

A Beginner's Guide to Learning Java from Scratch



FIRST EDITION – 2025

Java Programming & Object-Oriented Fundamentals

A Beginner's Guide to Learning Java from Scratch

Author:
MD ROBAIDUL ISLAM
B.Sc. (Engineering) in Electronics and Communication Engineering
M.Sc. (Engineering) in Computer Science and Engineering(Ongoing)

Institution:
Hajee Mohammad Danesh Science and Technology University, Dinajpur-5200.

Edition:
First Edition – 2025

Contact:
📧 Email: robaeid14@gmail.com

📚 Table of Contents

Chapter 1: Introduction to Java

What is Java?

Java is a **high-level**, **object-oriented**, and **platform-independent** programming language developed by **Sun Microsystems** in the mid-1990s. It is now owned and maintained by **Oracle Corporation**.

Java was designed to be easy to write, compile, debug, and run. Its "**Write Once, Run Anywhere**" philosophy means that compiled Java code can run on any device that has a Java Virtual Machine (JVM), regardless of computer architecture or operating system.

Key Features of Java

Feature	**Description**
Object-Oriented	Java follows the Object-Oriented Programming paradigm. Everything in Java is associated with classes and objects.
Platform-Independent	Java code is compiled into bytecode, which can be run on any platform with a JVM.
Simple & Easy to Learn	Java syntax is clean, readable, and beginner-friendly.
Robust & Secure	Java includes features like strong memory management and built-in security.
Multithreaded	Java can perform many tasks simultaneously through multithreading.
Portable	Java programs can be moved easily from one computer system to another.
Automatic Garbage Collection	Java automatically manages memory and removes unused objects from memory.

Why Learn Java?

- Java is one of the **most popular and widely used programming languages**.
- It is used to build **web applications, enterprise software, mobile apps (Android)**, games, and more.

- Java provides a **strong foundation** for learning other modern programming languages like Kotlin, Scala, and C#.
- Java is in **high demand** in the job market.
- Learning Java teaches **problem-solving skills** using a structured programming approach.

The Java Program Lifecycle

When you write and run a Java program, the process goes through the following steps:

1. **Write the Code**: You create a `.java` file with your program code.
2. **Compile the Code**: Use the Java Compiler (`javac`) to compile the code into bytecode (`.class` file).
3. **Run the Program**: The Java Virtual Machine (`java` command) executes the compiled bytecode.

Compilation Example:

```
javac HelloWorld.java    // Compiles the program
java HelloWorld          // Runs the compiled bytecode
```

Java is Everywhere!

Java is used in:

- **Android apps**
- **Banking and finance applications**
- **Web-based enterprise applications**
- **Big data technologies**
- **Embedded systems**
- **Scientific computing**
- **Cloud-based systems**

Real-Life Analogy

Imagine Java as a **recipe written in English**.

- The **compiler** translates it into a common language (bytecode).
- The **JVM** acts like a chef who understands this recipe and cooks it in any kitchen (device/OS).

No matter which country (platform) you’re in, the JVM ensures the recipe works the same way!

Mini Quiz

Try answering these simple questions to review what you've learned:

1. Who originally developed Java?
2. What does JVM stand for?
3. What command do we use to compile a Java program?
4. Name two features that make Java platform-independent.
5. Mention one real-life application where Java is used.

Chapter Summary

- Java is a versatile, object-oriented language.
- It is platform-independent due to the JVM.
- It's widely used in real-world applications and offers a great entry point for beginners.
- Understanding Java's basic structure and compilation process is your first step toward becoming a developer.

Chapter 2: Setting Up Java and Your IDE

What You Will Learn in This Chapter

- How to install the Java Development Kit (JDK)
- How to set up an IDE (VS Code, IntelliJ IDEA, or NetBeans)
- How to write, compile, and run a basic Java program

2.1 What is the JDK?

The **Java Development Kit (JDK)** is a software development environment used to develop Java applications. It includes:

- Java Compiler (`javac`)
- Java Runtime Environment (JRE)
- Java Virtual Machine (JVM)
- Standard libraries and tools

You must install the JDK before you can write and run Java programs.

2.2 Installing the JDK (Java 17 or Higher)

🔗 Download Link:

You can download from either of the following:

- Oracle JDK: https://www.oracle.com/java/technologies/javase-downloads.html
- OpenJDK: https://jdk.java.net/

📄 Installation Steps:

1. Go to the official website.
2. Download the installer for your operating system (Windows/Mac/Linux).
3. Run the installer and follow the prompts.
4. After installation, open a terminal (or Command Prompt) and type:

```
java -version
javac -version
```

You should see an output like:

```
java version "17.0.2"
javac 17.0.2
```

2.3 Setting Environment Variables (Windows Only)

1. Go to **System Properties** → **Environment Variables**.
2. Under **System Variables**, find and select `Path`, then click **Edit**.
3. Add the path to your JDK's `bin` folder, for example:
   `C:\Program Files\Java\jdk-17\bin`
4. Click OK and restart your terminal or system.

2.4 Choosing and Setting Up an IDE

To write Java code more efficiently, you should use an IDE (Integrated Development Environment). Here are three popular options:

◆ Option 1: VS Code (Visual Studio Code)

Download: https://code.visualstudio.com/

Steps to Set Up:

1. Install VS Code.
2. Open VS Code and go to the **Extensions** tab.
3. Install the **Java Extension Pack**, which includes:
   - Language Support for Java™ by Red Hat
   - Java Debugger
   - Maven for Java
   - Java Test Runner
   - IntelliCode

Pros: Lightweight, fast, supports many languages
Cons: Requires extra setup for beginners

◆ Option 2: IntelliJ IDEA (Community Edition)

Download: https://www.jetbrains.com/idea/download/

Steps to Set Up:

1. Download and install **IntelliJ IDEA Community Edition**.
2. Launch IntelliJ and create a **new Java project**.
3. IntelliJ will automatically detect your JDK.
4. Create a class with a `main()` method and run the program using the green play button.

Pros: Smart code completion, excellent for long-term use
Cons: Slightly heavier than VS Code

◆ Option 3: NetBeans IDE

Download: https://netbeans.apache.org/download/index.html

Steps to Set Up:

1. Download **Apache NetBeans** (choose the version with Java SE support).
2. Install NetBeans and launch it.
3. Click on **File** → **New Project**.
4. Choose **Java** → **Java Application** and click **Next**.
5. Name your project and select a location.
6. NetBeans will automatically create a file with a `main()` method.

Pros: Beginner-friendly, all-in-one Java support
Cons: Slightly slower startup, interface may feel outdated to some users

2.5 Writing and Running Your First Java Program

HelloWorld.java (in any IDE)

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

To Compile and Run Manually (If Not Using IDE):

```
javac HelloWorld.java      // Compiles the file
java HelloWorld            // Runs the file
```

Output:

```
Hello, world!
```

Note: If you're using an IDE like IntelliJ or NetBeans, just click the **Run** button.

2.6 Tips for Beginners

- Save your file with the `.java` extension.
- The class name should match the file name.
- Use an IDE to avoid common syntax errors.
- Don’t be afraid to experiment — debugging helps you learn.

Chapter Summary

- You installed the JDK and configured environment variables (if needed).
- You explored three IDE options: VS Code, IntelliJ IDEA, and NetBeans.
- You wrote and ran your first Java program successfully!

You're now fully set up to start learning core Java programming.

Chapter 3: Your First Java Program (Explained Line by Line)

What You Will Learn in This Chapter

- How a Java program is structured
- Explanation of each line in a basic Java program
- Key Java concepts: `class`, `main()`, and `System.out.println()`
- How to compile and run your Java code

3.1 Let's Write the Code First

Here is the complete code of a simple Java program:

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

Let’s break this down step by step.

3.2 Line-by-Line Explanation

◆ Line 1: `public class HelloWorld {`

- `public`: This means the class is accessible to all other classes.
- `class`: This is a keyword used to declare a class.
- `HelloWorld`: This is the name of the class.
    - **Important:** The filename must match the class name exactly (`HelloWorld.java`).

