Java for Beginners

readbytes

Java for Beginners

Learn to Code from Scratch

readbytes.github.io

2025-07-14

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to Java

1 Introduction to Java

1.1 Why Java? History and Use Cases

Java is one of the most influential programming languages in the world. Designed for versatility and cross-platform compatibility, it has powered innovations from web applications to mobile devices and embedded systems. To understand Java's significance, it's essential to explore its origins, evolution, foundational principles, and its wide-ranging applications today.

1.1.1 Origin and Evolution of Java

Java was developed in the early 1990s by James Gosling and his team at Sun Microsystems. Originally called "Oak," it was intended for interactive television, but the project pivoted to the web as the internet surged in popularity. In 1995, the language was officially launched as Java, with the slogan "Write Once, Run Anywhere." This tagline encapsulated Java's core promise: platform independence.

Java quickly rose in popularity due to its robustness, security features, and its virtual machine architecture. Over time, the Java platform evolved through major versions, each adding modern language features. From Java 1.2's introduction of the Swing GUI toolkit to Java 8's lambdas and streams, and Java 17's long-term support status, the language has remained relevant and continuously modernized.

In 2010, Oracle acquired Sun Microsystems, taking over the stewardship of Java. Since then, Java has adopted a more predictable release cycle, providing developers with regular feature updates while maintaining backward compatibility.

1.1.2 Core Principles of Java

Java was built on several foundational principles that continue to define it:

Platform Independence

Java applications are compiled into bytecode that runs on the Java Virtual Machine (JVM), rather than directly on hardware. This means the same code can run unmodified across different operating systems.

Here's a simple Java program that demonstrates portability by printing system information:

Full runnable code:

```java
public class SystemInfo {
    public static void main(String[] args) {
        System.out.println("OS: " + System.getProperty("os.name"));
        System.out.println("Java Version: " +
                           System.getProperty("java.version"));
    }
}
```

This code, once compiled, can be executed on Windows, macOS, or Linux using any JVM—illustrating Java's cross-platform capability.

Security

Java was designed with a strong emphasis on security. The JVM runs code within a controlled environment called the "sandbox," preventing untrusted code from accessing critical system resources. Additionally, features like automatic memory management and strong type checking reduce vulnerabilities like buffer overflows or memory leaks.

In enterprise settings, Java supports secure web application development through technologies like Java EE (now Jakarta EE) and frameworks like Spring, which include built-in protections against threats like SQL injection or cross-site scripting.

Object-Oriented Design

Java is inherently object-oriented. Its design is based on the concept of objects that encapsulate data and behavior. This leads to modular, reusable code that is easier to maintain and scale.

1.1.3 Real-World Applications of Java

Java's versatility has made it a staple in several domains:

Web Development

Java has long been used for server-side development. Frameworks like Spring Boot and Jakarta EE allow developers to build scalable and secure web applications. Large-scale sites like LinkedIn and Netflix use Java for backend services due to its performance and reliability.

Android Development

Android apps are primarily written in Java (and more recently, Kotlin). Java's role in Android development has led to its widespread adoption among mobile developers. From productivity apps to mobile games, Java code powers much of the Android ecosystem.

Enterprise Applications

Java is the backbone of many enterprise systems. Its strong typing, reliability, and vast ecosystem make it ideal for banking software, customer relationship management (CRM) systems, and large-scale inventory management platforms. Tools like Apache Tomcat, JBoss, and Oracle WebLogic Server are central to Java-based enterprise development.

Embedded Systems

Java is also used in embedded systems, such as smart cards, IoT devices, and Blu-ray players. The *Java ME* (Micro Edition) platform is optimized for devices with limited resources, extending Java's reach beyond traditional computing environments.

1.1.4 Javas Relevance Today

Despite being over two decades old, Java remains one of the most used programming languages globally. It consistently ranks high in developer surveys due to its reliability, scalability, and strong community support. The introduction of modern features like pattern matching, records, and improved concurrency in newer versions ensures that Java stays competitive with newer languages like Python or Go.

Java's vast ecosystem—encompassing mature tools, extensive libraries, and robust community support—makes it a safe and powerful choice for beginners and professionals alike. Whether it's building cloud-native applications, enterprise platforms, or mobile apps, Java continues to be a cornerstone of modern software development.

1.2 Installing the JDK and Setting Up Your IDE

Java development begins with installing the **Java Development Kit (JDK)**. Once installed, you can use an IDE (Integrated Development Environment) to write and run your Java code with ease.

1.2.1 Step 1: Download and Install the JDK

Recommended JDK version: Java SE 17 (Long-Term Support)

Download:

Go to the Oracle JDK Downloads page or use Adoptium for a free open-source version.

Install:

- **Windows:**
  - Run the `.exe` installer and follow the setup wizard.
- **macOS:**
  - Download the `.pkg` installer and double-click to install.

- **Linux (Debian/Ubuntu):**
  ```
  sudo apt update
  sudo apt install openjdk-17-jdk
  ```

1.2.2 Step 2: Set Environment Variables (Windows & macOS Only)

Setting the `JAVA_HOME` variable helps tools like IDEs and build systems locate your Java installation.

Windows:

1. Press **Win + S** → Search "Environment Variables".
2. Click **"Edit the system environment variables."**
3. Under **System Properties > Advanced**, click **Environment Variables**.
4. Click **New** under *System Variables*:
   - **Name:** `JAVA_HOME`
   - **Value:** Path to your JDK folder (e.g., `C:\Program Files\Java\jdk-17`)
5. Add `%JAVA_HOME%\bin` to the **Path** variable.

macOS (Terminal):

```
echo 'export JAVA_HOME=$(/usr/libexec/java_home)' >> ~/.zprofile
source ~/.zprofile
```

1.2.3 Step 3: Verify Java Installation

Open your terminal (Command Prompt, PowerShell, or Terminal app) and run:

```
java -version
```

Expected output:

```
java version "17.x.x"
Java(TM) SE Runtime Environment...
```

Also check:

```
javac -version
```

This confirms that the **Java compiler** is installed.

1.2.4 Step 4: Install a Beginner-Friendly IDE

Here are three recommended IDEs. Choose one based on your preference.

1.2.5 Option 1: IntelliJ IDEA Community Edition (Recommended)

Download:

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

Setup a New Project:

1. Open IntelliJ IDEA → Click **New Project**
2. Choose **Java** → Select the JDK you installed (click "Add SDK" if needed).
3. Name your project and click **Create**.

Add a Java class:

1. Right-click `src` → New → Java Class → Name it `Main`
2. Paste this code:

Full runnable code:

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

Run it:

- Click the green **Run** button next to the `main` method or top menu.

1.2.6 Option 2: Eclipse IDE

Download:

https://www.eclipse.org/downloads/

Setup:

1. Launch Eclipse → Select a workspace directory.
2. Go to **File > New > Java Project**
3. Right-click `src` → New → Class → Check "public static void main"
4. Paste your Java code and click **Run**

1.2.7 Option 3: Visual Studio Code (with Java Extension)

Download:

https://code.visualstudio.com

Install Extensions:

1. Open VS Code → Go to Extensions tab (Ctrl+Shift+X)
2. Install: **"Extension Pack for Java"**

Create a Project:

1. Click **Java Projects** (on the side bar) → Create Java Project
2. Choose **No Build Tools**, name it, and select a folder.
3. In `Main.java`, add:

Full runnable code:

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello from VS Code!");
    }
}
```

4. Click the **Run** icon or right-click → Run Java

1.2.8 Final Output

After following the steps above, your terminal or IDE should print:

```
Hello, Java!
```

This confirms your setup is working!

1.2.9 Recap

Step	Task
1	Install JDK
2	Set JAVA_HOME (Windows/macOS)
3	Verify Java in terminal
4	Install IDE (IntelliJ, Eclipse, or VS Code)
5	Create and run a Java program

1.3 Writing Your First Java Program

Once your Java development environment is set up, it's time to write and run your first Java programs. Java programs are made up of classes and methods, and every application must contain a `main` method, which is the entry point when the program runs.

Let's walk through the structure and syntax using two small programs. You'll learn what each part does and be encouraged to make your own changes.

1.3.1 Program 1: Printing a Simple Message

This is the classic first program in many languages—printing "Hello, world!" to the console.

Full runnable code:

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

Line-by-line explanation:

```java
public class HelloWorld {
```

- **`public`**: This keyword means the class is accessible from anywhere in the program.
- **`class`**: Declares a new class named `HelloWorld`.
- **`HelloWorld`**: The name of the class. It must match the filename (`HelloWorld.java`).

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

- **`public`**: This means the method can be called from outside the class—specifically by the Java Virtual Machine (JVM).
- **`static`**: This means the method belongs to the class, not to an instance (object) of the class.
- **`void`**: This means the method doesn't return any value.
- **`main`**: This is the entry point of any Java application.
- **`String[] args`**: A parameter that allows command-line arguments to be passed to the program.

```java
        System.out.println("Hello, world!");
```

- **`System.out.println()`**: A method used to print a line of text to the console.

```java
    }
}
```

- These closing braces end the `main` method and the class.

Try it out:

1. Save the file as `HelloWorld.java`.
2. Compile with: `javac HelloWorld.java`
3. Run with: `java HelloWorld`

Expected output:

```
Hello, world!
```

Experiment: Try changing the message to your name or something else:

```java
System.out.println("Hi, I'm learning Java!");
```

1.3.2 Program 2: A Simple Calculator

Now let's write a Java program that performs a basic arithmetic operation and a string manipulation.

Full runnable code:

```java
public class SimpleCalculator {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;
        int sum = a + b;

        System.out.println("Sum of a and b: " + sum);

        String firstName = "Java";
        String lastName = "Learner";
        String fullName = firstName + " " + lastName;

        System.out.println("Welcome, " + fullName + "!");
    }
}
```

Whats happening here?

```java
        int a = 10;
        int b = 5;
```

- **`int`** declares integer variables `a` and `b` and assigns values to them.

```java
        int sum = a + b;
```

- Calculates the sum of `a` and `b`, and stores it in `sum`.

```java
        System.out.println("Sum of a and b: " + sum);
```

- Prints the result using string concatenation (`+` combines strings and variables).

```
        String firstName = "Java";
        String lastName = "Learner";
```

- Declares two string variables.

```
        String fullName = firstName + " " + lastName;
```

- Combines the two strings with a space in between.

```
        System.out.println("Welcome, " + fullName + "!");
```

- Outputs a personalized welcome message.

Try it out:

1. Save as `SimpleCalculator.java`.
2. Compile: `javac SimpleCalculator.java`
3. Run: `java SimpleCalculator`

Expected output:

```
Sum of a and b: 15
Welcome, Java Learner!
```

Experiment: Try modifying:

- The numbers (`a` and `b`) to perform subtraction or multiplication.
- The names to display your own.

Example:

```
int difference = a - b;
System.out.println("Difference: " + difference);
```

1.4 Compiling and Running Java Code

Java is known for its "write once, run anywhere" philosophy, made possible by the **Java Virtual Machine (JVM)**. Unlike languages that compile directly to machine code (like C++), Java code is first compiled into an intermediate format called **bytecode**, which is then interpreted or compiled just-in-time (JIT) by the JVM at runtime.

Let's explore this process step by step and walk through a complete example.

1.4.1 Java Compilation and Execution Workflow

1. **Write source code** in a `.java` file (human-readable).
2. **Compile** it with `javac`, producing a `.class` file (bytecode).

3. **Run** it using the `java` command, which loads the class into the JVM.

1.4.2 Step 1: Create a Java File

Let's begin with a simple file called `HelloWorld.java`.

Full runnable code:

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello from the JVM!");
    }
}
```

Save this file as `HelloWorld.java` using any plain text editor (Notepad, VS Code, etc.).

1.4.3 Step 2: Compile the Java File

To compile the file, use the `javac` (Java Compiler) command:

```
javac HelloWorld.java
```

What happens:

- The compiler reads `HelloWorld.java` and produces a file called `HelloWorld.class`.
- This `.class` file contains **Java bytecode**, which is platform-independent.

You can verify that the class file was created by listing the directory:

```bash
ls              # macOS/Linux
dir             # Windows
```

You should see:

```
HelloWorld.java
HelloWorld.class
```

1.4.4 What Is Bytecode?

Bytecode is an intermediate, low-level representation of your Java code. It's not readable like source code, but it's also not hardware-specific. The **JVM** can read this format and either interpret it or compile it to machine code just in time (JIT) for efficient execution.

You can inspect bytecode using the `javap` tool:

```
javap HelloWorld
```

This shows a disassembly of the class, revealing its structure without the source code.

1.4.5 Step 3: Run the Program

Now, run the compiled program using the `java` command:

```
java HelloWorld
```

> **Note:** Do **not** include the `.class` extension.

Expected output:

```
Hello from the JVM!
```

1.4.6 Manual Compilation vs IDE Execution

Most IDEs like IntelliJ IDEA, Eclipse, or VS Code perform these steps (compile and run) behind the scenes. However, understanding what's happening under the hood helps you debug issues and build scripts for automation.

Feature	Command Line	IDE
Compilation	`javac HelloWorld.java`	Handled automatically on save/build
Execution	`java HelloWorld`	One-click "Run" button
Error Reporting	Shown in terminal	Displayed in output/error console
Project Structure	Manual setup	Templates and wizards available
Learning Benefit	Greater transparency and control	Faster for larger projects

1.4.7 What Happens in the JVM

When you run `java HelloWorld`, the following steps occur:

1. **Class loading**: The JVM loads `HelloWorld.class` into memory.
2. **Bytecode verification**: The JVM ensures the code is safe and doesn't violate security constraints.
3. **Execution**: The JVM interprets or JIT-compiles the bytecode and runs the program.

This allows your code to run the same way on Windows, macOS, or Linux without recompiling.

1.4.8 Try It Yourself: Practice Exercise

1. Modify your program:

   ```java
   System.out.println("Java is platform-independent!");
   ```

2. Save, compile, and run again:

   ```
   javac HelloWorld.java
   java HelloWorld
   ```

Try creating another file:

Full runnable code:

```java
public class Greet {
    public static void main(String[] args) {
        System.out.println("Greetings from Java!");
    }
}
```

Compile and run:

```
javac Greet.java
java Greet
```

Chapter 2.

Basic Syntax and Structure

1. Variables and Data Types
2. Operators and Expressions
3. Control Flow: if, else, switch
4. Looping Constructs: for, while, do-while

2 Basic Syntax and Structure

2.1 Variables and Data Types

In Java, variables are containers for storing data values. Every variable in Java has a *data type*, which determines the kind of data it can hold. Java is a *statically-typed* language, meaning you must declare the type of a variable before using it.