☞ A class is the basic building block of any Java program.

◆ Line 2: `public static void main(String[] args) {`

This is the **main method** — the entry point of every Java application.

Let’s break this down:

- `public`: This method is accessible from anywhere.
- `static`: The method can be run without creating an object of the class.
- `void`: The method doesn’t return any value.
- `main`: The name of the method. It is **always** written this way in Java.
- `String[] args`: This allows the program to accept **command-line arguments**.

 Think of the main method as the place where the program starts executing.

◆ Line 3: `System.out.println("Hello, world!");`

This line **prints** a message to the screen.

- `System`: A built-in class that contains useful members.
- `out`: A static member of the `System` class that represents the output stream.
- `println`: A method that prints a line of text and moves to the next line.

🖨 Output:

```
Hello, world!
```

◆ Line 4: `}`

This closes the `main()` method block.

◆ Line 5: `}`

This closes the class block.

3.3 Full Explanation in Plain Language

When you run this program:

1. Java looks for the `main()` method.
2. Inside `main()`, it finds a command to **print** `"Hello, world!"`.
3. It sends that message to the screen.
4. The program ends after that.

It’s that simple!

3.4 Compile and Run (Without IDE)

If you're not using an IDE, do this:

1. Save the file as `HelloWorld.java`
2. Open your terminal or command prompt
3. Navigate to the folder where your file is saved
4. Run:

```
javac HelloWorld.java      // Compiles the program
java HelloWorld            // Runs the compiled code
```

Output:

```
Hello, world!
```

3.5 Common Mistakes to Avoid

Mistake	What Happens
File name and class name don’t match	Compile-time error
Missing `main()` method	Program won't start
Forgetting semicolon `;`	Syntax error
Wrong casing in `System.out.println`	Compile-time error (`Java` is case-sensitive)

Chapter Summary

- Every Java program starts in a class.
- The `main()` method is the entry point.
- `System.out.println()` prints output.
- Java is **case-sensitive** — spelling and capitalization matter!

Chapter 4: Java Syntax and Data Types

What You Will Learn in This Chapter

- Basic Java syntax rules
- Understanding data types in Java
- Declaring and initializing variables
- Working with primitive data types

4.1 Java Syntax: The Basics

Java, like most programming languages, follows a set of rules that define its syntax. Here's a quick look at the key syntax elements:

4.1.1 Statements

A Java statement is a single instruction that tells the program to do something, and it always ends with a **semicolon** (`;`). For example:

```
System.out.println("Hello, world!");  // This is a statement
```

4.1.2 Blocks

A block of code is a group of statements wrapped in curly braces (`{}`), often used inside functions, loops, and classes.

Example:

```
public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello, world!");  // This is inside the main
//method block
    }
}
```

4.1.3 Case Sensitivity

Java is **case-sensitive**, which means that `variable`, `Variable`, and `VARIABLE` are all considered different identifiers.

4.2 Java Data Types: Introduction

Data types in Java specify what kind of data a variable can hold. Java has two categories of data types:

1. **Primitive data types**: Basic data types like integers, decimals, and characters.
2. **Reference data types**: Objects and arrays.

4.3 Primitive Data Types

Java has **8 primitive data types**, each with specific purposes:

Data Type	Description	Example
`byte`	8-bit integer	`byte a = 100;`
`short`	16-bit integer	`short b = 1000;`
`int`	32-bit integer	`int c = 100000;`
`long`	64-bit integer	`long d = 100000000L;`
`float`	32-bit floating-point number	`float e = 10.5f;`
`double`	64-bit floating-point number	`double f = 20.5;`
`char`	16-bit character	`char g = 'A';`
`boolean`	true/false values	`boolean h = true;`

4.3.1 Choosing the Right Data Type

- Use `int` for most integer values.
- Use `double` for decimal numbers.
- Use `boolean` for true/false values.
- Use `char` for single characters.

4.4 Declaring and Initializing Variables

Variables in Java are used to store data. You must **declare** a variable before you use it. You declare a variable by specifying its data type and name.

4.4.1 Declaring Variables

Here's how to declare variables:

```
int x;  // Declaring an integer variable named x
double price;  // Declaring a double variable named price
char grade;  // Declaring a char variable named grade
```

4.4.2 Initializing Variables

After declaring a variable, you can initialize it by assigning a value:

```
x = 5;  // Assigning the value 5 to the variable x
price = 19.99;  // Assigning a value to price
grade = 'A';  // Assigning a character to grade
```

You can also declare and initialize a variable in one step:

```
int x = 5;  // Declaration and initialization
double price = 19.99;  // Same here
```

4.5 Working with Data Types: Examples

4.5.1 Integer Types

```
public class DataTypesExample {
    public static void main(String[] args) {
        int age = 25;  // Integer data type
        long population = 7000000000L;  // Long data type, note the 'L' at 
the end
        System.out.println("Age: " + age);
        System.out.println("World population: " + population);
    }
}
```

4.5.2 Floating Point Types

```
public class DataTypesExample {
    public static void main(String[] args) {
        float price = 10.99f;  // Float data type, note the 'f' at the end
        double temperature = 25.5;  // Double data type
        System.out.println("Price: " + price);
        System.out.println("Temperature: " + temperature);
    }
}
```

4.5.3 Character Type

```
public class DataTypesExample {
    public static void main(String[] args) {
        char grade = 'A';  // Character data type
        System.out.println("Grade: " + grade);
    }
}
```

4.5.4 Boolean Type

```
public class DataTypesExample {
    public static void main(String[] args) {
        boolean isJavaFun = true;  // Boolean data type
        System.out.println("Is Java fun? " + isJavaFun);
    }
}
```

4.6 Type Casting (Converting Data Types)

Sometimes, you need to **convert** one data type to another. This process is known as **type casting**. There are two types of casting:

4.6.1 Implicit Casting (Widening)

This happens automatically when you convert a smaller type to a larger type.

```
int x = 100;
double y = x;  // Implicit casting from int to double
System.out.println(y);  // Output: 100.0
```

4.6.2 Explicit Casting (Narrowing)

This requires manual intervention because data might be lost when converting from a larger type to a smaller one.

```
double x = 9.78;
int y = (int) x;  // Explicit casting from double to int
System.out.println(y);  // Output: 9
```

Chapter Summary

- **Java syntax** rules like using semicolons and curly braces are essential for writing valid programs.
- Java has **8 primitive data types**: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`.
- You must **declare** and **initialize** variables before using them.
- **Type casting** is used to convert one data type to another.

Chapter 5: Variables and Operators

What You Will Learn in This Chapter

- How to declare and use variables in Java
- Different types of operators in Java
- How operators work with variables
- Basic arithmetic, relational, and logical operations

5.1 Understanding Variables

In Java, a **variable** is a container that holds data that can be changed during the execution of the program. Each variable in Java is associated with a **data type**, which determines what kind of data it can store.

5.1.1 Declaring and Initializing Variables

As mentioned in Chapter 4, you can declare and initialize variables in one line:

```
int age = 25;  // Declaring an integer variable and initializing it
```

5.1.2 Variable Naming Rules

Java has specific rules for naming variables:

- Variable names must start with a letter, underscore (_), or a dollar sign (`$`).
- The remaining characters can be letters, numbers, underscores, or dollar signs.
- Variable names are **case-sensitive** (`age` is different from `Age`).
- Keywords (like `int`, `class`, `public`) cannot be used as variable names.

5.2 Types of Operators

Operators are symbols that perform operations on variables or values. In Java, operators can be categorized into the following types:

1. **Arithmetic Operators**
2. **Relational Operators**
3. **Logical Operators**
4. **Assignment Operators**
5. **Unary Operators**

5.2.1 Arithmetic Operators

Arithmetic operators are used to perform basic arithmetic operations.

Operator	Description	Example
`+`	Addition	`int sum = a + b;`
`-`	Subtraction	`int diff = a - b;`
`*`	Multiplication	`int product = a * b;`
`/`	Division	`int quotient = a / b;`
`%`	Modulus (remainder)	`int remainder = a % b;`

Example:

```
public class ArithmeticExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;
        System.out.println("Sum: " + (a + b));  // Output: Sum: 15
        System.out.println("Product: " + (a * b));  // Output: Product: 50
    }
}
```

5.2.2 Relational Operators

Relational operators are used to compare two values.