Java data types are categorized into two main types:

- **Primitive types**
- **Reference types**

2.1.1 Primitive Types

Java has **eight primitive data types**, which are not objects and represent simple values.

Type	Description	Example
`int`	Integer values	`int age = 25;`
`double`	Decimal numbers (floating-point)	`double pi = 3.14;`
`boolean`	True or false values	`boolean isOpen = true;`
`char`	A single Unicode character	`char grade = 'A';`
`byte`	Small integers (-128 to 127)	`byte b = 100;`
`short`	Larger than `byte`	`short s = 32000;`
`long`	Very large integers	`long population = 8000000000L;`
`float`	Less precise decimals than `double`	`float temp = 36.6f;`

Note: `float` and `long` literals require a suffix (`f` or `L`).

2.1.2 Reference Types

Reference types store *references* (addresses) to objects rather than the actual data. Common examples include:

- **Strings**: Sequences of characters.

```java
String name = "Alice";
```

- **Arrays**: Collections of elements of the same type.

```java
int[] scores = {90, 85, 78};
```

Other reference types include objects created from classes.

2.1.3 Declaring and Initializing Variables

You declare a variable by specifying the type followed by the variable name. You can also assign an initial value at the time of declaration.

```
int age = 30;
double price = 9.99;
boolean isAvailable = true;
char initial = 'J';
String greeting = "Hello, world!";
```

You can also declare multiple variables of the same type in one line:

```
int a = 5, b = 10, c = 15;
```

2.1.4 Runnable Example 1: Declaring Multiple Variables

Full runnable code:

```
public class DeclareMultiple {
    public static void main(String[] args) {
        int x = 10, y = 20, sum;
        sum = x + y;
        System.out.println("Sum: " + sum);
    }
}
```

2.1.5 Runnable Example 2: Performing Arithmetic

Full runnable code:

```
public class ArithmeticExample {
    public static void main(String[] args) {
        double price = 19.99;
        int quantity = 3;
        double total = price * quantity;
        System.out.println("Total cost: $" + total);
    }
}
```

2.1.6 Runnable Example 3: Using Booleans in Conditions

Full runnable code:

```java
public class BooleanExample {
    public static void main(String[] args) {
        int age = 18;
        boolean isAdult = age >= 18;

        if (isAdult) {
            System.out.println("You are an adult.");
        } else {
            System.out.println("You are not an adult.");
        }
    }
}
```

2.1.7 Type Safety in Java

Java is **type-safe**, meaning that each variable must be declared with a specific type, and the compiler checks for type mismatches. This helps catch errors early during compilation rather than at runtime.

For example:

```java
int x = 10;
String s = "Hello";
// x = s; // NO Compile-time error: incompatible types
```

2.1.8 Type Conversion

Java supports two types of type conversion:

1. **Implicit (Widening) Conversion** – safe conversion from a smaller type to a larger type:

   ```java
   int a = 10;
   double b = a;  // int to double (safe)
   ```

2. **Explicit (Narrowing) Conversion** – converting from a larger type to a smaller type, which may lead to data loss:

   ```java
   double d = 9.78;
   int i = (int) d;  // Explicit cast: double to int
   System.out.println(i);  // Outputs 9
   ```

2.2 Operators and Expressions

In Java, **operators** are special symbols that perform operations on variables and values. Understanding how to use them is key to writing functional, logical programs.

Java provides several types of operators:

2.2.1 Arithmetic Operators

These are used to perform basic mathematical operations.

Operator	Description	Example (`a = 10, b = 3`)
`+`	Addition	`a + b → 13`
`-`	Subtraction	`a - b → 7`
`*`	Multiplication	`a * b → 30`
`/`	Division	`a / b → 3`
`%`	Modulus (remainder)	`a % b → 1`

Example:

```java
int a = 10, b = 3;
System.out.println("Sum: " + (a + b));
System.out.println("Remainder: " + (a % b));
```

2.2.2 Comparison Operators

These return `true` or `false` depending on the comparison result.

Operator	Description	Example
`==`	Equal to	`a == b → false`
`!=`	Not equal to	`a != b → true`
`>`	Greater than	`a > b → true`
`<`	Less than	`a < b → false`
`>=`	Greater or equal	`a >= b → true`
`<=`	Less or equal	`a <= b → false`

Example:

```java
int age = 18;
System.out.println(age >= 18); // true
```

2.2.3 Logical Operators

Used to combine multiple boolean expressions.

Operator	Description	Example
`&&`	Logical AND	`true && false → false`
||	Logical OR	true || false → true
`!`	Logical NOT	`!true → false`

Example:

```java
int age = 20;
boolean hasID = true;

if (age >= 18 && hasID) {
    System.out.println("Entry allowed.");
} else {
    System.out.println("Entry denied.");
}
```

2.2.4 Assignment Operators

Used to assign values to variables.

Operator	Description	Example
`=`	Assign	`x = 5`
`+=`	Add and assign	`x += 2` → `x = x + 2`
`-=`	Subtract and assign	`x -= 3`
`*=`	Multiply and assign	`x *= 4`
`/=`	Divide and assign	`x /= 2`
`%=`	Modulus and assign	`x %= 3`

Example:

```java
int score = 50;
score += 10;  // score is now 60
System.out.println(score);
```

2.2.5 Operator Precedence

Java evaluates expressions using **precedence rules**, similar to math:

1. **Parentheses** `()` have the highest precedence.

2. Then come **arithmetic operators**: `*`, `/`, `%` before `+`, `-`.
3. **Comparison operators** come after arithmetic.
4. Then **logical operators**: `&&` before `||`.
5. **Assignment** (`=`, `+=`, etc.) has the lowest precedence.

Tip: Use parentheses to clarify complex expressions and ensure the correct order of evaluation.

2.2.6 Example 1: Simple Calculator

Full runnable code:

```java
import java.util.Scanner;

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

        System.out.print("Enter first number: ");
        double num1 = input.nextDouble();

        System.out.print("Enter operator (+, -, *, /): ");
        char op = input.next().charAt(0);

        System.out.print("Enter second number: ");
        double num2 = input.nextDouble();

        double result;
        if (op == '+') result = num1 + num2;
        else if (op == '-') result = num1 - num2;
        else if (op == '*') result = num1 * num2;
        else if (op == '/') result = num1 / num2;
        else {
            System.out.println("Invalid operator!");
            return;
        }

        System.out.println("Result: " + result);
    }
}
```

2.2.7 Example 2: Condition Checker

Full runnable code:

```java
public class ConditionChecker {
    public static void main(String[] args) {
        int score = 85;
        boolean passed = score >= 60;
        boolean hasPerfectAttendance = true;

        if (passed && hasPerfectAttendance) {
            System.out.println("Eligible for reward!");
        } else {
            System.out.println("Not eligible.");
        }
    }
}
```

2.2.8 Example 3: Combining Expressions

Full runnable code:

```
public class ExpressionDemo {
    public static void main(String[] args) {
        int a = 5, b = 10;
        int result = (a + b) * 2; // Parentheses used for precedence
        System.out.println("Result: " + result);
    }
}
```

2.3 Control Flow: if, else, switch

In Java, conditional statements let your program make decisions and execute specific code depending on whether a condition is true or false. The two main conditional control structures are:

- `if/else if/else` (used for flexible, complex conditions)
- `switch` (used for multiple discrete values)

2.3.1 The `if` Statement

The simplest form checks a condition and runs code if it's true.

```
if (condition) {
    // Code runs if condition is true
}
```

2.3.2 Example: Check if a number is positive

Full runnable code:

```java
public class NumberCheck {
    public static void main(String[] args) {
        int num = 7;

        if (num > 0) {
            System.out.println("The number is positive.");
        }
    }
}
```

2.3.3 `if` / `else` Statement

The `else` block runs if the condition is false.

```java
if (condition) {
    // Runs if true
} else {
    // Runs if false
}
```

2.3.4 Example: Positive or negative number

Full runnable code:

```java
public class SignChecker {
    public static void main(String[] args) {
        int num = -3;

        if (num >= 0) {
            System.out.println("The number is non-negative.");
        } else {
            System.out.println("The number is negative.");
        }
    }
}
```

2.3.5 `if`, `else if`, and `else`

You can chain multiple conditions using `else if`.

```java
if (condition1) {
    // Code if condition1 is true
} else if (condition2) {
    // Code if condition2 is true
} else {
    // Code if none are true
}
```

2.3.6 Example: Grade evaluator

Full runnable code:

```java
public class GradeEvaluator {
    public static void main(String[] args) {
        int score = 76;

        if (score >= 90) {
            System.out.println("Grade: A");
        } else if (score >= 80) {
            System.out.println("Grade: B");
        } else if (score >= 70) {
            System.out.println("Grade: C");
        } else if (score >= 60) {
            System.out.println("Grade: D");
        } else {
            System.out.println("Grade: F");
        }
    }
}
```

This is useful when there are **ranges or non-exact values** to compare.

2.3.7 Nested `if` Statements

You can place one `if` inside another for more specific checks.

```java
int age = 20;
boolean hasID = true;

if (age >= 18) {
    if (hasID) {
        System.out.println("Access granted.");
    } else {
        System.out.println("ID required.");
    }
} else {
    System.out.println("Access denied. Must be 18+.");
}
```

Nested `if` is good when **one condition depends on another**.

2.3.8 The `switch` Statement

`switch` is cleaner than multiple `if` statements when you're checking a variable against **specific constant values**.

```
switch (variable) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    default:
        // code
}
```

Each `case` is matched against the value. `break` exits the switch after a match. Without it, execution "falls through" to the next case.

2.3.9 Example: Simple menu system

Full runnable code:

```
import java.util.Scanner;

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

        System.out.println("Menu:");
        System.out.println("1. Say Hello");
        System.out.println("2. Add Numbers");
        System.out.println("3. Exit");
        System.out.print("Choose an option: ");
        int choice = input.nextInt();

        switch (choice) {
            case 1:
                System.out.println("Hello!");
                break;
            case 2:
                System.out.println("5 + 3 = " + (5 + 3));
                break;
            case 3:
                System.out.println("Goodbye!");
                break;
            default:
                System.out.println("Invalid choice.");
            }
        }
    }
```

2.3.10 Choosing Between `if` and `switch`

Use `if` when:	Use `switch` when:
Conditions involve ranges (`score >= 90`)	You're checking one variable for matches
Logic depends on multiple variables	Comparing exact values (like menu options)
You need more complex boolean expressions	Values are constants (`int`, `char`, etc.)

2.3.11 4th Example: Number Sign Checker with `else if`

Full runnable code:

```
public class SignClassifier {
    public static void main(String[] args) {
        int num = 0;

        if (num > 0) {
            System.out.println("Positive");
        } else if (num < 0) {
            System.out.println("Negative");
        } else {
            System.out.println("Zero");
        }
    }
}
```

2.3.12 Logic Flow Summary

- **`if`** evaluates a condition. If it's true, the block runs.
- **`else if`** allows additional checks if the previous ones failed.
- **`else`** catches all remaining cases.
- **`switch`** evaluates a single expression and executes the matching case.

2.4 Looping Constructs: for, while, do-while

In Java, loops allow you to **repeat a block of code** multiple times based on a condition. This is especially useful when processing collections, validating input, or performing repetitive tasks like displaying menus or calculating values.

Java has **three main types of loops**:

- `for` loop
- `while` loop
- `do-while` loop

Let's explore each with practical examples and learn when to use which.

2.4.1 `for` Loop

The `for` loop is used when you know in advance **how many times** you want to repeat a block.

Syntax:

```java
for (initialization; condition; update) {
    // code to repeat
}
```

2.4.2 Example: Counting from 1 to 5

Full runnable code:

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

2.4.3 Example: Print a multiplication table

Full runnable code:

```java
public class MultiplicationTable {
    public static void main(String[] args) {
        int number = 7;
        for (int i = 1; i <= 10; i++) {
            System.out.println(number + " x " + i + " = " + (number * i));
        }
    }
}
```

Use case: Best for **count-controlled loops**, like generating tables, repeating steps a fixed number of times, or iterating through arrays.

2.4.4 `while` Loop

The `while` loop runs as long as a condition is true. It's ideal when you don't know how many times you'll loop, but the condition controls when to stop.

Syntax:

```
while (condition) {
    // code to repeat
}
```

2.4.5 Example: Input validation

Full runnable code:

```
import java.util.Scanner;

public class InputValidator {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int age;

        System.out.print("Enter your age (0-120): ");
        age = input.nextInt();

        while (age < 0 || age > 120) {
            System.out.print("Invalid. Please enter a valid age: ");
            age = input.nextInt();
        }

        System.out.println("Age entered: " + age);
    }
}
```

Use case: Ideal for **condition-controlled loops**, like checking for valid input or reading data until a stop condition is met.

2.4.6 `do-while` Loop

The `do-while` loop is similar to `while`, but the block always runs **at least once**, regardless of the condition.

Syntax:

```java
do {
    // code to repeat
} while (condition);
```

2.4.7 Example: Menu-driven program

Full runnable code:

```java
import java.util.Scanner;

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

        do {
            System.out.println("\nMenu:");
            System.out.println("1. Greet");
            System.out.println("2. Show Time");
            System.out.println("3. Exit");
            System.out.print("Choose an option: ");
            choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    System.out.println("Hello!");
                    break;
                case 2:
                    System.out.println("It's Java time!");
                    break;
                case 3:
                    System.out.println("Goodbye!");
                    break;
                default:
                    System.out.println("Invalid option.");
            }

        } while (choice != 3);
    }
}
```

Use case: Best when you **need the loop to run at least once**, such as showing menus or retry prompts.

2.4.8 Breaking and Continuing Loops

`break` Exits the loop immediately:

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) break;
    System.out.println(i);
}
// Output: 1 2 3 4
```

`continue` Skips the current iteration:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    System.out.println(i);
}
// Output: 1 2 4 5
```

Full runnable code:

```
public class Test {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i == 5) break;
            System.out.println(i);
        }

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

2.4.9 Visual Output Example: Printing a Triangle Pattern

Full runnable code:

```
public class PatternPrinter {
    public static void main(String[] args) {
        int rows = 5;

        for (int i = 1; i <= rows; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.print("* ");
            }
            System.out.println(); // Move to the next line
        }
    }
}
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

2.4.10 Choosing the Right Loop

Loop Type	Best Used When...
`for`	You know how many times to repeat (e.g., count 1–100)
`while`	Repeat until a condition becomes false (e.g., input check)
`do-while`	Run at least once before checking condition (e.g., menu)

Chapter 3.

Working with Methods

3 Working with Methods

3.1 Defining and Calling Methods

In Java, **methods** are blocks of code that perform specific tasks. They help you **organize code**, **avoid repetition**, and **make your programs more readable and reusable**.

This section introduces how to define your own methods, how to give them names and parameters, and how to call them from the `main` method.