Operator	Description	Example
`==`	Equal to	`a == b` returns `true` if `a` equals `b`
`!=`	Not equal to	`a != b` returns `true` if `a` is not equal to `b`
`>`	Greater than	`a > b` returns `true` if `a` is greater than `b`
`<`	Less than	`a < b` returns `true` if `a` is less than `b`
`>=`	Greater than or equal to	`a >= b`
`<=`	Less than or equal to	`a <= b`

Example:

```
public class RelationalExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;
      System.out.println("a is equal to b: " + (a == b));  // Output: false
      System.out.println("a is greater than b: " + (a > b));  // Output: true
    }
}
```

5.2.3 Logical Operators

Logical operators are used to perform logical operations, often in conditional statements.

Operator	Description	Example
`&&`	Logical AND	`a > 5 && b < 10` returns `true` if both conditions are true
||	Logical OR	`a > 5 || b < 10` returns `true` if at least one of the conditions is true
`!`	Logical NOT	`!(a > 5)` returns `true` if `a` is not greater than 5

Example:

```
public class LogicalExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;
        System.out.println("Both conditions are true: " + (a > 5 && b < 10));
// Output: true
        System.out.println("At least one condition is true: " + (a > 5 || b >
10));  // Output: true
    }
}
```

5.2.4 Assignment Operators

Assignment operators are used to assign values to variables.

Operator	Description	Example
`=`	Simple assignment	`int a = 10;`
`+=`	Add and assign	`a += 5;` (same as `a = a + 5`)
`-=`	Subtract and assign	`a -= 5;` (same as `a = a - 5`)
`*=`	Multiply and assign	`a *= 5;` (same as `a = a * 5`)
`/=`	Divide and assign	`a /= 5;` (same as `a = a / 5`)

Example:

```
public class AssignmentExample {
    public static void main(String[] args) {
        int a = 10;
        a += 5;  // a = a + 5
        System.out.println("New value of a: " + a);  // Output: 15
    }
 }
```

5.2.5 Unary Operators

Unary operators operate on a single operand.

Operator	Description	Example
`++`	Increment by 1	`a++` (post-increment) or `++a` (pre-increment)
`--`	Decrement by 1	`a--` (post-decrement) or `--a` (pre-decrement)
`+`	Unary plus (indicates positive value)	`+a`
`-`	Unary minus (negates value)	`-a`

Example:

```
public class UnaryExample {
    public static void main(String[] args) {
        int a = 10;
        System.out.println("a++: " + a++);  // Output: 10 (post-increment)
        System.out.println("++a: " + ++a);  // Output: 12 (pre-increment)
    }
}
```

5.3 Operator Precedence

In Java, operators have a **precedence** (priority order) that determines which operation is performed first when multiple operators are used in an expression. For example, multiplication has higher precedence than addition:

```
int result = 10 + 5 * 2;  // Output: 20 (multiplication happens first)
```

Use parentheses `()` to alter the order of operations:

```
int result = (10 + 5) * 2;  // Output: 30 (addition happens first)
```

Chapter Summary

- **Variables** store data that can change during program execution.
- **Arithmetic operators** perform basic math operations.
- **Relational operators** compare two values.
- **Logical operators** help in decision-making.
- **Assignment operators** assign values to variables.
- **Unary operators** work with a single operand.

Chapter 6: Taking Input and Displaying Output

What You Will Learn in This Chapter

- How to display output using `System.out.println()`
- How to take user input using `Scanner` class
- Input methods for different data types
- A basic input-output Java program

6.1 Displaying Output in Java

To show text or values in the console, we use:

```
System.out.println("Your message here");
```

- `System.out` is the output stream
- `println()` prints with a newline
- `print()` prints without newline

Example:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, Java Learner!");
        System.out.print("This is on the same line.");
        System.out.println(" This is next line.");
    }
}
```

6.2 Taking Input in Java

Java provides the **`Scanner`** class in the `java.util` package to read input from the user.

Syntax:

```
import java.util.Scanner;

Scanner input = new Scanner(System.in);
```

Reading Input

Data Type	Scanner Method	Example
int	`nextInt()`	`int num = input.nextInt();`
float	`nextFloat()`	`float f = input.nextFloat();`
double	`nextDouble()`	`double d = input.nextDouble();`
String	`next()`	`String s = input.next();`
Line	`nextLine()`	`String line = input.nextLine();`

6.3 Example Program: Input and Output

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter your name: ");
        String name = scanner.nextLine();

        System.out.print("Enter your age: ");
        int age = scanner.nextInt();

        System.out.println("Hello " + name + ", you are " + age +
        " years old.");
    }
}
```

Sample Output:

```
Enter your name: Neloy
Enter your age: 26
Hello Neloy, you are 26 years old.
```

6.4 Notes on Common Input Mistakes

- Use `nextLine()` after `nextInt()` or `nextDouble()` with caution. It may skip lines due to buffer issues.
- Use `scanner.nextLine()` to consume the leftover newline if needed.

```
scanner.nextLine(); // Clears the newline
```

Chapter Summary

- Use `System.out.print()` and `System.out.println()` for output.
- Use `Scanner` to take different types of input.
- Always import `java.util.Scanner`.
- Practice combining inputs and outputs in small programs.

Chapter 7: Control Flow (If Statements, Loops)

What You Will Learn in This Chapter

- How to use `if`, `else`, and `else if` for decision-making
- How to use `switch` statements
- How loops work in Java (For, While, and Do-While loops)
- When and how to use loops in your programs

7.1 Decision Making with `if` Statements

The `if` statement is used to test a condition, and based on the result, the program will execute a particular block of code.

7.1.1 The Basic `if` Statement

The `if` statement checks if a condition is **true**. If it is, the block of code inside the `if` statement is executed.

```
public class IfExample {
    public static void main(String[] args) {
        int a = 10;
        if (a > 5) {
            System.out.println("a is greater than 5");  // This will be
//printed
        }
    }
}
```

7.1.2 `if-else` Statement

An `if-else` statement is used when you want to execute one block of code if the condition is `true`, and another block of code if the condition is `false`.

```
public class IfElseExample {
    public static void main(String[] args) {
        int a = 3;
        if (a > 5) {
            System.out.println("a is greater than 5");
        } else {
            System.out.println("a is not greater than 5");  // This will be
// printed
        }
    }
}
```

7.1.3 `else if` Statement

When you need to check multiple conditions, you can use the `else if` statement. This allows you to test more than two conditions.

```
public class ElseIfExample {
    public static void main(String[] args) {
        int a = 7;
        if (a > 10) {
            System.out.println("a is greater than 10");
        } else if (a == 7) {
            System.out.println("a is equal to 7");  // This will be printed
        } else {
            System.out.println("a is less than 7");
        }
    }
}
```

7.2 The `switch` Statement

The `switch` statement is an alternative to multiple `if-else` statements. It allows you to test an expression against multiple values.

7.2.1 Basic `switch` Statement

```
public class SwitchExample {
    public static void main(String[] args) {
        int day = 2;
        switch (day) {
            case 1:
                System.out.println("Monday");
                break;
            case 2:
                System.out.println("Tuesday");  // This will be printed
                break;
            case 3:
                System.out.println("Wednesday");
                break;
            default:
                System.out.println("Invalid day");
        }
    }
}
```

7.2.2 The `break` and `default` in `switch`

- The `break` statement exits the `switch` block after executing the matched case.
- The `default` case is optional and is executed when no case matches.

7.3 Loops in Java

Loops are used to repeat a block of code multiple times until a condition is met. Java provides three types of loops: `for`, `while`, and `do-while`.

7.3.1 The `for` Loop

The `for` loop is used when you know how many times you want to repeat a statement.

```
public class ForLoopExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            System.out.println("i = " + i);  // Prints i from 1 to 5
        }
    }
}
```

Explanation:

- The initialization (`int i = 1`) is executed once before the loop starts.
- The condition (`i <= 5`) is evaluated before every iteration. If true, the loop executes.
- The increment (`i++`) is executed after each iteration.

7.3.2 The `while` Loop

The `while` loop is used when the number of iterations is not known in advance, and the loop continues until a condition becomes false.

```
public class WhileLoopExample {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 5) {
            System.out.println("i = " + i);  // Prints i from 1 to 5
            i++;
        }
    }
}
```

Explanation:

- The condition is checked before each iteration. If it's true, the loop executes.
- The loop will terminate when the condition becomes false.

7.3.3 The `do-while` Loop

The `do-while` loop is similar to the `while` loop, but the condition is checked **after** each iteration. This ensures that the loop executes at least once.

```
public class DoWhileLoopExample {
    public static void main(String[] args) {
        int i = 1;
        do {
            System.out.println("i = " + i);  // Prints i from 1 to 5
            i++;
        } while (i <= 5);
    }
}
```

Explanation:

- The loop always runs at least once, even if the condition is false initially.
- The condition is checked after the code inside the loop is executed.

7.4 Nested Loops

You can place one loop inside another. This is called a **nested loop**.

```
public class NestedLoopExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                System.out.println("i = " + i + ", j = " + j);
            }
        }
    }
}
```

Explanation:

- The outer loop runs 3 times (for `i = 1`, `i = 2`, `i = 3`).
- For each iteration of the outer loop, the inner loop runs 3 times.

7.5 Using `break` and `continue` in Loops

- The `break` statement is used to exit a loop immediately.
- The `continue` statement is used to skip the current iteration and proceed to the next one.

7.5.1 Using `break`

```
public class BreakExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                break;  // Exits the loop when i equals 5
            }
            System.out.println("i = " + i);
        }
    }
}
```

7.5.2 Using `continue`

```
public class ContinueExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                continue;  // Skips printing 3
            }
            System.out.println("i = " + i);
        }
    }
}
```

Chapter Summary

- The `if`, `else`, and `else if` statements allow decision-making in your program.
- The `switch` statement is a better alternative for multiple conditions based on a single variable.
- Java provides three types of loops: `for`, `while`, and `do-while`.
- Loops help in executing repetitive tasks until a condition is met.
- `break` and `continue` control the flow of loops by either exiting or skipping iterations.

Chapter 8: Methods and Functions in Java

What You Will Learn in This Chapter

- What are methods/functions?
- Why use methods in Java
- Defining and calling methods
- Method parameters and return values
- Method overloading
- Static vs. Non-static methods
- Best practices for writing methods

8.1 What is a Method?

A **method** in Java is a **block of code** that performs a specific task. You can **call** it whenever you need that task to be executed, helping reduce code duplication.

8.2 Why Use Methods?

- Reuse code
- Improve readability
- Make code modular
- Make debugging easier

8.3 Defining a Method

Syntax:

```
returnType methodName(parameter1, parameter2, ...) {
    // Method body
    return value;
}
```

Example:

```
public static int add(int a, int b) {
    return a + b;
}
```

8.4 Calling a Method

You can call a method by using its name and passing the required arguments.

Example:

```
public class Main {
    public static int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        int result = add(5, 3);
        System.out.println("Sum: " + result);
    }
}
```

Output:

```
Sum: 8
```

8.5 Method with No Return Value

Use `void` when the method does not return anything.

```
public static void greet(String name) {
    System.out.println("Hello, " + name + "!");
}
```

8.6 Method Overloading

Method overloading means having multiple methods with the **same name** but **different parameters**.

Example:

```
public class Main {
    public static int multiply(int a, int b) {
        return a * b;
    }

    public static double multiply(double a, double b) {
        return a * b;
    }

    public static void main(String[] args) {
        System.out.println(multiply(2, 3));      // int version
        System.out.println(multiply(2.5, 3.0));  // double version
    }
}
```

8.7 Static vs Non-Static Methods

- **Static methods** belong to the class and can be called without creating an object.
- **Non-static methods** require you to create an object of the class.

Example (Non-static method):

```
public class Calculator {
    public int square(int x) {
        return x * x;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Square: " + calc.square(5));
    }
}
```

8.8 Best Practices for Methods

- Method name should be a verb (`calculateSum`, `printDetails`)
- Keep method length small and specific
- Prefer meaningful parameter names
- Use comments for complex logic
- Follow naming conventions: `camelCase` for method names

Chapter Summary

- Methods allow code reuse and modular design
- Define a method using `returnType methodName(parameters)`
- Call methods using their name and arguments
- Use method overloading for flexibility
- Choose between static and non-static based on need

Chapter 9: Arrays and Collections

What You Will Learn in This Chapter

- How to store multiple values using arrays
- The difference between one-dimensional and multi-dimensional arrays
- Introduction to `ArrayList` and basic Java Collections
- When to use arrays vs collections

9.1 What is an Array?

An **array** is a container object that holds a fixed number of values of a single type. Each item is stored at a specific index.

9.1.1 Declaring and Initializing Arrays

```
int[] numbers = new int[5];  // Declare an array of size 5
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;
```

Or directly:

```
int[] numbers = {10, 20, 30, 40, 50};  // Shortcut initialization
```

9.1.2 Accessing Array Elements

```
System.out.println(numbers[2]);  // Outputs 30
```

9.1.3 Looping Through Arrays

```
for (int i = 0; i < numbers.length; i++) {
    System.out.println("Element at index " + i + " is: " + numbers[i]);
}
```

9.2 One-Dimensional vs Multi-Dimensional Arrays

9.2.1 One-Dimensional Array

```
String[] fruits = {"Apple", "Banana", "Cherry"};
System.out.println(fruits[1]);  // Banana
```

9.2.2 Two-Dimensional Array

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

You can loop through 2D arrays using nested loops:

```
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}
```

9.3 Introduction to Java Collections

Java **Collections** are dynamic data structures that grow and shrink as needed. They belong to the `java.util` package.

9.4 Using `ArrayList`

An `ArrayList` is like an array, but it can grow and shrink at runtime.

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> colors = new ArrayList<>();

        // Adding elements
        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");

        // Accessing elements
        System.out.println(colors.get(1));  // Green

        // Looping
        for (String color : colors) {
            System.out.println(color);
        }
        // Removing an item
        colors.remove("Green");

        // Size
        System.out.println("Size: " + colors.size());
    }
}
```

Key Methods of `ArrayList`:

Method	Description
`add(value)`	Adds value to the list
`get(index)`	Returns the item at index
`remove(index)`	Removes the item at index
`size()`	Returns current size of list
`clear()`	Removes all elements

9.5 Arrays vs ArrayList

Feature	Array	ArrayList
Size	Fixed	Dynamic
Data Type	Primitive + Objects	Objects only
Performance	Faster	Slower (due to dynamic nature)
Syntax	Simple	More methods and flexibility

Chapter Summary

- Arrays are fixed-size containers for elements of the same type.
- Arrays use indexes starting from 0.
- Multi-dimensional arrays are arrays of arrays.
- `ArrayList` is a dynamic alternative to arrays.
- Use arrays when size is fixed and performance is critical; use `ArrayList` when flexibility is needed.

Chapter 10: Object-Oriented Programming (OOP) Concepts in Java

What You Will Learn in This Chapter

- What Object-Oriented Programming (OOP) is
- Core principles of OOP: Class, Object, Inheritance, Encapsulation, Polymorphism, and Abstraction
- Real-world examples and how to apply them in Java
- Java syntax for implementing OOP

10.1 What is Object-Oriented Programming (OOP)?

Object-Oriented Programming is a programming paradigm that models software around **objects**, which represent real-world entities. Objects have **properties (data)** and **behaviors (methods)**.

Key idea: Organize code into reusable pieces called **classes**, and create **objects** from them.

10.2 Class and Object

10.2.1 Class

A class is a blueprint or template for creating objects.