3.1.1 What Is a Method?

A **method** in Java is a group of statements that performs a task. It may accept input, do some processing, and optionally return a value.

Here is the general structure of a method:

```
returnType methodName(parameters) {
    // method body
}
```

- `returnType`: The type of value the method returns (e.g., `int`, `String`, `double`, or `void` for no return).
- `methodName`: A name that describes what the method does (use camelCase).
- `parameters`: Inputs the method needs (can be empty).

3.1.2 Example 1: A Method That Adds Two Numbers

Let's define a method that takes two integers as input and returns their sum.

Full runnable code:

```java
public class AddExample {

    // Method that returns the sum of two integers
    public static int add(int a, int b) {
        int result = a + b;
        return result;
    }

    public static void main(String[] args) {
        int x = 5;
        int y = 7;
        int sum = add(x, y); // Method call
        System.out.println("The sum is: " + sum);
    }
}
```

3.1.3 Explanation:

- `public static int add(int a, int b)`: A method named `add` that returns an `int`.
- `int sum = add(x, y)`: Calls the method and stores the result in `sum`.
- The method is **called from `main`**, which is where program execution begins.

3.1.4 Example 2: A Method That Returns a Greeting

Now let's define a method that returns a personalized greeting using a `String` parameter.

Full runnable code:

```
public class GreetingExample {

    // Method that returns a greeting message
    public static String greet(String name) {
        return "Hello, " + name + "!";
    }

    public static void main(String[] args) {
        String message = greet("Alice");
        System.out.println(message);
    }
}
```

3.1.5 Explanation:

- **Return type**: `String`, because the method returns text.
- **Parameter**: `name` of type `String`.
- When the method is called, it returns a custom message.

This method improves **reusability** — you can greet anyone by just changing the name.

3.1.6 Example 3: A Method That Calculates Area

Let's write a method to calculate the area of a rectangle.

Full runnable code:

```java
public class AreaCalculator {

    // Method to calculate the area of a rectangle
    public static double calculateArea(double width, double height) {
        return width * height;
    }

    public static void main(String[] args) {
        double w = 4.5;
        double h = 3.2;
        double area = calculateArea(w, h);
        System.out.println("Area: " + area);
    }
}
```

3.1.7 Key Concepts:

- **Return type**: `double`, because area may have decimal values.
- **Parameters**: Two `double` values for width and height.
- **Return value**: The product of `width` and `height`.

3.1.8 Notes on Method Structure and Readability

When defining methods:

- **Use descriptive names**: `calculateArea` is more meaningful than `doStuff`.
- **Keep methods focused**: A method should do *one specific thing*.
- **Indent code clearly**: Improves readability and reduces errors.
- **Use consistent formatting**: Helps others (and you!) read your code later.

3.1.9 Calling Methods from `main`

All Java programs start from the `main` method:

```java
public static void main(String[] args) {
    // This is where your program starts
}
```

You call methods from `main` by using their name and passing any required arguments. The method's return value (if any) can be stored in a variable or printed directly.

3.1.10 Why Use Methods?

- **Reusability**: Write once, use many times.
- **Clarity**: Divide large programs into manageable pieces.
- **Testing**: Test individual parts of your code easily.
- **Avoid duplication**: Replace repeated code with method calls.

3.2 Parameters and Return Values

In Java, **parameters** and **return values** are essential parts of how methods communicate with the rest of your program. Parameters let you pass input into a method, and return values allow the method to pass results back.

This section builds on what you learned in Section 1 by expanding how methods handle input and output. You'll learn how to define methods with multiple parameters, use `void` when no return is needed, and return various types like `int`, `String`, and `boolean`.

3.2.1 What Are Parameters and Return Values?

- **Parameters** are variables listed inside the parentheses of a method declaration. They receive the values (called *arguments*) you pass when calling the method.
- A **return value** is the result a method gives back using the `return` keyword. If a method doesn't return anything, its return type is `void`.

3.2.2 Method with Multiple Parameters

You can define a method that accepts more than one parameter by separating them with commas.

3.2.3 Example: Sum of Three Numbers

Full runnable code:

```java
public class MultiParameterExample {

    // Method with three parameters
    public static int addThreeNumbers(int a, int b, int c) {
        return a + b + c;
    }

    public static void main(String[] args) {
        int result = addThreeNumbers(5, 10, 15); // Passing three arguments
        System.out.println("Sum: " + result);    // Output: Sum: 30
    }
}
```

- The method `addThreeNumbers` accepts three `int` parameters.
- The result is returned using `return a + b + c;`.
- From `main`, the method is called with `5`, `10`, and `15` as arguments.

3.2.4 Method with No Return (`void`)

Sometimes, you don't need a method to return a value — you just want it to perform an action, like printing something.

3.2.5 Example: Displaying a Welcome Message

Full runnable code:

```
public class VoidExample {

    // Method with no return value
    public static void showWelcomeMessage(String name) {
        System.out.println("Welcome, " + name + "!");
    }

    public static void main(String[] args) {
        showWelcomeMessage("Alice"); // Output: Welcome, Alice!
    }
}
```

- The method uses `void` because it doesn't return anything.
- It takes a single `String` parameter and prints a message.

3.2.6 Methods Returning Different Types

Java methods can return any data type. Let's look at a few examples:

3.2.7 a. Method Returning `int`

```java
public static int square(int number) {
    return number * number;
}
```

Usage:

```java
int result = square(6);  // result = 36
```

3.2.8 b. Method Returning `String`

```java
public static String getFullName(String firstName, String lastName) {
    return firstName + " " + lastName;
}
```

Usage:

```java
String name = getFullName("John", "Doe");  // name = "John Doe"
```

3.2.9 c. Method Returning `boolean`

```java
public static boolean isEven(int number) {
    return number % 2 == 0;
}
```

Usage:

```java
if (isEven(4)) {
    System.out.println("The number is even.");
}
```

These examples show how return types allow your methods to send different kinds of data back to the caller.

3.2.10 Interactive Example: Age Checker

Let's create a small program that checks if someone is an adult based on their age.

Full runnable code:

```java
import java.util.Scanner;

public class AgeChecker {
    // Method that returns a boolean
    public static boolean isAdult(int age) {
        return age >= 18;
    }

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

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

        if (isAdult(userAge)) {
            System.out.println("You are an adult.");
        } else {
            System.out.println("You are not an adult.");
        }
    }
}
```

3.2.11 Breakdown:

- **Parameter:** `int age`
- **Return type:** `boolean`
- The `main` method uses a `Scanner` to take input and passes it to `isAdult()`.
- The method returns `true` or `false`, which is used in an `if` condition.

This shows how methods with parameters and return values can interact with user input to make decisions.

3.2.12 Passing Arguments and Capturing Returns

When you call a method, the **arguments** must match the **parameters** in both **type** and **order**.

```
public static double average(int a, int b) {
    return (a + b) / 2.0;
}

// Calling the method
double result = average(8, 10);  // result = 9.0
```

- You pass `8` and `10` as arguments.
- The result is captured in the variable `result`.

Always ensure:

- The number of arguments matches the number of parameters.

- The types are compatible.

3.3 Method Overloading

In Java, you can define **multiple methods with the same name** as long as they have **different parameter lists**. This is called **method overloading**. It allows a class to perform similar tasks in slightly different ways without creating completely different method names.

3.3.1 What Is Method Overloading?

Method overloading means defining more than one method in the same class with the **same name**, but with different:

- Number of parameters
- Types of parameters
- Order of parameters

Important: The method **name stays the same**, but the **parameter list must differ** in some way.

3.3.2 Why overload methods?

- To make code **simpler and more intuitive**
- To let a method handle **different input formats or values**
- To avoid unnecessary and confusing method names like `calculateAreaSquare`, `calculateAreaRectangle`, etc.

3.3.3 Overloading in Action: Calculating Area

Let's look at a practical example: calculating the area of different shapes using **overloaded methods**.

3.3.4 Method 1: Area of a Square

```java
public static double calculateArea(double side) {
    return side * side;
}
```

- Takes **1 parameter** (side length)
- Returns area of a square

3.3.5 Method 2: Area of a Rectangle

```java
public static double calculateArea(double length, double width) {
    return length * width;
}
```

- Takes **2 parameters**
- Returns area of a rectangle

3.3.6 Method 3: Area of a Circle

```java
public static double calculateArea(double radius, boolean isCircle) {
    if (isCircle) {
        return Math.PI * radius * radius;
    } else {
        return -1; // Invalid input
    }
}
```

- Takes a `double` and a `boolean`
- Only calculates area if `isCircle` is `true`

3.3.7 Putting It Together: Full Example

Full runnable code:

```java
public class AreaOverload {

    // Square
    public static double calculateArea(double side) {
        return side * side;
    }

    // Rectangle
    public static double calculateArea(double length, double width) {
        return length * width;
    }

    // Circle
    public static double calculateArea(double radius, boolean isCircle) {
        if (isCircle) {
            return Math.PI * radius * radius;
        } else {
            return -1;
        }
    }

    public static void main(String[] args) {
        System.out.println("Area of square (5): " + calculateArea(5));
        System.out.println("Area of rectangle (4, 6): " + calculateArea(4, 6));
        System.out.println("Area of circle (radius 3.0): " + calculateArea(3.0, true));
    }
}
```

3.3.8 Output:

```
Area of square (5): 25.0
Area of rectangle (4, 6): 24.0
Area of circle (radius 3.0): 28.274333882308138
```

3.3.9 How Does Java Know Which Method to Call?

Java determines which overloaded method to execute using **method signature**, which includes:

- The method name
- The number and type of parameters

At compile time, Java checks the **arguments you pass** and chooses the method whose **parameters match best**.

3.3.10 Examples:

- `calculateArea(5)` → Matches method with **1 `double`** → square
- `calculateArea(4, 6)` → Matches method with **2 `double`** → rectangle
- `calculateArea(3.0, true)` → Matches method with **`double` and `boolean`** → circle

3.3.11 Rules of Overloading

- You **can't overload** a method by just changing the **return type**.
- Parameter **types, order, or count** must be different.
- Methods must be in the **same class** (or inherited into it) to be considered overloaded.

Invalid Example (doesn't compile):

```
public static int example() { return 1; }
public static double example() { return 2.0; } // NO Error: duplicate method
```

3.3.12 Benefits of Method Overloading

Improves readability

Using the same method name for similar operations makes code easier to understand.

Enhances reusability

You can handle multiple input variations without rewriting code.

Simplifies interfaces

Users of your method don't have to remember different method names for similar tasks.

3.4 The `main` Method Explained

Every Java program begins execution from a special method called the **`main` method**. This method acts as the **entry point** — the place where the program starts running.

If your class doesn't have a correctly defined `main` method, the Java Virtual Machine (JVM) won't know where to begin, and your program won't run.

3.4.1 The Full Syntax

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

Let's break down each part of this line:

3.4.2 `public`

This is an **access modifier**. It means the `main` method is **accessible from anywhere**, including by the JVM when the program starts.

3.4.3 `static`

This means the method **belongs to the class** rather than to any specific object. Since the JVM needs to call this method before any objects exist, `main` must be `static`.

3.4.4 `void`

This indicates that the method **does not return a value**. The `main` method simply starts the program — it doesn't give back a result.

3.4.5 `main`

This is the **name of the method**. It must be spelled exactly like this — lowercase `main` — because the JVM looks for it when launching a program.

3.4.6 `String[] args`

This is an **array of Strings**. It holds any **command-line arguments** passed when the program is run. If you don't provide any, the array is simply empty.

3.4.7 Example: Printing Command-Line Arguments

Here's a simple program that prints whatever arguments you pass to it:

Full runnable code:

```
public class CommandLineDemo {

    public static void main(String[] args) {
        System.out.println("Arguments received:");
        for (int i = 0; i < args.length; i++) {
            System.out.println("Arg " + i + ": " + args[i]);
        }
    }
}
```

3.4.8 How to Run:

If you compile and run the program from a command line:

```
java CommandLineDemo Hello 123 Java
```

3.4.9 Output:

```
Arguments received:
Arg 0: Hello
Arg 1: 123
Arg 2: Java
```

3.4.10 Whats Happening?

- You passed three arguments: `Hello`, `123`, and `Java`.
- The `main` method stores them in the `args` array.
- The program loops through the array and prints each one.

3.4.11 Why Is the `main` Method Crucial?

- It's the **starting point** for every standalone Java application.
- Without it, your class cannot be run by itself — the JVM won't know where to begin.
- Even in more complex programs with many classes and methods, execution always begins with the `main` method.

3.4.12 Can You Have Other Methods Too?

Yes! In addition to the main method, you can define and use other methods to organize your code and avoid repetition.

Full runnable code:

```java
public class Welcome {

    public static void printGreeting() {
        System.out.println("Welcome to Java!");
    }

    public static void main(String[] args) {
        printGreeting(); // Call to another method
    }
}
```

Chapter 4.

Understanding Objects and Classes

1. Defining Classes and Creating Objects
2. Fields and Methods in Classes
3. Constructors and `this` Keyword
4. Introduction to Encapsulation

4 Understanding Objects and Classes

4.1 Defining Classes and Creating Objects

In Java, **everything revolves around objects**. Objects represent real-world things or concepts by combining **data** and **behavior**. To create objects, you first need a **class**, which acts like a blueprint describing what an object should look like and what it can do.

In this section, we'll learn how to define a class and create objects from it, covering key concepts like class structure, naming conventions, and the `new` keyword. We'll also peek under the hood to see what happens during object creation.

4.1.1 What Is a Class?

A **class** is a template or blueprint that defines:

- **Fields** (also called variables or attributes): To hold data/state
- **Methods**: To define behavior/actions

The class itself is not an object but a design for objects.

4.1.2 Basic Structure of a Java Class

Here's a simple example of a class named `Person`:

```java
public class Person {
    // Fields (attributes)
    String name;
    int age;

    // Method (behavior)
    void sayHello() {
        System.out.println("Hello! My name is " + name + ".");
    }
}
```

4.1.3 Explanation:

- The keyword `public` means this class is accessible from other parts of the program.
- The class name `Person` starts with an uppercase letter by Java naming conventions.
- Fields `name` and `age` hold data about a person.
- The method `sayHello()` prints a greeting using the person's name.

4.1.4 Naming Conventions

- Class names should be **nouns** and start with an uppercase letter (e.g., `Car`, `Book`, `Employee`).
- Use **camel case** for multiple words: `BankAccount`, `StudentRecord`.
- Fields and methods usually start with a lowercase letter: `firstName`, `calculateTotal()`.

4.1.5 Creating Objects (Instantiating a Class)

To create an actual object from the class blueprint, you use the **`new` keyword** followed by the class constructor (which we'll cover later but can be called simply by the class name and parentheses for now).

4.1.6 Example: Creating and Using a `Person` Object

```java
public class PersonExample {
    public static void main(String[] args) {
        Person person1 = new Person(); // Create a new Person object

        // Assign values to fields
        person1.name = "Alice";
        person1.age = 30;

        // Call method on the object
        person1.sayHello();
    }
}
```

Output:

```
Hello! My name is Alice.
```

Full runnable code:

```java
class Person {
    // Fields (attributes)
    String name;
    int age;

    // Method (behavior)
    void sayHello() {
        System.out.println("Hello! My name is " + name + ".");
    }
}
public class Test {
    public static void main(String[] args) {
        Person person1 = new Person(); // Create a new Person object
        // Assign values to fields
        person1.name = "Alice";
        person1.age = 30;

        // Call method on the object
        person1.sayHello();
    }
}
```

4.1.7 Another Example: Defining and Creating a Car Object

Full runnable code:

```
class Car {
    String make;
    String model;
    int year;

    void displayInfo() {
        System.out.println(year + " " + make + " " + model);
    }
}

public class CarDemo {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.make = "Toyota";
        myCar.model = "Camry";
        myCar.year = 2020;

        myCar.displayInfo();
    }
}
```

Output:

```
2020 Toyota Camry
```

4.1.8 What Happens When You Create an Object?

When you write:

```
Person person1 = new Person();
```

- Java **allocates memory on the heap** to store the new `Person` object.
- The variable `person1` holds a **reference** (think of it as an address or pointer) to this object in memory.
- If you create another object with:

```java
Person person2 = new Person();
```

`person2` refers to a **different object** in memory.

4.1.9 Reference vs. Value

- **Primitive types** (like `int`, `double`, `boolean`) store **values directly**.
- **Objects** are accessed via **references**, meaning variables hold addresses pointing to where the actual data lives in memory.

This is why:

```java
Person person1 = new Person();
Person person2 = person1; // Both point to the same object!
```

Changing a field through `person2` will affect the same object as `person1`.

4.1.10 Write Your Own Class and Create Objects

Try this:

- Define a class `Book` with fields like `title`, `author`, and `pages`.
- Add a method `displayDetails()` to print the book info.
- In `main`, create an object of `Book`, assign values, and call the method.

Example skeleton:

```java
class Book {
    // fields here

    // method here
}

public class BookDemo {
    public static void main(String[] args) {
        // create Book object and use it
    }
}
```

4.2 Fields and Methods in Classes

In Java, **classes** combine both **data** and **behavior** to model real-world objects. The data is stored in **fields** (also called variables), and the behavior is defined by **methods**. This section explains how to declare fields and methods inside a class, and the difference between

instance variables and **class (static) variables**.

4.2.1 Fields: Storing Data in a Class

Fields represent the properties or attributes of an object. When you create an object (an instance of a class), it can hold its own unique data in its instance variables.

4.2.2 Instance Variables

- Belong to **each individual object**.
- Each object has its own copy.
- Example: Each `Student` object has its own `name` and `grade`.

```java
public class Student {
    String name;   // instance variable
    int grade;     // instance variable
}
```

4.2.3 Class (Static) Variables

- Belong to the **class itself**, not individual objects.
- Shared across all instances.
- Useful for constants or counters.

```java
public class Student {
    String name;
    int grade;
    static int studentCount; // static variable shared by all Students
}
```

Every time a new `Student` is created, you could increase `studentCount` to track the total number of students.

4.2.4 Methods: Defining Behavior

Methods are blocks of code that describe what objects can do or what operations can be performed on their data.

4.2.5 Method Structure

```
returnType methodName(parameterList) {
    // method body
}
```

- **returnType**: The type of value the method returns (`void` if nothing is returned).
- **methodName**: A descriptive name following camelCase.
- **parameters**: Input variables (optional).

4.2.6 Example: `Book` Class with Fields and Methods

Full runnable code:

```
public class Book {
    // Fields
    String title;
    String author;
    int pages;
    static int bookCount = 0; // counts all Book objects created

    // Constructor (to be covered later), for now assume default constructor

    // Method to display book info
    void displayInfo() {
        System.out.println(title + " by " + author + ", " + pages + " pages");
    }

    // Method to check if the book is a long read
    boolean isLongBook() {
        return pages > 300;
    }
}

public class BookDemo {
    public static void main(String[] args) {
        Book myBook = new Book(); // Create a new Book object

        // Set fields
        myBook.title = "Java Basics";
        myBook.author = "Jane Doe";
        myBook.pages = 350;

        // Call methods
        myBook.displayInfo();

        if (myBook.isLongBook()) {
            System.out.println("This is a long book.");
        } else {
            System.out.println("This book is short.");
        }
    }
}
```

4.2.7 Output:

```
Java Basics by Jane Doe, 350 pages
This is a long book.
```

4.2.8 Key Takeaways

- **Instance variables** hold data unique to each object.
- **Static variables** belong to the class and are shared by all objects.
- Methods use fields to perform tasks and return results.
- Fields and methods together **encapsulate** an object's data and behavior.

4.3 Constructors and `this` Keyword

When you create an object in Java, it's important to **initialize** its fields — that is, set its initial state. This is where **constructors** come into play. Constructors are special methods designed to set up new objects right when they are created.

In this section, we will explore what constructors are, how they differ from regular methods, how to use the `this` keyword to distinguish between fields and parameters, and how constructors can be overloaded and chained.

4.3.1 What Is a Constructor?

A **constructor** is a special method that is automatically called when a new object is created using the `new` keyword. Its main purpose is to **initialize the object's fields**.

4.3.2 How Constructors Differ from Regular Methods

- **No return type**, not even `void`.
- Name **must exactly match the class name**.
- Called implicitly during object creation.

- Used only to initialize objects.

4.3.3 Default Constructor

If you don't provide any constructors in your class, Java automatically creates a **default constructor** with no parameters that does nothing but create the object.

```java
public class Rectangle {
    int width;
    int height;
}
```

In this case, you can create objects like:

```java
Rectangle rect = new Rectangle();
```

But the fields `width` and `height` remain uninitialized (default to 0).

4.3.4 Parameterized Constructor

To set fields with meaningful values at creation, you define a constructor that takes parameters:

```java
public class Rectangle {
    int width;
    int height;

    // Parameterized constructor
    public Rectangle(int width, int height) {
        this.width = width;   // 'this' distinguishes field from parameter
        this.height = height;
    }
}
```

4.3.5 The this Keyword

In the constructor above, the keyword `this` refers to the **current object**. It helps differentiate between:

- The **field** `width` and `height`
- The **constructor parameters** `width` and `height`

Without `this`, writing `width = width;` would just assign the parameter to itself and leave the field unchanged.

4.3.6 Constructor Overloading and Chaining

You can define multiple constructors with different parameter lists. This is **constructor overloading**.

4.3.7 Example: Overloaded Constructors with Chaining

Full runnable code:

```java
public class Rectangle {
    int width;
    int height;

    // Default constructor
    public Rectangle() {
        this(10, 10);  // Calls the parameterized constructor with default values
    }

    // Parameterized constructor
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
}

public class RectangleDemo {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();        // Uses default constructor
        Rectangle r2 = new Rectangle(5, 8);    // Uses parameterized constructor

        System.out.println("Rectangle 1: " + r1.width + "x" + r1.height);
        System.out.println("Rectangle 2: " + r2.width + "x" + r2.height);
    }
}
```

Here:

- The **default constructor** calls the **parameterized constructor** using `this(...)`.
- This technique is called **constructor chaining** and avoids code duplication.

Output:

```
Rectangle 1: 10x10
Rectangle 2: 5x8
```

4.3.8 Practical Example 2: Constructor Overloading in a `Person` Class

Full runnable code:

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

    // Constructor with name only
    public Person(String name) {
        this(name, 0);  // Calls the other constructor, sets age to 0
    }

    // Constructor with name and age
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void display() {
        System.out.println(name + ", Age: " + age);
    }
}

public class PersonDemo {
    public static void main(String[] args) {
        Person p1 = new Person("Alice");
        Person p2 = new Person("Bob", 25);

        p1.display();  // Output: Alice, Age: 0
        p2.display();  // Output: Bob, Age: 25
    }
}
```

4.3.9 Why Use Constructors?

- **Automatic initialization:** Objects come ready to use with all important fields set.
- **Cleaner code:** Avoids repetitive assignment outside the class.
- **Multiple ways to create objects:** Different constructors offer flexibility.
- **Constructor chaining:** Promotes reuse and simplifies complex initialization.

4.3.10 Summary

- Constructors are special methods called when an object is created.
- They have no return type and share the class's name.
- The `this` keyword differentiates fields from parameters.
- Constructors can be overloaded to accept different inputs.
- Constructor chaining uses `this(...)` to call one constructor from another.
- Constructors make object creation clean, flexible, and error-free.

4.4 Introduction to Encapsulation

In object-oriented programming, **encapsulation** is one of the core principles that helps keep your code organized, secure, and easy to maintain. But what exactly is encapsulation, and why is it so important?

4.4.1 What Is Encapsulation?

Encapsulation means **wrapping** the data (fields) and the code that manipulates the data (methods) into a single unit — a class — and **restricting direct access to some of the object's components**.

In simpler terms:

- You **hide the internal details** of an object.
- You expose only what is necessary through **public methods**.
- You protect the data from **accidental or unauthorized changes**.

This is often described as **data hiding** and is essential for building reliable software.

4.4.2 Why Is Encapsulation Important?

- **Data Protection:** By making fields `private`, you prevent external code from directly modifying important data.
- **Control:** You control how data is accessed or updated via methods (called **getters** and **setters**).
- **Abstraction:** The user of your class doesn't need to know how data is stored or managed internally — only how to interact with it.
- **Maintainability:** If you decide to change the internal implementation later, external code remains unaffected as long as the interface (methods) stays the same.
- **Reduced Complexity:** Encapsulation separates concerns — data management is kept safe inside the class, simplifying how other parts of the program interact with objects.

4.4.3 How to Encapsulate Data in Java

Step 1: Make Fields `private`

This restricts direct access from outside the class.

```java
public class Student {
    private String name;    // private field
    private int age;        // private field
}
```

Step 2: Provide Public Getter and Setter Methods

These allow controlled access and modification of the fields.

Full runnable code:

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

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter for age with validation
    public void setAge(int age) {
        if (age >= 0) {            // simple validation
            this.age = age;
        }
    }
}
// Example: Using Encapsulation

public class StudentDemo {
    public static void main(String[] args) {
        Student student = new Student();

        // Using setters to set values
        student.setName("John");
        student.setAge(20);

        // Using getters to retrieve values
        System.out.println("Name: " + student.getName());
        System.out.println("Age: " + student.getAge());

        // Attempt to set invalid age
        student.setAge(-5); // Age remains unchanged due to validation
        System.out.println("Updated Age: " + student.getAge());
    }
}
```

Output:

```
Name: John
Age: 20
Updated Age: 20
```

4.4.4 How Encapsulation Supports Maintainability and Reduces Complexity

Imagine if the `Student` class allowed direct access to the `age` field — any part of your program could set it to an invalid number (like `-5`), causing errors or inconsistencies.

By using **private fields** and **setters with validation**, you can:

- Prevent invalid data from entering your objects.
- Change the way you store or compute fields internally without affecting external code.
- Easily add logging, debugging, or other features inside getters and setters later.

This **modular approach** makes your code easier to understand, test, and modify.

4.4.5 Encouragement: Refactor Your Previous Classes

If you have written classes like `Person`, `Book`, or `Car` without encapsulation, try refactoring them:

- Change their fields to `private`.
- Add getter and setter methods.
- Optionally, add validation or other logic inside setters.
- Test how you interact with objects using these methods instead of direct field access.

4.4.6 Summary

- **Encapsulation** hides internal data by making fields `private`.
- Public **getter and setter methods** provide controlled access.
- It protects data integrity, improves code modularity, and supports abstraction.
- Encapsulation simplifies program maintenance and reduces complexity.

By practicing encapsulation, you will write safer, clearer, and more professional Java code — a vital skill as your programs grow larger and more complex.

Chapter 5.
Data Structures Arrays and Strings

5 Data Structures Arrays and Strings

5.1 One-Dimensional and Multidimensional Arrays

In programming, it's common to work with collections of related data. An **array** is one of the most basic and useful data structures for storing multiple values of the same type in a single, organized place.

In this section, we'll introduce **one-dimensional arrays** and then extend to **multidimensional arrays**, especially two-dimensional arrays. You'll learn how to declare, create, and access elements, along with examples of looping through arrays and a simple exercise to reinforce these concepts.

5.1.1 What Is an Array?

An **array** is a **fixed-size** collection of elements, all of the same type. Once created, the size of the array cannot change.

Think of an array as a row of numbered boxes where you can store values. Each box is called an **element**, and its position is called an **index**.

5.1.2 One-Dimensional Arrays

Declaring an Array

To declare a one-dimensional array, you specify the type and use square brackets:

```
int[] numbers;
```

This creates a variable `numbers` that can hold an array of integers but doesn't create the array yet.

Instantiating an Array

You create the actual array with the `new` keyword and specify its size:

```
numbers = new int[5];  // An array that holds 5 integers
```

You can also combine declaration and instantiation:

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

Initializing and Accessing Elements

You can assign values to array elements using their index, which starts at 0:

```
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
```

To access or print elements:

```
System.out.println(numbers[0]);  // Prints 10
```

Example: Looping Through a One-Dimensional Array

```
int[] numbers = {5, 10, 15, 20, 25};  // Initialize with values

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

Output:

```
Element at index 0 is 5
Element at index 1 is 10
Element at index 2 is 15
Element at index 3 is 20
Element at index 4 is 25
```

5.1.3 Multidimensional Arrays

A **two-dimensional array** can be thought of as a grid or table — rows and columns of elements.

Declaring a 2D Array

```
int[][] matrix;  // Declaration of a 2D integer array
```

Instantiating a 2D Array

```
matrix = new int[3][4];  // 3 rows, 4 columns
```

You can also combine these:

```
int[][] matrix = new int[3][4];
```

Accessing Elements in a 2D Array

You use **two indices** — one for the row and one for the column:

```
matrix[0][0] = 1;   // First row, first column
matrix[2][3] = 10;  // Third row, fourth column
```

Example: Nested Loop to Iterate Over a 2D Array

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

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

Output:

```
1 2 3
4 5 6
7 8 9
```

Full runnable code:

```
public class Test {
    public static void main(String[] args) {
        int[][] matrix = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

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

5.1.4 Exercise: Summing Elements of an Array

Try this small task:

- Create an array of integers.
- Use a loop to sum all the elements.
- Print the total sum.