```
public class Car {
    String color;
    int speed;

    void drive() {
        System.out.println("The car is driving.");
    }
}
```

10.2.2 Object

An object is an instance of a class.

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();  // Creating an object
        myCar.color = "Red";
        myCar.speed = 120;

        System.out.println("Car color: " + myCar.color);
        myCar.drive();  // Calling method
    }
}
```

10.3 Four Pillars of OOP

OOP is based on **four main principles**:

1. Encapsulation

Encapsulation is **hiding internal details** and only exposing necessary parts via methods. We use `private` variables and `public` getter/setter methods.

```
public class Person {
    private String name;

    public void setName(String newName) {
        name = newName;
    }

    public String getName() {
        return name;
    }
}
```

Why Encapsulation?

- Protects data
- Prevents unauthorized access
- Makes code more maintainable

2. Inheritance

Inheritance allows a class to inherit properties and methods from another class.

Keyword: `extends`

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

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

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();  // Inherited from Animal
        d.bark();   // Dog’s own method
    }
}
```

Why Inheritance?

- Promotes code reuse
- Improves organization

3. Polymorphism

Polymorphism means **many forms**. The same method can behave differently depending on context.

Compile-Time Polymorphism (Method Overloading)

Same method name with different parameters.

```
public class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
}
```

Run-Time Polymorphism (Method Overriding)

Subclass changes the method behavior of its superclass.

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

class Cat extends Animal {
    void sound() {
        System.out.println("Cat meows");
    }
}
```

4. Abstraction

Abstraction means hiding **complex details** and showing only **essential features**.

Using Abstract Class

```
abstract class Shape {
    abstract void draw();  // Abstract method
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle");
    }
}
```

Using Interface

```
interface Vehicle {
    void start();
}

class Bike implements Vehicle {
    public void start() {
        System.out.println("Bike started");
    }
}
```

10.4 Access Modifiers in OOP

Modifier	Accessible within class	Accessible within package	Accessible in subclass	Accessible everywhere
`private`	✓	✗	✗	✗
`default`	✓	✓	✗	✗
`protected`	✓	✓	✓	✗
`public`	✓	✓	✓	✓

Chapter Summary

- OOP helps organize code using objects and classes.
- **Encapsulation** hides data and provides access via methods.
- **Inheritance** allows one class to reuse code from another.
- **Polymorphism** enables different behaviors for the same method.
- **Abstraction** hides complexity using abstract classes or interfaces.

Chapter 11: Constructors and the `this` Keyword

What You Will Learn in This Chapter

- What a constructor is and why it's used
- Types of constructors in Java
- Difference between constructor and method
- Use of the `this` keyword in Java
- Practical examples with explanation

11.1 What is a Constructor?

A **constructor** is a special method that is called automatically when an object is created. It has the same name as the class and **does not have a return type**, not even `void`.

Purpose: To initialize objects.

Example:

```
public class Student {
    String name;

    // Constructor
    Student() {
        name = "Unknown";
        System.out.println("Constructor called!");
    }

    void showName() {
        System.out.println("Student name: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student();  // Constructor will be called
        s.showName();
    }
}
```

11.2 Types of Constructors

Java provides two types of constructors:

1. Default Constructor

Constructor with no parameters.

```
public class Book {
    Book() {
        System.out.println("Default constructor called");
    }
}
```

2. Parameterized Constructor

Constructor that takes arguments.

```
public class Book {
    String title;

    Book(String t) {
        title = t;
    }

    void display() {
        System.out.println("Title: " + title);
    }
}
```

11.3 Constructor vs Method

Feature	Constructor	Method
Name	Same as class	Any name
Return Type	No return type	Must have return type (even void)
Called By	Called automatically when object is created	Called explicitly
Purpose	Initializes object	Defines behavior of object

11.4 The `this` Keyword in Java

`this` is a special keyword in Java that refers to the **current object**.

11.4.1 Use Case 1: To resolve naming conflict

```
public class Employee {
    String name;

    Employee(String name) {
        this.name = name;  // Refers to the instance variable
    }

    void display() {
        System.out.println("Name: " + this.name);
    }
}
```

11.4.2 Use Case 2: To call another constructor (Constructor Chaining)

```
public class Box {
    int length, width;

    Box() {
        this(10, 20);  // Calls the parameterized constructor
    }

    Box(int l, int w) {
        length = l;
        width = w;
    }

    void showSize() {
        System.out.println("Size: " + length + "x" + width);
    }
}
```

11.4.3 Use Case 3: To pass current object as a parameter

```
public class Demo {
    void show(Demo obj) {
        System.out.println("Object received");
    }

    void send() {
        show(this);  // Passing current object
    }
}
```

Chapter Summary

- Constructors initialize objects and run when an object is created.
- Default constructors take no arguments, while parameterized constructors do.
- `this` keyword helps differentiate between instance variables and parameters with the same name.
- `this()` can be used to call other constructors within the same class.

Chapter 12: Static Keyword and Static Members

What You Will Learn in This Chapter

- What the `static` keyword means in Java
- Static variables, methods, and blocks
- Difference between static and non-static members
- When and why to use static members
- Practical examples

12.1 What is `static` in Java?

The keyword **`static`** in Java means the member belongs to the **class rather than the instance**. It can be used with:

- Variables
- Methods
- Blocks
- Nested Classes

Static members are shared across all objects of a class.

12.2 Static Variables (Class Variables)

A static variable is shared by **all instances** of the class. It is **allocated memory only once**.

Example:

```
public class Student {
    int id;
    String name;
    static String college = "ABC College";  // Static variable

    Student(int i, String n) {
        id = i;
        name = n;
    }

    void display() {
        System.out.println(id + " " + name + " " + college);
    }
}
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(1, "Alice");
        Student s2 = new Student(2, "Bob");

        s1.display();
        s2.display();
    }
}
```

Output:

```
1 Alice ABC College
2 Bob ABC College
```

12.3 Static Methods

Static methods **belong to the class** and can be called without creating an object.They **can only access static variables** and call other static methods.

Example:

```
public class MathHelper {
    static int square(int x) {
        return x * x;
    }
}
public class Main {
    public static void main(String[] args) {
        int result = MathHelper.square(5);  // No object needed
        System.out.println("Square: " + result);
    }
}
```

12.4 Static Block

A static block runs **once when the class is loaded**. Useful for static initialization.

```
public class Example {
    static int value;

    static {
        value = 100;
        System.out.println("Static block executed");
    }

    public static void main(String[] args) {
        System.out.println("Value: " + value);
    }
}
```

Output:

```
Static block executed
Value: 100
```

12.5 Static vs Non-Static Members

Feature	Static Member	Non-Static Member
Accessed via	Class name	Object reference
Memory	Once per class	New copy per object
Can access	Only static members	Both static and non-static
Called by	No object needed	Requires object

12.6 When to Use Static?

Use `static` when:

- You need a utility method (e.g., `Math.sqrt()`)
- You want to share a variable across all objects
- You need to initialize something once using a static block

Chapter Summary

- `static` means the member belongs to the class, not to objects.
- Static variables are shared by all instances.
- Static methods can be called without creating objects.
- Use static blocks for class-level initialization.

Chapter 13: Packages and Access Modifiers

What You Will Learn in This Chapter

- What are packages in Java
- Types of packages: built-in and user-defined
- How to create and use packages
- Java access modifiers: public, private, protected, and default
- Access levels and their impact

13.1 What is a Package in Java?

A **package** is a namespace that organizes a set of related classes and interfaces. Think of it like a folder in your computer that groups files.