Example code snippet:

```
int[] values = {3, 7, 2, 9, 4};
int sum = 0;

for (int i = 0; i < values.length; i++) {
    sum += values[i];
}
```

```
System.out.println("Sum of elements: " + sum);
```

Expected output:

```
Sum of elements: 25
```

5.1.5 Exercise: Printing a 2D Grid

Try printing a 5x5 grid of stars (*) using nested loops:

```
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++) {
        System.out.print("* ");
    }
    System.out.println();
}
```

Output:

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

5.1.6 Summary

- Arrays store **fixed-size** collections of elements of the **same type**.
- **One-dimensional arrays** are like lists indexed from 0 to length-1.
- **Two-dimensional arrays** are arrays of arrays, representing grids.
- You can access elements using index notation (`arr[index]` or `matrix[row][col]`).
- Loops (especially `for` loops) are essential to process arrays efficiently.
- Practicing with summing or printing grids helps solidify your understanding.

Mastering arrays is crucial since they form the foundation for more advanced data structures and algorithms in Java.

5.2 Common Array Algorithms

Arrays are fundamental in programming, but just storing data isn't enough — you need to **process** it efficiently. This section walks you through some essential array algorithms:

finding the maximum/minimum values, reversing arrays, searching (linear and binary), and sorting (selection or bubble sort). Along the way, you'll see how to think about algorithms and their efficiency, and then get a challenge to practice!

5.2.1 Finding Maximum and Minimum Values

To find the **maximum** or **minimum** in an array, you typically scan through all elements, updating your current max or min as you go.

5.2.2 Example: Find Maximum

```java
public static int findMax(int[] arr) {
    int max = arr[0];  // Assume first element is max initially
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];  // Update max if current element is bigger
        }
    }
    return max;
}
```

Similarly, for minimum, change the comparison to `arr[i] < min`.

5.2.3 Reversing an Array

Reversing means swapping elements from the start and end moving toward the center.

```java
public static void reverse(int[] arr) {
    int left = 0;
    int right = arr.length - 1;

    while (left < right) {
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
        left++;
        right--;
    }
}
```

5.2.4 Linear Search

A simple search algorithm that checks each element until it finds the target or reaches the end.

```java
public static int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) {
            return i;  // Return index if found
        }
    }
    return -1;  // Not found
}
```

Efficiency: Runs in **O(n)** time, checking each element once.

5.2.5 Binary Search

Used on **sorted arrays**, binary search cuts the search space in half each time.

```java
public static int binarySearch(int[] arr, int target) {
    int left = 0;
    int right = arr.length - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;  // Not found
}
```

Efficiency: Runs in **O(log n)** time — much faster than linear search for large arrays.

5.2.6 Sorting Arrays

Sorting organizes elements in ascending (or descending) order. Two simple sorting algorithms are **selection sort** and **bubble sort**.

5.2.7 Selection Sort

Finds the smallest element and swaps it with the first unsorted element, then repeats.

```java
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int minIndex = i;

        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        // Swap
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
```

5.2.8 Bubble Sort

Repeatedly swaps adjacent elements if they are in the wrong order, “bubbling” the largest element to the end.

```java
public static void bubbleSort(int[] arr) {
    boolean swapped;

    for (int i = 0; i < arr.length - 1; i++) {
        swapped = false;

        for (int j = 0; j < arr.length - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }

        // If no swaps, array is sorted
        if (!swapped) break;
    }
}
```

Full runnable code:

```java
public class ArrayAlgorithms {

    public static int findMax(int[] arr) {
        int max = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }
        return max;
    }

    public static int findMin(int[] arr) {
        int min = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] < min) {
                min = arr[i];
            }
        }
        return min;
    }

    public static void reverse(int[] arr) {
        int left = 0, right = arr.length - 1;
        while (left < right) {
            int temp = arr[left];
            arr[left] = arr[right];
            arr[right] = temp;
            left++;
            right--;
        }
    }

    public static int linearSearch(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) return i;
        }
        return -1;
    }

    public static int binarySearch(int[] arr, int target) {
        int left = 0, right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == target) return mid;
            else if (arr[mid] < target) left = mid + 1;
            else right = mid - 1;
        }
        return -1;
    }

    public static void selectionSort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
```

```java
    }

    public static void bubbleSort(int[] arr) {
        boolean swapped;
        for (int i = 0; i < arr.length - 1; i++) {
            swapped = false;
            for (int j = 0; j < arr.length - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }
            if (!swapped) break;
        }
    }

    public static void printArray(int[] arr) {
        for (int val : arr) {
            System.out.print(val + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        int[] array = {34, 7, 23, 32, 5, 62};

        System.out.println("Original array:");
        printArray(array);

        System.out.println("Maximum: " + findMax(array));
        System.out.println("Minimum: " + findMin(array));

        System.out.println("Reversed array:");
        reverse(array);
        printArray(array);

        int searchVal = 23;
        System.out.println("Linear search for " + searchVal + ": " + linearSearch(array, searchVal));

        selectionSort(array);
        System.out.println("Selection sorted:");
        printArray(array);

        System.out.println("Binary search for " + searchVal + ": " + binarySearch(array, searchVal));

        int[] array2 = {12, 3, 45, 2, 18};
        bubbleSort(array2);
        System.out.println("Bubble sorted another array:");
        printArray(array2);
    }
}
```

5.2.9 Reflecting on Algorithmic Thinking and Efficiency

- **Algorithmic thinking** means breaking down a problem into clear, logical steps.
- Consider **efficiency**: how fast and resource-friendly your algorithm is.
- For example, binary search is much faster than linear search but requires sorted data.
- Sorting algorithms vary in speed; bubble sort is simple but slow for large datasets.

5.2.10 Mini Challenge: Implement a Custom Array Filter

Try this on your own:

- Write a method `filterEvenNumbers(int[] arr)` that returns a new array containing only the even numbers from the input array.
- Hint: You might first count how many even numbers there are, create a new array of that size, then copy over the even elements.

5.2.11 Summary

- **Find max/min:** Traverse once, keep track of extreme values.
- **Reverse array:** Swap pairs moving inward.
- **Linear search:** Simple but slow search through each element.
- **Binary search:** Fast search on sorted arrays by dividing the search space.
- **Selection and bubble sort:** Basic sorting methods illustrating swapping and comparison.
- **Algorithm efficiency:** Important to consider for large datasets.
- **Practice:** Try filtering or transforming arrays yourself.

Mastering these fundamental algorithms will give you a strong foundation to understand more complex data structures and algorithms later.

5.3 Working with Strings and StringBuilder

Strings are one of the most common data types you'll use in Java. They allow you to store and manipulate text, from simple words to entire sentences. In this section, we'll explore how strings work in Java, why they are **immutable**, and how the **StringBuilder** class offers an efficient way to modify text.

5.3.1 How Strings Work in Java

In Java, a **String** is an object that represents a sequence of characters, such as `"Hello, world!"`. Java stores strings in a special memory area called the **string pool**. When you create a string literal, Java checks the pool first to reuse existing strings, improving performance and memory usage.

5.3.2 Immutability of Strings

Strings in Java are **immutable**, meaning once created, they **cannot be changed**. Any operation that seems to modify a string actually creates a **new string object**.

Example:

```java
String s = "Hello";
s = s + " World";  // Creates a new string "Hello World"
```

The original `"Hello"` string remains unchanged, and `s` now points to the new string.

5.3.3 Common String Methods

Here are some useful methods to work with strings:

5.3.4 `.length()`

Returns the number of characters in the string.

```java
String text = "Java";
System.out.println(text.length());  // Output: 4
```

5.3.5 `.charAt(int index)`

Returns the character at the specified position (index starts at 0).

```java
char c = text.charAt(1);
System.out.println(c);  // Output: 'a'
```

5.3.6 `.substring(int beginIndex, int endIndex)`

Returns a substring from `beginIndex` up to (but not including) `endIndex`.

```java
String sub = text.substring(1, 3);
System.out.println(sub);  // Output: "av"
```

5.3.7 `.indexOf(String str)`

Returns the index of the first occurrence of the specified substring, or -1 if not found.

```java
int pos = text.indexOf("va");
System.out.println(pos);  // Output: 1
```

5.3.8 String Concatenation and Performance

Using the `+` operator to join strings is easy but can be **inefficient** if done repeatedly inside loops because it creates many temporary String objects.

5.3.9 Introducing StringBuilder for Mutable Strings

`StringBuilder` is a class designed to create and modify strings **efficiently**. Unlike `String`, a `StringBuilder` object is **mutable**, so you can append, insert, or delete characters without creating new objects each time.

5.3.10 Creating and Using a StringBuilder

```java

StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb.toString());  // Output: Hello World
```

You can chain calls:

```java
sb.append("!").append(" How are you?");
```

5.3.11 Modifying Text with StringBuilder

- `.append(String str)` – Adds text to the end.
- `.insert(int offset, String str)` – Inserts text at a specific position.
- `.replace(int start, int end, String str)` – Replaces a part of the string.
- `.delete(int start, int end)` – Deletes characters in a range.
- `.toString()` – Converts back to a regular `String`.

5.3.12 Example: Building a Sentence with StringBuilder

Full runnable code:

```java
public class StringBuilderDemo {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Java");

        sb.append(" is");
        sb.append(" fun");
        sb.insert(9, " really");
        sb.replace(0, 4, "Programming");

        System.out.println(sb.toString());
    }
}
```

Output:

```
Programming is really fun
```

5.3.13 Comparison Table: String vs. StringBuilder

Feature	String	StringBuilder
Mutability	Immutable	Mutable
Modification	Creates new object on change	Modifies same object
Performance (repeated concatenation)	Slow (due to object creation)	Fast (efficient in loops)
Thread safety	Thread-safe (immutable)	Not thread-safe
Common use cases	Fixed or rarely changed text	Building/modifying text dynamically

5.3.14 Summary

- Strings in Java are **immutable** objects stored in a **string pool** for memory efficiency.
- Basic `String` methods like `.length()`, `.charAt()`, `.substring()`, and `.indexOf()` help you inspect and extract parts of text.
- Using `+` for string concatenation creates many objects and can hurt performance in loops.
- `StringBuilder` is a mutable alternative, allowing efficient text building and modifications without creating new objects repeatedly.
- Choose `String` for fixed text, and `StringBuilder` when you need to **build or modify strings dynamically**.

Try experimenting with `String` and `StringBuilder` in your own programs to see how they behave differently and to improve the performance of your text-processing code.

5.4 String Comparison and Manipulation

Working with strings often requires comparing their content and manipulating their text. Java provides several ways to compare strings and methods to perform common string transformations. This section explains the differences between these approaches and shows practical examples.

5.4.1 Comparing Strings: `.equals()`, `.equalsIgnoreCase()`, and `==`

`.equals()` Comparing Content

The `.equals()` method compares **the actual content** of two strings:

```
String a = "Java";
String b = new String("Java");

System.out.println(a.equals(b));  // true, because content is the same
```

`.equalsIgnoreCase()` Case-Insensitive Comparison

If you need to compare strings ignoring uppercase or lowercase differences, use `.equalsIgnoreCase()`:

```
String a = "Java";
String b = "java";

System.out.println(a.equalsIgnoreCase(b));  // true
```

== Comparing References (Memory Locations)

Using `==` checks if **two variables point to the exact same object** in memory, not if their contents match:

```java
String a = "Java";
String b = new String("Java");

System.out.println(a == b);  // false, different objects
```

Pitfall: Since many strings can have the same content but be different objects, `==` can give unexpected results when comparing strings.

5.4.2 Practical String Manipulation Methods

Removing Whitespace: `.trim()`

Removes leading and trailing spaces from a string.

```java
String input = "  Hello World  ";
System.out.println("[" + input.trim() + "]");  // Output: [Hello World]
```

Splitting Strings: `.split()`

Splits a string into parts based on a delimiter, returning an array.

```java
String csv = "apple,banana,orange";
String[] fruits = csv.split(",");

for (String fruit : fruits) {
    System.out.println(fruit);
}
```

Output:

```
apple
banana
orange
```

Replacing Substrings: `.replace()`

Replaces occurrences of a target substring with another string.

```java
String text = "I like cats";
String newText = text.replace("cats", "dogs");
System.out.println(newText);  // Output: I like dogs
```

5.4.3 Exercise: Check if a String is a Palindrome

A **palindrome** reads the same forwards and backwards, like `"madam"`.

```java
public static boolean isPalindrome(String str) {
    str = str.toLowerCase().replaceAll("\\s+", ""); // Normalize: lowercase, remove spaces

    int left = 0;
    int right = str.length() - 1;

    while (left < right) {
        if (str.charAt(left) != str.charAt(right)) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

Example usage:

```java
System.out.println(isPalindrome("Race car"));  // true
System.out.println(isPalindrome("Hello"));     // false
```

Full runnable code:

```java
public class PalindromeChecker {

    public static boolean isPalindrome(String str) {
        str = str.toLowerCase().replaceAll("\\s+", ""); // Normalize: lowercase, remove spaces

        int left = 0;
        int right = str.length() - 1;

        while (left < right) {
            if (str.charAt(left) != str.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }
        return true;
    }

    public static void main(String[] args) {
        System.out.println(isPalindrome("Race car"));  // true
        System.out.println(isPalindrome("Hello"));     // false
        System.out.println(isPalindrome("A Santa at NASA")); // true
    }
}
```

5.4.4 Summary

- Use `.equals()` to compare **string content**.
- Use `.equalsIgnoreCase()` for case-insensitive comparisons.
- Avoid `==` for strings unless you want to check if both variables refer to the exact same

object.
- Common string manipulations include `.trim()`, `.split()`, and `.replace()`.
- String processing tasks, like checking for palindromes, combine these methods with loops and conditions.

By understanding how to correctly compare and manipulate strings, you'll avoid bugs and write cleaner, more effective Java programs.

Chapter 6.
Encapsulation and Access Control

6 Encapsulation and Access Control

6.1 Getters and Setters

In Java, **encapsulation** means keeping the internal details of a class hidden and exposing only what is necessary through well-defined interfaces. One common way to enforce encapsulation is by making fields (variables) **private** and providing public **getter** and **setter** methods to access and modify these fields. This approach ensures controlled access to the data and improves code maintainability.

6.1.1 What Are Getters and Setters?

- A **getter** method (also called an accessor) returns the value of a private field.
- A **setter** method (also called a mutator) sets or updates the value of a private field.

Together, getters and setters provide controlled access to fields without exposing them directly.

6.1.2 Why Use Getters and Setters?

- **Control:** You can add logic inside setters to validate inputs before assigning values.
- **Read-only or Write-only access:** You may provide only getters (read-only) or only setters (write-only) depending on the need.
- **Maintainability:** Changing how a field is stored or computed internally won't affect external code using your getters and setters.
- **Encapsulation:** Fields stay private and protected from unintended changes.

6.1.3 Naming Conventions

Getters and setters follow a standard naming pattern:

- For a field `fieldName`:
  - Getter: `getFieldName()` (returns the field's value)
  - Setter: `setFieldName(type value)` (updates the field's value)

If the field is a boolean, the getter can also be named `isFieldName()`.

Example:

```java
private int age;

public int getAge() { return age; }
public void setAge(int age) { this.age = age; }
```

6.1.4 Example Class Using Getters and Setters

Here's a simple class `Person` with private fields and public getters and setters:

Full runnable code:

```java
class Person {
    private String name;
    private int age;

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter for age with validation
    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        } else {
            System.out.println("Age cannot be negative.");
        }
    }
}

// Using Getters and Setters in `main`

public class Main {
    public static void main(String[] args) {
        Person person = new Person();

        // Set values using setters
        person.setName("Alice");
        person.setAge(30);

        // Try setting an invalid age
        person.setAge(-5);  // Prints: Age cannot be negative.