Java packages help:

- Avoid name conflicts
- Control access
- Reuse code

13.2 Types of Packages

Type	Description
Built-in	Provided by Java (e.g., `java.util`, `java.io`)
User-defined	Created by the programmer

13.3 Using Built-in Packages

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);  // Using java.util package
        System.out.print("Enter name: ");
        String name = sc.nextLine();
        System.out.println("Hello, " + name);
    }
}
```

13.4 Creating a User-defined Package

Step 1: Create the package

📄 `mypackage/Message.java`

```
package mypackage;

public class Message {
    public void display() {
        System.out.println("Hello from a custom package!");
    }
}
```

Step 2: Use the package

📄 `Main.java`

```
import mypackage.Message;

public class Main {
    public static void main(String[] args) {
        Message msg = new Message();
        msg.display();
    }
}
```

Note: Folder name must match the package name. Compile using:

```
javac mypackage/Message.java
javac -cp . Main.java
```

13.5 Access Modifiers in Java

Access modifiers define the visibility/scope of variables, methods, and classes.

13.5.1 The Four Access Modifiers

Modifier	Class	Package	Subclass	World
`private`	✓	✗	✗	✗
(default)	✓	✓	✗	✗
`protected`	✓	✓	✓	✗
`public`	✓	✓	✓	✓

13.5.2 Example: Access Modifier Use

```
public class AccessDemo {
    private int a = 10;
    public int b = 20;
    protected int c = 30;
    int d = 40; // default

    public void show() {
        System.out.println("Private: " + a);  // accessible inside class
        System.out.println("Public: " + b);
        System.out.println("Protected: " + c);
        System.out.println("Default: " + d);
    }
}
public class Main {
    public static void main(String[] args) {
        AccessDemo obj = new AccessDemo();
        obj.show();
        System.out.println("Public: " + obj.b);  // Allowed
        // System.out.println("Private: " + obj.a); ✕ Error!
    }
}
```

Chapter Summary

- **Packages** help organize Java classes.
- **Built-in packages** like `java.util` are ready to use.
- **User-defined packages** can be created using the `package` keyword.
- Java offers four **access modifiers** to control visibility:
    - `private`, *(default)*, `protected`, `public`

Chapter 14: Inheritance Deep Dive and `super` Keyword

What You Will Learn in This Chapter

- What is inheritance in Java
- Types of inheritance supported by Java
- The `super` keyword and its uses
- Method overriding with inheritance
- Real-world examples of inheritance

14.1 What is Inheritance?

Inheritance is one of the key features of Object-Oriented Programming. It allows a new class (child/subclass) to acquire the properties and methods of an existing class (parent/superclass).

Benefits:

- Code reuse
- Better organization
- Easier maintenance

Example:

```
class Animal {
    void sound() {
        System.out.println("Animals make sounds");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();  // Inherited method
        d.bark();   // Own method
    }
}
```

14.2 Types of Inheritance in Java

Java supports the following types of inheritance:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**

● Note: Java does **not support multiple inheritance** with classes to avoid ambiguity. However, it supports multiple inheritance using interfaces (covered in Chapter 14).

1. Single Inheritance

```
class Parent {
    void show() {
        System.out.println("Parent class");
    }
}

class Child extends Parent {
    void display() {
        System.out.println("Child class");
    }
}
```

2. Multilevel Inheritance

```
class Grandparent {
    void display1() {
        System.out.println("Grandparent class");
    }
}

class Parent extends Grandparent {
    void display2() {
        System.out.println("Parent class");
    }
}

class Child extends Parent {
    void display3() {
        System.out.println("Child class");
    }
}
```

3. Hierarchical Inheritance

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

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

class Cat extends Animal {
    void meow() {
        System.out.println("Cat meows");
    }
}
```

14.3 The `super` Keyword

`super` is a reference variable used to refer to the **immediate parent class** object.

Uses of `super`:

1. To call the **parent class constructor**
2. To call the **parent class method**
3. To access the **parent class field**

Example 1: `super` to Call Parent Method

```
class Parent {
    void message() {
        System.out.println("Message from Parent");
    }
}

class Child extends Parent {
    void message() {
        super.message();  // Calls Parent's method
        System.out.println("Message from Child");
    }
}
```

Example 2: `super` to Access Parent Field

```
class Parent {
    int x = 10;
}

class Child extends Parent {
    int x = 20;

    void display() {
        System.out.println("Child x = " + x);
        System.out.println("Parent x = " + super.x);
    }
}
```

Example 3: `super()` to Call Parent Constructor

```
class Parent {
    Parent() {
        System.out.println("Parent constructor");
    }
}

class Child extends Parent {
    Child() {
        super();  // Optional, called automatically
        System.out.println("Child constructor");
    }
}
```

14.4 Method Overriding

When a subclass provides its own version of a method that is already defined in its superclass, it is called **method overriding**.

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

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

Note: Use the `@Override` annotation to avoid errors and clarify intent.

Chapter Summary

- Inheritance allows one class to acquire the fields and methods of another.
- Java supports **single**, **multilevel**, and **hierarchical** inheritance.
- The `super` keyword is used to call parent class members.
- Method overriding enables runtime polymorphism and custom behavior in subclasses.

Chapter 15: Polymorphism in Java (Compile-time and Runtime)

What You Will Learn in This Chapter

- What is polymorphism in Java
- Types of polymorphism: Compile-time and Runtime
- Method overloading vs method overriding
- Advantages of polymorphism
- Examples and use cases

15.1 What is Polymorphism?

Polymorphism means "many forms". In Java, polymorphism allows objects to behave in different ways depending on the context.

It enables the **same method name** to perform **different tasks**, depending on the object that invokes it or the number/type of arguments passed.

15.2 Types of Polymorphism

Type	Description	Achieved By
Compile-time	Method is selected at compile time	Method Overloading
Runtime	Method is selected during program execution	Method Overriding

15.3 Compile-time Polymorphism (Method Overloading)

Method Overloading happens when:

- Multiple methods in the same class
- Have the same name
- But different parameters (type, number, or order)

Example:

```
public class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
}
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(2, 3));
        System.out.println(calc.add(4.5, 5.5));
        System.out.println(calc.add(1, 2, 3));
    }
}
```

15.4 Runtime Polymorphism (Method Overriding)

Method Overriding occurs when:

- A subclass provides its own implementation of a method
- That is already defined in its superclass
- The decision is made at **runtime**

Example:

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

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

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

```
public class Main {
    public static void main(String[] args) {
        Animal a;  // Parent reference

        a = new Cat();
        a.sound();  // Cat's version

        a = new Dog();
        a.sound();  // Dog's version
    }
}
```

15.5 Why Use Polymorphism?

- Improves code reusability and scalability
- Allows for flexible and dynamic behavior
- Supports the **Open/Closed Principle**: open for extension, closed for modification
- Makes it easier to manage large programs with multiple object types

15.6 Difference Between Overloading and Overriding

Feature	Overloading	Overriding
Occurs in	Same class	Parent and child classes
Parameters	Must be different	Must be the same
Inheritance required	✗ Not needed	✓ Required
Polymorphism type	Compile-time	Runtime
Method Signature	Must change (number/type of params)	Must be same

Chapter Summary

- **Polymorphism** allows objects to take many forms.
- **Method overloading** provides compile-time polymorphism by allowing multiple methods with the same name but different parameters.
- **Method overriding** provides runtime polymorphism by allowing child classes to change the behavior of inherited methods.
- Polymorphism improves code flexibility, reusability, and maintainability.

Chapter 16: Abstraction and Interfaces in Java

What You Will Learn in This Chapter

- What is abstraction in Java
- Abstract classes and methods
- How to use abstract classes
- What are interfaces in Java
- Differences between abstract classes and interfaces
- Real-life examples of abstraction

16.1 What is Abstraction?

Abstraction is a process of hiding the implementation details and showing only the essential features of an object.

Think of a car: you use the steering wheel to turn, but you don't need to know how the steering mechanism works inside.

In Java, **abstraction** is achieved using:

- **Abstract classes**
- **Interfaces**

16.2 Abstract Classes and Methods

An **abstract class** is a class that cannot be instantiated (you cannot create objects of it). It can have abstract methods (without body) and concrete methods (with body).