        // Get values using getters
        System.out.println("Name: " + person.getName());  // Name: Alice
        System.out.println("Age: " + person.getAge());    // Age: 30
    }
}
```

6.1.5 Auto-Generating Getters and Setters

Most modern IDEs like Eclipse, IntelliJ IDEA, and NetBeans can **auto-generate** getters and setters for you. This saves time and ensures consistent naming and formatting.

For example, in IntelliJ IDEA, you can:

- Right-click inside the class.
- Choose **Generate > Getter and Setter**.
- Select the fields.
- The IDE will create all required methods.

6.1.6 Summary

- Getters and setters are methods to **read and write private fields** safely.
- They **enforce encapsulation** by controlling how fields are accessed and modified.
- Follow naming conventions: `getFieldName()`, `setFieldName()`, or `isFieldName()` for booleans.
- Use setters to add validation or restrict updates.
- IDEs simplify creating these methods, encouraging good encapsulation practices.

Try creating your own classes with private fields and use getters and setters to interact with them. This foundational skill will improve your ability to write clean, secure, and maintainable Java code.

6.2 Access Modifiers: public, private, protected

Java uses **access modifiers** to control the visibility of classes, fields, methods, and constructors. These modifiers help you enforce encapsulation and protect data by restricting where different parts of your program can access certain members.

6.2.1 The Four Access Modifiers

Java has **four** access levels:

Modifier	Visible in Same Class	Same Package	Subclass (any package)	Other Packages (non-subclass)
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
default (no modifier)	Yes	Yes	No	No
`private`	Yes	No	No	No

6.2.2 `public`

- Members declared `public` can be accessed **from anywhere**, both inside and outside the package.
- Use it for methods or fields you want available to any code that uses your class.

Example:

```java
public class BankAccount {
    public String accountNumber;
}
```

6.2.3 `private`

- The most restrictive modifier.
- Members are accessible **only within the same class**.
- Use for sensitive data or implementation details you want to hide completely.

Example:

```java
public class BankAccount {
    private double balance;
}
```

6.2.4 `protected`

- Members are accessible within the **same package** and by **subclasses** even if those subclasses are in different packages.
- Use `protected` for data or methods intended for subclasses but hidden from unrelated classes.

Example:

```java
protected double interestRate;
```

6.2.5 Default (Package-Private)

- When no modifier is specified, it's accessible **only within the same package**.
- Use it when classes or members should be shared with closely related classes but hidden from the rest of the program.

Example:

```java
int transactionCount;  // accessible only in the package
```

6.2.6 Practical Example: BankAccount Class

```java
package banking;

public class BankAccount {
    private String accountNumber;    // Hidden from all except BankAccount
    private double balance;          // Private balance

    protected double interestRate;   // Visible to subclasses & same package

    public BankAccount(String accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
        this.interestRate = 0.01;   // 1% interest rate
    }

    // Public method to access balance
    public double getBalance() {
        return balance;
    }

    // Public method to deposit money
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    // Private method (used internally)
    private void applyInterest() {
        balance += balance * interestRate;
    }
}
```

6.2.7 Accessing Members from Different Contexts

Assuming the above `BankAccount` class is in package `banking`:

6.2.8 From the Same Class

Inside `BankAccount`, **all members** (`private`, `protected`, `public`) are accessible.

6.2.9 From Another Class in the Same Package

```java
package banking;

public class BankManager {
    public void checkAccount() {
        BankAccount acc = new BankAccount("1234", 1000);

        // acc.accountNumber -> Not accessible (private)
        // acc.balance -> Not accessible (private)
        System.out.println(acc.getBalance());    // Accessible (public)
        System.out.println(acc.interestRate);    // Accessible (protected + same package)
    }
}
```

6.2.10 From a Subclass in a Different Package

```java
package vipbanking;

import banking.BankAccount;

public class VIPAccount extends BankAccount {

    public VIPAccount(String accNum, double balance) {
        super(accNum, balance);
        System.out.println(interestRate);  // Accessible (protected)
    }

    public void showAccountNumber() {
        // System.out.println(accountNumber); // Error: private field not accessible
    }
}
```

6.2.11 From an Unrelated Class in a Different Package

```java
package publicapp;

import banking.BankAccount;

public class App {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount("9999", 5000);

        // acc.balance -> Not accessible (private)
        // acc.interestRate -> Not accessible (protected + different package + not subclass)
        System.out.println(acc.getBalance());  // Accessible (public)
    }
}
```

6.2.12 When to Use Each Modifier?

Modifier	Use Case
`private`	Protect sensitive data, internal logic
default	Share among related classes in same package
`protected`	Allow subclass access, hide from unrelated
`public`	Expose API, methods intended for all clients

6.2.13 Summary

- Use **private** for maximum encapsulation and data protection.
- Use **protected** to allow subclass access while hiding from other classes.
- Use **default (package-private)** to restrict access within a package.
- Use **public** to make members visible everywhere.
- Understanding and applying access modifiers properly is key to **safe, maintainable** Java programs.

Try modifying the `BankAccount` class or create your own class using different access modifiers to observe their effects in various packages and subclasses!

6.3 Packages and Import Statements

In Java, as your projects grow, organizing your code becomes essential. **Packages** help you group related classes and interfaces together, making your code easier to manage, reuse, and avoid naming conflicts. This section explains how to create packages, use `import` statements to access classes across packages, and explores special import forms.

6.3.1 What Are Packages?

A **package** is essentially a folder (namespace) that contains related Java classes and interfaces. It acts like a container to organize your code logically.

6.3.2 Why Use Packages?

- Prevent class name conflicts (e.g., two classes named `User` in different packages).
- Make your code easier to maintain.
- Control access with package-private visibility.
- Help others understand your code structure.

6.3.3 Defining a Package

To place a class inside a package, you declare the package at the very top of your `.java` file.

```java
package com.example.models;

public class Person {
    private String name;

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

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

Here, `Person.java` belongs to the package `com.example.models`.

6.3.4 Importing Classes from Other Packages

To use classes from another package, you need to **import** them. The `import` statement tells Java where to find these classes.

```java
package com.example.app;

import com.example.models.Person;

public class MainApp {
    public static void main(String[] args) {
        Person person = new Person("Alice");
        System.out.println(person.getName());
    }
}
```

Without the `import`, you'd have to use the fully qualified name:

```java
com.example.models.Person person = new com.example.models.Person("Alice");
```

6.3.5 Example: Two Packages in Action

File: Person.java

```java
package com.example.models;

public class Person {
    private String name;

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

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

File: MainApp.java

```java
package com.example.app;

import com.example.models.Person;

public class MainApp {
    public static void main(String[] args) {
        Person p = new Person("Bob");
        System.out.println("Hello, " + p.getName());
    }
}
```

Here, `MainApp` in `com.example.app` imports and uses `Person` from `com.example.models`.

6.3.6 Wildcard Imports

Instead of importing each class individually, you can use a **wildcard (*)** to import all classes in a package:

```java
import com.example.models.*;
```

This imports all public classes inside `com.example.models`.

Note: Wildcard imports do not import subpackages automatically. For example, `com.example.models.*` won't import `com.example.models.subpackage`.

6.3.7 Static Imports

Java also allows **static imports** to directly use static members (methods or variables) without qualifying them by class name.

Example:

Full runnable code:

```java
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;

public class Circle {
    public double getCircumference(double radius) {
        return 2 * PI * radius;  // No need to write Math.PI
    }

    public double getSquareRoot(double value) {
        return sqrt(value);      // No need to write Math.sqrt
    }
}
```

6.3.8 Summary

- **Packages** organize related classes and help avoid name conflicts.
- Declare a package at the top of each `.java` file using `package your.package.name;`.
- Use `import` statements to bring other packages' classes into your code for easier referencing.
- Wildcard imports (`*`) bring in all classes in a package, but avoid overusing them to keep clarity.
- **Static imports** simplify using static methods and fields from other classes.

By structuring your code into packages and understanding imports, you'll create clear, modular, and maintainable Java applications.

Chapter 7.

Inheritance and Polymorphism

1. Understanding `extends` and `super`
2. Overriding Methods
3. Using `instanceof` and Type Casting
4. Dynamic Method Dispatch

7 Inheritance and Polymorphism

7.1 Understanding `extends` and `super`

Inheritance is a fundamental concept in object-oriented programming that allows one class to **inherit** fields and methods from another. In Java, inheritance is expressed using the `extends` keyword, enabling you to create hierarchical relationships between classes and promote **code reuse**.

7.1.1 What is Inheritance?

Inheritance allows a new class (called a **child** or **subclass**) to acquire properties and behaviors from an existing class (called the **parent** or **superclass**). This helps avoid duplication and makes your code more organized.

7.1.2 Using `extends` to Define a Child Class

The `extends` keyword indicates that a class inherits from another:

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

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

Here, `Dog` inherits the `eat()` method from `Animal` and also adds its own method `bark()`.

7.1.3 Example Usage

```java
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();   // Inherited from Animal
        dog.bark();  // Defined in Dog
    }
}
```

Output:

```
This animal eats food.
The dog barks.
```

Full runnable code:

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

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

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();   // Inherited from Animal
        dog.bark();  // Defined in Dog
    }
}
```

7.1.4 The super Keyword: Accessing Parent Class Members

`super` allows subclasses to refer to their superclass, which is useful in two common scenarios:

7.1.5 Calling the Parent Constructor

When a subclass has a constructor, it can call the parent class constructor using `super(...)`. This initializes the parent part of the object properly.

```java
public class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }
}

public class Dog extends Animal {
    public Dog(String name) {
        super(name);  // Calls Animal's constructor
    }
    public void printName() {
        System.out.println("Dog's name: " + name);
    }
}
```

Usage:

```
Dog dog = new Dog("Buddy");
dog.printName();  // Output: Dog's name: Buddy
```

7.1.6 Calling an Overridden Method from the Parent Class

If a subclass overrides a method, it can still access the original version using `super.methodName()`.

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

public class Dog extends Animal {
    @Override
    public void sound() {
        super.sound();  // Calls Animal's sound()
        System.out.println("Dog barks");
    }
}
```

Usage:

```
Dog dog = new Dog();
dog.sound();
```

Output:

```
Animal makes a sound
Dog barks
```

Full runnable code:

```
    class Animal {
        String name;

        public Animal(String name) {
            this.name = name;
        }

        public void sound() {
            System.out.println("Animal makes a sound");
        }
    }
    class Dog extends Animal {
        public Dog(String name) {
            super(name);
        }

        public void printName() {
            System.out.println("Dog's name: " + name);
        }

        @Override
        public void sound() {
            super.sound(); // Calls Animal's sound()
            System.out.println("Dog barks");
        }
    }
public class Main {
    public static void main(String[] args) {
        Dog dog1 = new Dog("Buddy");
        dog1.printName(); // Output: Dog's name: Buddy
        dog1.sound();     // Output: Animal makes a sound
                          //         Dog barks
    }
}
```

7.1.7 Benefits of Inheritance

- **Code Reuse:** Avoid rewriting code by inheriting fields and methods.
- **Logical Hierarchy:** Models real-world relationships (e.g., Dog **is an** Animal).
- **Extensibility:** Easily add new features in child classes without changing parent classes.

7.1.8 Summary

- Use `extends` to create a subclass that inherits from a superclass.
- The child class automatically gains the parent's public and protected fields and methods.
- Use `super(...)` to invoke a parent class constructor inside a subclass constructor.
- Use `super.methodName()` to call a parent's method that has been overridden.
- Inheritance simplifies code maintenance by promoting reuse and logical class hierarchies.

Try creating your own class hierarchies using `extends` and practice using `super` to access parent class constructors and methods. This foundational skill is key to mastering Java's object-oriented programming.

7.2 Overriding Methods

In object-oriented programming, **method overriding** allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This lets subclasses customize or extend behavior inherited from parent classes, enabling **polymorphism** and flexible code design.

7.2.1 What is Method Overriding?

When a subclass defines a method with the **same signature** (name, return type, and parameters) as a method in its superclass, it overrides that method. Calls to the method on an object of the subclass will execute the subclass's version instead of the superclass's.

7.2.2 The `@Override` Annotation

The `@Override` annotation is optional but highly recommended. It tells the compiler that you intend to override a method, enabling it to check your method signature for correctness and avoid mistakes such as misspellings or incorrect parameters.

Example:

```java
@Override
public void draw() {
    // subclass-specific code
}
```

If the method doesn't correctly override a superclass method, the compiler will raise an error.

7.2.3 Rules for Overriding

- Method name, return type, and parameters **must match** exactly.
- The overriding method cannot have more restrictive access than the overridden method (e.g., a `public` method cannot be overridden with `private`).
- The overriding method can throw fewer or more specific exceptions but not broader ones.

7.2.4 Example: Shape, Circle, and Rectangle

```
public class Shape {
    public void draw() {
        System.out.println("Drawing a generic shape");
    }
}

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

public class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}
```

7.2.5 Polymorphic Behavior with Overriding

Because all these classes share the same method signature, you can write code that treats all shapes uniformly while still calling their specific implementations:

```
public class Main {
    public static void main(String[] args) {
        Shape[] shapes = { new Shape(), new Circle(), new Rectangle() };

        for (Shape shape : shapes) {
            shape.draw();  // Calls the appropriate method based on the object's actual type
        }
    }
}
```

Output:

```
Drawing a generic shape
Drawing a circle
Drawing a rectangle
```

This demonstrates **polymorphism** — the ability of different objects to respond uniquely to the same method call.

Full runnable code:

```
    class Shape {
        public void draw() {
            System.out.println("Drawing a generic shape");
        }
    }

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

    class Rectangle extends Shape {
        @Override
        public void draw() {
            System.out.println("Drawing a rectangle");
        }
    }


public class Main {


    public static void main(String[] args) {
        Shape[] shapes = { new Shape(), new Circle(), new Rectangle() };

        for (Shape shape : shapes) {
            shape.draw();  // Polymorphic call
        }
    }
}
```

7.2.6 Why Is Overriding Important?

- Enables **flexible and extensible** code designs.
- Supports **runtime polymorphism** where method calls are resolved based on object type.
- Allows subclasses to **customize** or **extend** the behavior of inherited methods.

7.2.7 Summary

- Method overriding means redefining a superclass method in a subclass with the same signature.
- Use the `@Override` annotation to help catch errors and improve readability.
- Overridden methods enable polymorphic behavior, making your code adaptable and scalable.
- Always ensure method signatures match and access levels are compatible.

Experiment by creating your own class hierarchy with overridden methods and observe how Java chooses which method to call at runtime. This technique is foundational for advanced

Java programming.

7.3 Using `instanceof` and Type Casting

In Java's inheritance hierarchy, objects of subclasses can be treated as instances of their superclass, but sometimes you need to identify an object's actual type at **runtime** or convert references between classes. Java provides the **`instanceof`** operator for type checking and supports **type casting** to convert references safely.

7.3.1 What is `instanceof`?

The `instanceof` operator tests whether an object is an instance of a specified class or interface. It returns `true` if the object can be safely treated as that type, otherwise `false`.

Syntax:

```
objectRef instanceof ClassName
```

Example:

```
if (obj instanceof Dog) {
    System.out.println("obj is a Dog");
}
```

7.3.2 Upcasting and Downcasting Explained

Upcasting (Implicit)

Assigning a subclass object to a superclass reference is called **upcasting** and happens automatically (implicitly):

```
Dog dog = new Dog();
Animal animal = dog;  // Upcasting Dog to Animal
```

Upcasting is safe because a `Dog` **is an** `Animal`, so you can use the `animal` reference to access only the members defined in `Animal`.

Downcasting (Explicit)

Assigning a superclass reference back to a subclass reference is called **downcasting** and requires an explicit cast because it's potentially unsafe:

```
Animal animal = new Dog();
Dog dog = (Dog) animal;  // Downcasting Animal to Dog
```

If `animal` actually refers to a `Dog`, the cast works. If it refers to a different subclass or a plain `Animal`, a **`ClassCastException`** will be thrown at runtime.

Using `instanceof` to Ensure Safe Downcasting

Before downcasting, you should check the object's type using `instanceof` to avoid exceptions:

```java
if (animal instanceof Dog) {
    Dog dog = (Dog) animal;  // Safe downcast
    dog.bark();
} else {
    System.out.println("The animal is not a dog");
}
```

7.3.3 Practical Example with Inheritance Hierarchy

Full runnable code:

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

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

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();  // Upcasting

        myAnimal.makeSound();  // Animal method or overridden version

        // myAnimal.bark(); // Compile error: method not visible via Animal reference

        if (myAnimal instanceof Dog) {
            Dog myDog = (Dog) myAnimal;  // Downcasting after check
            myDog.bark();
        }

        Animal anotherAnimal = new Cat();

        if (anotherAnimal instanceof Dog) {
            Dog dog = (Dog) anotherAnimal;  // This block won't execute
        } else {
            System.out.println("Not a Dog, cannot cast");
        }
    }
}
```

Output:

```
Animal makes a sound
Dog barks
Not a Dog, cannot cast
```

7.3.4 Common Pitfalls and Best Practices

- **Never downcast without an `instanceof` check**, or risk a runtime `ClassCastException`.
- Avoid excessive use of `instanceof`; it can indicate design issues. Prefer **polymorphism** (overriding methods) to handle different behaviors.
- Remember that **upcasting is implicit and safe**, but limits accessible methods to those of the superclass.
- `instanceof` returns `false` if the object is `null`.

7.3.5 Summary

- Use `instanceof` to check an object's actual type at runtime safely.
- Upcasting (subclass → superclass) happens implicitly and is safe.
- Downcasting (superclass → subclass) requires an explicit cast and should be guarded with `instanceof`.
- Careful use of type checks and casts avoids runtime errors and supports robust Java code.

Try experimenting by creating your own class hierarchy and practice safe casting with `instanceof`. Understanding these concepts is vital for mastering Java's polymorphic behaviors.

7.4 Dynamic Method Dispatch

Java's power in object-oriented programming largely comes from its ability to support **polymorphism**, allowing a single reference type to point to objects of multiple types. The core mechanism enabling this is called **dynamic method dispatch**. This runtime process decides which overridden method implementation to execute based on the actual object type, not the declared reference type.

7.4.1 What is Dynamic Method Dispatch?

Dynamic method dispatch is Java's way of selecting the correct version of an overridden method when a superclass reference points to a subclass object. At **compile time**, the compiler checks that the method exists in the reference type, but the **actual method called at runtime** depends on the object's real class.

This makes Java's method calls **dynamic** and flexible.

7.4.2 Example: Shape Drawing

Consider a simple hierarchy of shapes:

```java
class Shape {
    public void draw() {
        System.out.println("Drawing a generic shape");
    }
}

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

class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}
```

Here, all subclasses override the `draw()` method to provide their specific drawing behavior.

7.4.3 Demonstration of Dynamic Dispatch

```java
public class Main {
    public static void main(String[] args) {
        Shape shape;

        shape = new Shape();
        shape.draw();  // Calls Shape's draw()
        shape = new Circle();
        shape.draw();  // Calls Circle's draw()

        shape = new Rectangle();
        shape.draw();  // Calls Rectangle's draw()
    }
}
```

Output:

```
Drawing a generic shape
Drawing a circle
Drawing a rectangle
```

Despite `shape` being declared as type `Shape`, the **runtime object type** determines which `draw()` method runs.

Full runnable code:

```
class Shape {
    public void draw() {
        System.out.println("Drawing a generic shape");
    }
}

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

class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape shape;

        shape = new Shape();
        shape.draw();  // Drawing a generic shape

        shape = new Circle();
        shape.draw();  // Drawing a circle

        shape = new Rectangle();
        shape.draw();  // Drawing a rectangle
    }
}
```

7.4.4 Why Does This Matter?

If Java used the reference type to decide which method to call (static dispatch), the output would always be:

```
Drawing a generic shape
Drawing a generic shape
Drawing a generic shape
```

This would ignore the subclasses' custom behavior, making polymorphism impossible.

Dynamic dispatch enables:

- **Flexibility:** One piece of code can work with multiple object types.
- **Extensibility:** Add new subclasses without changing existing code.
- **Code Reuse:** Methods can be designed to operate on superclass types while invoking subclass-specific behavior.

7.4.5 How Does Dynamic Dispatch Work Internally?

- When a method is called on an object, the JVM looks up the method in the **actual object's class**.
- It uses a method table (vtable) to find the correct overridden method.
- This lookup happens at runtime, enabling the correct method execution.

7.4.6 Practical Use: Polymorphic Collections

```java
Shape[] shapes = { new Circle(), new Rectangle(), new Shape() };

for (Shape s : shapes) {
    s.draw();  // Calls the appropriate draw() method dynamically
}
```

Output:

```
Drawing a circle
Drawing a rectangle
Drawing a generic shape
```

This pattern is common in frameworks, GUI toolkits, and APIs where objects behave differently but share a common interface.

7.4.7 Summary

- **Dynamic method dispatch** is Java's runtime method selection based on actual object type.
- It ensures overridden methods in subclasses are called even when accessed through a superclass reference.
- This is fundamental to Java's **polymorphism**, allowing flexible, extensible designs.
- Without dynamic dispatch, Java couldn't provide the elegant and powerful object-oriented behaviors it's known for.

Experiment by creating your own class hierarchy with overridden methods and try calling those methods through superclass references. Watch how Java dynamically selects the correct implementation — this is the heart of polymorphism in Java!

Chapter 8.
Abstract Classes and Interfaces

8 Abstract Classes and Interfaces

8.1 Abstract Methods and Classes

In Java, **abstract classes** and **abstract methods** offer a powerful way to define common behavior while leaving some parts incomplete—forcing subclasses to fill in the details. This approach provides a flexible blueprint for creating related classes with shared features, without allowing direct instantiation of the abstract class itself.

8.1.1 What is an Abstract Class?

An **abstract class** is a class that cannot be instantiated directly. It can contain both **concrete methods** (with full implementations) and **abstract methods** (without implementations). Abstract classes are declared with the keyword `abstract`.

```java
public abstract class Vehicle {
    // Abstract method (no implementation)
    public abstract void move();

    // Concrete method (implemented)
    public void start() {
        System.out.println("Vehicle is starting");
    }
}
```

Because `Vehicle` is abstract, you cannot create objects of type `Vehicle`:

```java
Vehicle v = new Vehicle();  // Compilation error!
```

8.1.2 What is an Abstract Method?

An **abstract method** declares a method signature but no body. It tells subclasses, "You must provide your own implementation of this method."

Example:

```java
public abstract void move();
```

Every concrete subclass **must override** and provide an implementation for all abstract methods inherited from its abstract superclass unless the subclass is also abstract.

8.1.3 When to Use Abstract Classes?

Use abstract classes when:

- You want to provide **common fields or methods** shared by all subclasses.
- You want to define a **contract** that subclasses must follow by implementing abstract methods.
- You want to **partially implement** a concept, leaving some behavior undefined.

If all methods are abstract and no state (fields) is needed, consider using an interface instead (covered later).

8.1.4 Example: Abstract `Vehicle` Class and Subclasses

```java
// Abstract superclass
public abstract class Vehicle {
    public abstract void move();  // Abstract method

    public void start() {          // Concrete method
        System.out.println("Vehicle is starting");
    }
}

// Concrete subclass Car
public class Car extends Vehicle {
    @Override
    public void move() {
        System.out.println("Car is driving on the road");
    }
}

// Concrete subclass Bike
public class Bike extends Vehicle {
    @Override
    public void move() {
        System.out.println("Bike is pedaling on the path");
    }
}
```

8.1.5 Using the Classes

```java
public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        Vehicle myBike = new Bike();

        myCar.start();
        myCar.move();
        myBike.start();
        myBike.move();
    }
}
```

Output:

```
Vehicle is starting
Car is driving on the road
Vehicle is starting
Bike is pedaling on the path
```

Notice that even though `myCar` and `myBike` are both declared as `Vehicle` references, the correct subclass implementations of `move()` are called. This is polymorphism working hand-in-hand with abstraction.

Full runnable code:

```
abstract class Vehicle {
    public abstract void move();  // Abstract method

    public void start() {         // Concrete method
        System.out.println("Vehicle is starting");
    }
}

class Car extends Vehicle {
    @Override
    public void move() {
        System.out.println("Car is driving on the road");
    }
}

class Bike extends Vehicle {
    @Override
    public void move() {
        System.out.println("Bike is pedaling on the path");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        Vehicle myBike = new Bike();

        myCar.start();
        myCar.move();

        myBike.start();
        myBike.move();
    }
}
```

8.1.6 Benefits of Abstract Classes

- **Code reuse:** Put shared code (like `start()`) in one place.
- **Design clarity:** Enforce that subclasses provide implementations for essential behavior.
- **Safe abstraction:** Prevent creating generic objects that don't make sense (e.g., a plain `Vehicle`).
- **Extensibility:** New types of vehicles can easily be added by subclassing and providing specific implementations.

8.1.7 Summary

- Abstract classes cannot be instantiated and may contain abstract methods without bodies.
- Abstract methods force subclasses to implement specific behavior.
- Use abstract classes to provide a common base with shared code plus some required customization.
- Abstract classes enable clean, maintainable designs that promote code reuse and logical hierarchy.

Try creating your own abstract class and subclasses to practice how abstract methods enforce behavior while allowing flexible implementations.

8.2 Implementing Interfaces

In Java, **interfaces** define a **contract** that classes agree to follow by implementing specific methods. Unlike classes, interfaces only specify *what* should be done, not *how* to do it. This helps create flexible, loosely coupled code where multiple classes can share common behaviors without inheriting from the same superclass.

8.2.1 What is an Interface?

An interface is like a **blueprint** for classes. It declares method signatures that implementing classes must provide. This ensures consistency and allows code to rely on these common methods without needing to know the exact class details.

8.2.2 Defining an Interface

The syntax for defining an interface is:

```java
public interface Flyable {
    void fly();  // abstract method
}
```

All methods in an interface are implicitly `public` and abstract (before Java 8). Classes that implement this interface must provide a concrete `fly()` method.

8.2.3 Implementing an Interface

To implement an interface, a class uses the `implements` keyword:

```java
public class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Bird is flying");
    }
}
```

The class **must** implement all methods declared in the interface unless the class is abstract.

8.2.4 Multiple Interface Implementation

Java allows a class to implement **multiple interfaces**, enabling the class to have diverse capabilities:

```java
public interface Swimmable {
    void swim();
}

public class Fish implements Swimmable {
    @Override
    public void swim() {
        System.out.println("Fish is swimming");
    }
}

public class Duck implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("Duck is flying");
    }

    @Override
    public void swim() {
        System.out.println("Duck is swimming");
    }
}
```

8.2.5 Default Methods (Java 8)

Since Java 8, interfaces can include **default methods** — methods with a body that implementing classes inherit automatically:

```java
public interface Flyable {
    void fly();

    default void takeOff() {
        System.out.println("Taking off");
    }
}

public class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Bird is flying");
    }
}
```

The `Bird` class can call `takeOff()` even if it doesn't override it, providing flexibility and backward compatibility when evolving interfaces.

8.2.6 Example: Using `Flyable` and `Swimmable`

```java
public class Main {
    public static void main(String[] args) {
        Flyable bird = new Bird();
        Swimmable fish = new Fish();
        Duck duck = new Duck();

        bird.fly();           // Bird is flying
        fish.swim();          // Fish is swimming
        duck.fly();           // Duck is flying
        duck.swim();          // Duck is swimming
        duck.takeOff();       // Taking off (from Flyable default method)
    }
}
```

8.2.7 Why Use Interfaces?

- **Loose Coupling:** Code can depend on interfaces rather than concrete classes, improving flexibility.

- **Multiple Capabilities:** Classes can combine multiple behaviors by implementing several interfaces.
- **Standardization:** Interfaces define clear contracts for what methods a class must have.
- **Evolution:** Default methods allow interfaces to evolve without breaking existing code.