Syntax:

```
abstract class Animal {
    abstract void sound();  // Abstract method
    void eat() {
        System.out.println("This animal eats food.");
    }
}
```

Using Abstract Class:

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

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();
        d.eat();  // Inherited concrete method
    }
}
```

Note: If a class has even one abstract method, it must be declared as `abstract`.

16.3 Interfaces in Java

An **interface** is a completely abstract class that contains only abstract methods (until Java 7). From Java 8 onward, interfaces can also have **default** and **static** methods.

Declaring an Interface:

```
interface Shape {
    void draw();  // abstract method by default
}
```

Implementing Interface:

```
class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}
public class Main {
    public static void main(String[] args) {
        Shape s = new Circle();
        s.draw();
    }
}
```

✓ Use the `implements` keyword for interfaces and `extends` for classes.

Interface with Multiple Implementations

```
class Rectangle implements Shape {
    public void draw() {
        System.out.println("Drawing a Rectangle");
    }
}
public class Main {
    public static void main(String[] args) {
        Shape s1 = new Circle();
        Shape s2 = new Rectangle();
        s1.draw();
        s2.draw();
    }
}
```

16.4 Interface vs Abstract Class

Feature	Abstract Class	Interface
Inheritance	`extends`	`implements`
Methods	Can have abstract and concrete	All methods are abstract (Java 7)
Variables	Can have instance variables	Only static final variables
Constructors	Can have constructors	Cannot have constructors
Multiple inheritance	Not supported with classes	Supported using interfaces
Access Modifiers	Can have any modifier	Methods are `public` by default

16.5 Real-Life Analogy

Imagine an **interface** as a **contract**:

If you "sign" (implement) it, you promise to provide definitions for all its methods.

Example:

- `Remote` (interface): defines buttons like `powerOn()`, `volumeUp()`
- `TV` (class): implements `Remote` and defines how each button behaves

Chapter Summary

- **Abstraction** hides unnecessary details and focuses on what an object does.
- **Abstract classes** can have both abstract and concrete methods and are used when classes are closely related.
- **Interfaces** provide full abstraction and are used to achieve multiple inheritance and contracts.
- Java supports **default** and **static** methods in interfaces (from Java 8 onward).

Chapter 17: Encapsulation and the `this` Keyword

What You Will Learn in This Chapter

- What is encapsulation in Java
- How to implement encapsulation using classes and methods
- Benefits of encapsulation
- What is the `this` keyword
- How and when to use `this` keyword

17.1 What is Encapsulation?

Encapsulation is the process of wrapping **data (variables)** and **code (methods)** into a single unit (class). It is also known as **data hiding**.

In Java, we achieve encapsulation by:

1. Declaring variables as **private**
2. Providing **public getter and setter methods** to access and update the values

Example of Encapsulation:

```
public class Student {
    private String name;
    private int age;

    // Setter methods
    public void setName(String n) {
        name = n;
    }

    public void setAge(int a) {
        age = a;
    }

    // Getter methods
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
}
public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("Rafi");
        s.setAge(21);

        System.out.println("Name: " + s.getName());
        System.out.println("Age: " + s.getAge());
    }
}
```

17.2 Advantages of Encapsulation

- **Data hiding**: Only public methods can access private data
- **Increased security**
- **Control**: You can control what data is accessed and how
- **Easier to maintain and modify**
- **Improves flexibility and reusability**

17.3 The `this` Keyword

The **`this`** keyword in Java is a reference variable that refers to the **current object**.

Common Uses of `this`:

1. Distinguish between class variables and parameters
2. Call other constructors in the same class
3. Pass the current object as an argument
4. Return the current object from a method

1. Using `this` to Refer to Instance Variable

```
public class Employee {
    private String name;

    public Employee(String name) {
        this.name = name;  // Refers to instance variable
    }

    public void showName() {
        System.out.println("Name: " + this.name);
    }
}
```

2. Using `this()` to Call Another Constructor

```
public class Demo {
    int a, b;

    Demo() {
        this(10, 20);  // Calls parameterized constructor
        System.out.println("Default constructor");
    }

    Demo(int x, int y) {
        a = x;
        b = y;
        System.out.println("Parameterized constructor");
    }
}
```

3. Using `this` as Method Argument

```
public class Demo {
    void display(Demo obj) {
        System.out.println("Method called using this keyword");
    }

    void call() {
        display(this);  // Passing current object
    }
}
```

17.4 Key Differences: `this` vs Encapsulation

Concept	Purpose	Syntax Example
`this` keyword	Refers to current object	`this.name = name;`
Encapsulation	Data hiding via private fields	`private String name;` + setters/getters

Chapter Summary

- **Encapsulation** bundles data and methods into one class and restricts direct access to data.
- You make variables **private** and access them via **getters and setters**.
- The **`this`** keyword refers to the current object and is commonly used to resolve naming conflicts or refer to constructors/methods within the same class.
- Both encapsulation and `this` improve code readability, safety, and modularity.

Chapter 18: Exception Handling in Java

What You Will Learn in This Chapter

- What are exceptions in Java
- Types of exceptions
- How to handle exceptions using `try`, `catch`, `finally`, and `throw`
- The difference between checked and unchecked exceptions
- Creating your own exceptions
- Best practices in exception handling

18.1 What is an Exception?

An **exception** is an **unexpected event or error** that occurs during program execution and disrupts the normal flow of the program.

Java provides a robust mechanism to **catch and handle exceptions** using specific keywords.

18.2 Types of Exceptions

Type	Description	Examples
Checked	Checked at compile-time	`IOException`, `SQLException`
Unchecked	Occur during runtime	`ArithmeticException`, `NullPointerException`

18.3 Java Exception Hierarchy

```
Object
 └── Throwable
      ├── Error (not meant to be handled)
      └── Exception
           ├── Checked Exceptions
           └── Unchecked Exceptions (RuntimeException)
```

18.4 Basic Exception Handling Syntax

```
try {
    // Code that might throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
} finally {
    // Code that always runs, even if exception occurs
}
```

18.5 Example: Handling ArithmeticException

```
public class Main {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;  // Will throw exception
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        } finally {
            System.out.println("This block always executes.");
        }
    }
}
```

18.6 Multiple Catch Blocks

You can handle different exceptions separately:

```
try {
    String str = null;
    System.out.println(str.length());
} catch (ArithmeticException e) {
    System.out.println("Math error!");
} catch (NullPointerException e) {
    System.out.println("Null value found!");
}
```

18.7 The `throw` Keyword

You can manually throw an exception using `throw`:

```
public class Main {
    static void validateAge(int age) {
        if (age < 18)
            throw new ArithmeticException("Not eligible to vote");
        else
            System.out.println("You are eligible to vote.");
    }

    public static void main(String[] args) {
        validateAge(15);
    }
}
```

18.8 The `throws` Keyword

If a method might throw a **checked exception**, use `throws` in the method signature:

```
import java.io.*;

public class Main {
    static void readFile() throws IOException {
        FileReader fr = new FileReader("file.txt");
        fr.read();
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("File not found.");
        }
    }
}
```

18.9 Creating Your Own (Custom) Exception

```
class MyException extends Exception {
    MyException(String message) {
        super(message);
    }
}

public class Main {
    static void check(int value) throws MyException {
        if (value < 100)
            throw new MyException("Value is less than 100");
        else
            System.out.println("Value is acceptable.");
    }

    public static void main(String[] args) {
        try {
            check(50);
        } catch (MyException e) {
            System.out.println("Caught custom exception: " + e.getMessage());
        }
    }
}
```

18.10 Best Practices for Exception Handling

- Catch **specific exceptions** instead of a generic `Exception`
- Avoid using exceptions for **control flow**
- Always **close resources** (like files or DB connections)
- Use `finally` or **try-with-resources** (Java 7+) to clean up
- Log the exceptions instead of just printing them

Chapter Summary

- **Exceptions** help manage runtime errors gracefully
- Use `try`, `catch`, `finally`, `throw`, and `throws` to handle them
- **Checked exceptions** must be handled at compile-time
- **Unchecked exceptions** occur at runtime and can be avoided by careful coding
- You can define your own **custom exceptions** to handle specific errors

Chapter 19: Java Best Practices

What You Will Learn in This Chapter

- Naming conventions and coding style
- Code structure and readability
- Proper use of classes, methods, and variables
- Exception handling and resource management
- Tips for writing clean, efficient, and maintainable Java code

19.1 Follow Java Naming Conventions

Consistent naming improves readability and teamwork.

Element	Convention	Example
Class name	PascalCase	`StudentInfo`, `BankAccount`
Method name	camelCase	`calculateSalary()`, `printDetails()`
Variable name	camelCase	`userName`, `totalPrice`
Constant (final)	UPPER_CASE_WITH_UNDERSCORES	`MAX_LIMIT`, `PI_VALUE`

19.2 Keep Methods Small and Focused

Each method should do **one thing** and do it well.

✕ Bad:

```
public void processOrder() {
    // handles user input, calculates price, updates DB, sends email - all in
// one!
}
```

✓ Good:

```
public void readInput() {}
public double calculatePrice() {}
public void updateDatabase() {}
public void sendEmail() {}
```

19.3 Comment Thoughtfully

- Avoid obvious comments.
- Use comments to explain **why**, not **what**.

✓ Good:

```
// Retry the operation 3 times to handle temporary network failures
```

✗ Bad:

```
// This is a loop
for (int i = 0; i < 10; i++) { ... }
```

19.4 Use Meaningful Variable Names

Avoid single-letter or vague variable names.

✗ Bad:

```
int a = 10;
String s = "John";
```

✓ Good:

```
int age = 10;
String studentName = "John";
```

19.5 Exception Handling

Handle exceptions properly using `try-catch` blocks. Avoid catching generic `Exception` unless necessary.

✓ Good:

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero!");
}
```

19.6 Avoid Hardcoding

Avoid hardcoding values inside the code. Use constants or configuration files.

✅ **Good:**

```
final double TAX_RATE = 0.15;
```

19.7 Use `final` Where Appropriate

Use `final` for variables you don't want to change.

```
final int MAX_USERS = 100;
```

19.8 DRY Principle (Don't Repeat Yourself)

Avoid duplicating code. Use methods to reuse logic.

19.9 Organize Your Code

- Use proper indentation (4 spaces or tab)
- Group related methods together
- Separate logic into classes and packages

19.10 Use Access Modifiers Properly

Modifier	Access Level
`private`	Within same class only
`default`	Same package
`protected`	Same package + subclasses
`public`	Everywhere

Make fields `private` and use getters/setters.

Chapter Summary

- Follow Java naming conventions
- Keep methods short and meaningful
- Use exceptions and constants properly
- Write clean, readable, and reusable code
- Organize and structure code logically

Chapter 20: Final Project – A Mini Java OOP App

What You Will Build

In this chapter, we will develop a simple **Java OOP-based mini project**: a **Student Management System (Console-Based)**.

You will apply everything you've learned so far:

- Classes and Objects
- Encapsulation (getters/setters)
- Constructors
- Inheritance
- Polymorphism
- Basic Java logic, input/output

🏗 Project Overview

App Name: `Student Management System`

Features:

- Add new students
- Display student information
- Use of classes and OOP concepts

20.1 Step 1 – Create the Base Class `Person`

```
public class Person {
    protected String name;
    protected int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

20.2 Step 2 – Create the Derived Class `Student`

```
public class Student extends Person {
    private String studentId;
    private String course;

    public Student(String name, int age, String studentId, String course) {
        super(name, age);
        this.studentId = studentId;
        this.course = course;
    }

    // Getter and Setter
    public String getStudentId() {
        return studentId;
    }

    public String getCourse() {
        return course;
    }

    @Override
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Student ID: " + studentId);
        System.out.println("Course: " + course);
    }
}
```

20.3 Step 3 – The Main App Logic

```
import java.util.ArrayList;
import java.util.Scanner;

public class StudentManagementApp {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        ArrayList<Student> studentList = new ArrayList<>();

        while (true) {
            System.out.println("\n===== Student Management Menu =====");
            System.out.println("1. Add New Student");
            System.out.println("2. Display All Students");
            System.out.println("3. Exit");
            System.out.print("Enter choice: ");
            int choice = input.nextInt();
            input.nextLine();  // Clear the buffer

            if (choice == 1) {
                System.out.print("Enter Name: ");
                String name = input.nextLine();
                System.out.print("Enter Age: ");
                int age = input.nextInt();
                input.nextLine();  // clear buffer
                System.out.print("Enter Student ID: ");
                String id = input.nextLine();
                System.out.print("Enter Course: ");
                String course = input.nextLine();

                Student student = new Student(name, age, id, course);
                studentList.add(student);
                System.out.println("Student added successfully!");

            } else if (choice == 2) {
                System.out.println("\n=== Student List ===");
                for (Student s : studentList) {
                    s.displayInfo();
                    System.out.println("--------------------");
                }

            } else if (choice == 3) {
                System.out.println("Exiting... Thank you!");
                break;

            } else {
                System.out.println("Invalid choice. Please try again.");
            }
        }

        input.close();
    }
}
```

20.4 Key OOP Concepts Used

Concept	Implementation
Class & Object	`Person`, `Student`, `StudentManagementApp`
Inheritance	`Student extends Person`
Encapsulation	Private fields with getters
Polymorphism	Overridden `displayInfo()` method
Constructor	Used to initialize objects

□ Sample Output

```
===== Student Management Menu =====
1. Add New Student
2. Display All Students
3. Exit
Enter choice: 1
Enter Name: Alice
Enter Age: 20
Enter Student ID: S101
Enter Course: Computer Science
Student added successfully!

===== Student Management Menu =====
1. Add New Student
2. Display All Students
3. Exit
Enter choice: 2

=== Student List ===
Name: Alice
Age: 20
Student ID: S101
Course: Computer Science
--------------------
```

Chapter Summary

- This project ties together all core Java and OOP concepts.
- You practiced creating classes, using inheritance, encapsulation, polymorphism, and user input/output handling.

Chapter 21: What Next? (Career Tips & Advanced Topics)

21.1 Java Career Landscape in Bangladesh (2025)

Bangladesh's tech industry is experiencing significant growth, projected to reach $5 billion by 2025, with over 200,000 new jobs expected by 2024. Java remains one of the most in-demand programming languages, with numerous job opportunities in Dhaka and other tech hubs. (Nucamp)

In-Demand Skills:

- **Java** (Spring Boot, REST APIs, Microservices)
- **Cloud Platforms**: AWS, Azure
- **DevOps Tools**: Docker, Kubernetes
- **AI & Machine Learning**: Integration with Java applications
- **Cybersecurity**: Secure coding practices(Nucamp)

Salary Insights:

- **Junior Java Developer**: Approximately BDT 30,000/month
- **Mid-Level Developer**: Around BDT 91,000/month
- **Senior Developer**: Up to BDT 100,000/month (Glassdoor)

21.2 Opportunities Abroad for Bangladeshi Java Developers

Bangladeshi Java developers are increasingly finding opportunities abroad, especially through remote work platforms. Platforms like Crossover offer remote Java developer positions with competitive salaries, allowing professionals to work for international companies from Bangladesh. (Crossover)

21.3 Advanced Java Topics to Explore

To stay competitive and advance your career, consider delving into:

- **Spring Framework**: Master Spring Boot, Spring Security, and Spring Cloud.
- **Microservices Architecture**: Design and implement scalable microservices.
- **Reactive Programming**: Learn frameworks like Project Reactor.
- **Cloud-Native Development**: Deploy applications on AWS or Azure.

- **Containerization**: Use Docker and Kubernetes for deployment.
- **Big Data Integration**: Work with Hadoop and Apache Spark.(The Knowledge Academy)

21.4 Career Tips for Bangladeshi Students and Programmers

- **Continuous Learning**: Stay updated with the latest Java trends and technologies.
- **Networking**: Join local tech communities and attend workshops.
- **Portfolio Development**: Build and showcase projects on platforms like GitHub.
- **Certifications**: Consider obtaining certifications like Oracle Certified Professional Java Programmer.
- **Soft Skills**: Enhance communication and problem-solving abilities.(The Knowledge Academy, Nucamp)

Chapter Summary

- Bangladesh's tech sector offers abundant opportunities for Java developers.
- Remote work has opened doors to international positions.
- Advanced skills in cloud computing, microservices, and big data are highly valued.
- Continuous learning and networking are key to career advancement.