Full runnable code:

```java
interface Swimmable {
    void swim();
}

interface Flyable {
    void fly();

    default void takeOff() {
        System.out.println("Taking off");
    }
}

class Fish implements Swimmable {
    @Override
    public void swim() {
        System.out.println("Fish is swimming");
    }
}

class Duck implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("Duck is flying");
    }

    @Override
    public void swim() {
        System.out.println("Duck is swimming");
    }
}

class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Bird is flying");
    }
}

public class Main {
    public static void main(String[] args) {
        Flyable bird = new Bird();
        Swimmable fish = new Fish();
        Duck duck = new Duck();

        bird.fly();      // Bird is flying
        fish.swim();     // Fish is swimming
        duck.fly();      // Duck is flying
        duck.swim();     // Duck is swimming
        duck.takeOff();  // Taking off (from default method)
    }
}
```

8.2.8 Summary

- Interfaces declare method signatures that implementing classes must define.
- Use `implements` to adopt an interface's contract.
- Classes can implement multiple interfaces to gain diverse behaviors.
- Default methods provide optional shared implementations inside interfaces.
- Interfaces enable flexible, maintainable designs by promoting loose coupling.

Try defining your own interfaces and implementing them in different classes to see how interfaces empower polymorphism and clean architecture in Java.

8.3 Interface vs Abstract Class

In Java, both **interfaces** and **abstract classes** help define abstract types that other classes can build upon. However, they serve different purposes and have important differences. Understanding when to use each is key to designing clean, maintainable code.

8.3.1 Key Differences Between Interface and Abstract Class

Feature	Interface	Abstract Class
Multiple Inheritance	Supports multiple inheritance (a class can implement many interfaces)	Does *not* support multiple inheritance (a class can extend only one class)
Method Implementation	Before Java 8, methods were abstract only; since Java 8, interfaces can have default and static methods with implementations	Can have both abstract and fully implemented methods
Fields/State	Can only have `public static final` constants (no instance variables)	Can have instance variables (state)
Constructors	No constructors (cannot instantiate)	Can have constructors to initialize state

Feature	Interface	Abstract Class
Purpose	Specifies *capabilities* or *contracts* that classes agree to implement	Provides a base *partial implementation* with shared code
Usage Scenario	Use when unrelated classes share common behavior, or to define capabilities like `Comparable`, `Runnable`	Use when classes share code, fields, or default behavior and belong to the same family
Inheritance Requirement	Classes must implement interface methods or be declared abstract	Subclasses inherit abstract methods to implement and concrete methods to reuse

8.3.2 When to Use an Interface?

- To define **capabilities** or roles that a class can play (e.g., `Flyable`, `Serializable`).
- When you want a class to have **multiple behaviors** by implementing several interfaces.
- For maximum flexibility, especially if your design might require classes to extend different superclasses.
- Since Java 8, interfaces can include **default methods**, enabling evolution without breaking existing code.

Example:

```java
public interface Flyable {
    void fly();
    default void takeOff() {
        System.out.println("Taking off");
    }
}
```

8.3.3 When to Use an Abstract Class?

- When you want to share **common code and state** among related classes.
- When you want to provide **default implementations** and maintain some fields.
- When the relationship is naturally a **"is-a"** hierarchy (e.g., `Vehicle` superclass).
- Abstract classes can define constructors to initialize shared fields.

Example:

```java
public abstract class Vehicle {
    protected int speed;

    public Vehicle(int speed) {
        this.speed = speed;
    }

    public abstract void move();

    public void stop() {
        System.out.println("Vehicle stopped");
    }
}
```

8.3.4 Compatibility and Recent Enhancements

- Before Java 8, interfaces could *only* declare abstract methods.
- Java 8 introduced **default** and **static methods** in interfaces, allowing methods with implementations.
- Despite these enhancements, interfaces **cannot** have instance variables or constructors.
- Abstract classes remain useful when you need to maintain state or constructor logic.

Example:

Full runnable code:

```java
abstract class Vehicle {
    protected int speed;

    public Vehicle(int speed) {
        this.speed = speed;
    }

    public abstract void move();

    public void stop() {
        System.out.println("Vehicle stopped");
    }
}

class Car extends Vehicle {
    public Car(int speed) {
        super(speed);
    }

    @Override
    public void move() {
        System.out.println("Car is moving at " + speed + " km/h");
    }
}

class Bike extends Vehicle {
    public Bike(int speed) {
        super(speed);
    }

    @Override
    public void move() {
        System.out.println("Bike is cruising at " + speed + " km/h");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car(100);
        Vehicle myBike = new Bike(25);

        myCar.move();
        myCar.stop();

        myBike.move();
        myBike.stop();
    }
}
```

8.3.5 Summary

- **Interfaces** define *what* a class can do; they provide a **contract** with no (or limited) implementation.
- **Abstract classes** provide a **partial implementation** with shared code and state.
- Use **interfaces** for multiple inheritance of type and defining capabilities.
- Use **abstract classes** for sharing code and common state in a class hierarchy.
- Java's enhancements blur some lines, but design intent and use case guide the best choice.

Experiment with both interfaces and abstract classes in your projects. Seeing their differences firsthand will help you decide which tool fits your design goals.

8.4 Functional Interfaces and Lambda Basics

Java 8 introduced a powerful new feature called **lambda expressions**, which bring functional programming concepts into Java. At the heart of this feature are **functional interfaces**—interfaces with exactly one abstract method—that enable writing concise, expressive code.

8.4.1 What is a Functional Interface?

A **functional interface** is an interface with **a single abstract method** (SAM). It represents a single behavior or action, making it a perfect target for lambda expressions.

Java provides the `@FunctionalInterface` annotation to explicitly declare an interface as functional, which helps the compiler catch mistakes:

```java
@FunctionalInterface
public interface Greeting {
    void sayHello(String name);
}
```

8.4.2 Lambda Expression Syntax

A lambda expression provides a clear, concise way to implement the single abstract method of a functional interface without writing an entire class or anonymous class.

General syntax:

```java
(parameters) -> expression
```

or

```java
(parameters) -> { statements; }
```

8.4.3 Example: Using Lambda with `Runnable`

Before Java 8, implementing `Runnable` required an anonymous inner class:

```java
Runnable task = new Runnable() {
    @Override
    public void run() {
        System.out.println("Task running...");
    }
};
new Thread(task).start();
```

With lambdas:

```java
Runnable task = () -> System.out.println("Task running...");
new Thread(task).start();
```

This is shorter and easier to read.

8.4.4 Example: Using Lambda with `Comparator`

`Comparator<T>` is a functional interface used to compare objects:

Anonymous class way:

```
Comparator<String> comp = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
};
```

With lambda:

```
Comparator<String> comp = (s1, s2) -> s1.length() - s2.length();
```

8.4.5 Benefits of Lambdas and Functional Interfaces

- **Conciseness:** Less boilerplate code than anonymous classes.
- **Readability:** Focus on *what* the code does rather than *how*.
- **Expressiveness:** Easier to write inline behavior.
- **Functional programming:** Integrates well with APIs that treat behavior as data.

8.4.6 Integration with Java Stream API

Functional interfaces are key to the Java Stream API, which enables powerful data processing pipelines.

Example: Filtering and printing a list of names that start with "J":

Full runnable code:

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Alice", "Jack", "Bob");

        names.stream()
             .filter(name -> name.startsWith("J"))  // Lambda as Predicate functional interface
             .forEach(name -> System.out.println(name));  // Lambda as Consumer functional interface
    }
}
```

Output:

```
John
Jack
```

8.4.7 Common Functional Interfaces in `java.util.function`

Java provides many built-in functional interfaces:

Interface	Abstract Method	Description
`Runnable`	`void run()`	Represents a task to run
`Comparator<T>`	`int compare(T o1, T o2)`	Compares two objects
`Predicate<T>`	`boolean test(T t)`	Tests a condition on input
`Consumer<T>`	`void accept(T t)`	Performs an action on input
`Function<T,R>`	`R apply(T t)`	Transforms input to output
`Supplier<T>`	`T get()`	Supplies a value without input

These interfaces are designed to be used with lambdas and method references.

8.4.8 Summary

- **Functional interfaces** have exactly one abstract method, making them targets for lambda expressions.
- Lambdas provide a **shorter and clearer** syntax to implement single-method interfaces.
- They reduce boilerplate, improve readability, and support a functional programming style.
- Functional interfaces power the **Java Stream API** and other modern Java features.
- Experiment with lambdas implementing familiar interfaces like `Runnable` and `Comparator` to get comfortable.

Give it a try: rewrite your anonymous classes using lambdas and explore how functional interfaces simplify your code!

Chapter 9.

Exception Handling

1. try, catch, finally
2. Throwing and Catching Exceptions
3. Creating Custom Exceptions
4. Checked vs Unchecked Exceptions

9 Exception Handling

9.1 try, catch, finally

In Java, **exceptions** represent unexpected events or errors that occur during program execution, such as trying to divide by zero or accessing a file that doesn't exist. To gracefully handle these situations without crashing, Java provides the **`try-catch-finally`** mechanism, allowing you to catch and manage errors while maintaining program control.

9.1.1 The `try` Block

The `try` block contains code that **might throw an exception**. It's the "risky" section where something could go wrong.

```java
try {
    int result = 10 / 0;  // This will throw ArithmeticException
}
```

If an exception occurs inside the `try` block, Java immediately stops executing that block and looks for an appropriate `catch` block to handle it.

9.1.2 The `catch` Block

The `catch` block **handles exceptions** thrown in the corresponding `try` block. You can specify the type of exception you want to catch.

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

You can also catch multiple exception types by adding multiple `catch` blocks:

```java
try {
    String text = null;
    System.out.println(text.length());  // Throws NullPointerException
} catch (ArithmeticException e) {
    System.out.println("Math error: " + e.getMessage());
} catch (NullPointerException e) {
    System.out.println("Null reference detected!");
}
```

9.1.3 The `finally` Block

The `finally` block contains code that **always runs** after the `try` and `catch` blocks, regardless of whether an exception occurred or not. It's commonly used for **cleaning up resources** like closing files or database connections.

```java
try {
    System.out.println("Trying risky operation");
} catch (Exception e) {
    System.out.println("Exception handled");
} finally {
    System.out.println("This runs no matter what");
}
```

Even if you return from the `try` or `catch` block, the `finally` block will still execute.

9.1.4 Nested try Blocks

You can nest `try-catch` blocks inside one another to handle exceptions at different granularities:

```java
try {
    try {
        int[] numbers = {1, 2};
        System.out.println(numbers[5]);  // Throws ArrayIndexOutOfBoundsException
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Inner catch: Invalid index");
    }
    int a = 10 / 0;  // Throws ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Outer catch: Division by zero");
}
```

Full runnable code:

```java
public class Main {
    public static void main(String[] args) {
        try {
            try {
                int[] numbers = {1, 2};
                System.out.println(numbers[5]);  // Throws ArrayIndexOutOfBoundsException
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Inner catch: Invalid index");
            }
            int a = 10 / 0;  // Throws ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Outer catch: Division by zero");
        }
    }
}
```

9.1.5 Best Practices

- **Avoid empty catch blocks:** They silently ignore errors, making debugging difficult.

```
// Bad: silently ignores exceptions
catch (Exception e) {
    // nothing here
}
```

- **Log exceptions:** Use `e.printStackTrace()` or logging frameworks to record errors for diagnosis.

```
catch (Exception e) {
    e.printStackTrace();
}
```

- **Catch specific exceptions:** Avoid catching generic `Exception` unless necessary to handle unexpected errors.
- **Clean up resources:** Always use `finally` (or try-with-resources in newer Java) to close resources like files or connections.

9.1.6 Complete Example

Full runnable code:

```
import java.io.*;

public class ExceptionDemo {
    public static void main(String[] args) {
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new FileReader("file.txt"));
            String line = reader.readLine();
            System.out.println(line);
        } catch (FileNotFoundException e) {
            System.out.println("File not found!");
        } catch (IOException e) {
            System.out.println("I/O error occurred");
        } finally {
            try {
                if (reader != null) {
                    reader.close();
                    System.out.println("Reader closed");
                }
            } catch (IOException e) {
                System.out.println("Failed to close reader");
            }
        }
    }
}
```

This example opens a file, reads a line, and handles possible exceptions while ensuring the file resource is closed in the `finally` block.

9.1.7 Summary

- The **`try` block** contains code that may throw exceptions.
- The **`catch` blocks** handle specific exceptions thrown in the `try` block.
- The **`finally` block** always executes, used mainly for cleanup.
- Catch multiple exceptions to handle different error types properly.
- Avoid empty catches, log errors, and always clean up resources to write robust code.

Understanding and properly using `try-catch-finally` is essential for writing reliable Java programs that can gracefully recover from errors without crashing.

9.2 Throwing and Catching Exceptions

In Java, you don't just wait for exceptions to happen—you can **explicitly throw** them yourself using the `throw` keyword. This allows you to enforce rules, validate inputs, and signal problems in your code clearly. In this section, we'll explore how to throw exceptions, catch them, and keep your program stable.

9.2.1 Throwing Exceptions with `throw`

The `throw` statement lets you create and throw an exception manually. For example, if a method receives an invalid argument, you can throw an `IllegalArgumentException`:

```
public void setAge(int age) {
    if (age < 0) {
        throw new IllegalArgumentException("Age cannot be negative");
    }
    this.age = age;
}
```

When `throw` is executed, the normal program flow stops, and Java looks for a matching `catch` block.

9.2.2 Checked vs Unchecked Exceptions (Brief Overview)

- **Checked exceptions** (e.g., `IOException`) must be declared in the method signature with `throws` and handled by the caller.

- **Unchecked exceptions** (runtime exceptions like `IllegalArgumentException` or `NullPointerException`) do *not* require declaration or mandatory handling.

In the example above, `IllegalArgumentException` is an unchecked exception, so you don't have to declare it in the method signature.

9.2.3 Catching Thrown Exceptions

To handle exceptions thrown by your code or others, use a `try-catch` block:

```java
try {
    setAge(-5);
} catch (IllegalArgumentException e) {
    System.out.println("Error: " + e.getMessage());
}
```

This catches the exception thrown by `setAge()` and prints an error message, preventing the program from crashing.

9.2.4 Complete Example: Input Validation with Exception Throwing and Catching

Full runnable code:

```java
import java.util.Scanner;

public class AgeValidator {

    public static void setAge(int age) {
        if (age < 0 || age > 150) {
            throw new IllegalArgumentException("Invalid age: " + age);
        }
        System.out.println("Age set to " + age);
    }

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

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

        try {
            setAge(inputAge);
        } catch (IllegalArgumentException e) {
            System.out.println("Caught exception: " + e.getMessage());
            System.out.println("Please enter a valid age between 0 and 150.");
        }

        scanner.close();
    }
}
```

Explanation:

- The `setAge` method validates the age.
- If the input is invalid, it **throws** an `IllegalArgumentException`.
- The `main` method **catches** this exception and displays a friendly message.
- The program continues running without crashing, showing how throwing and catching exceptions maintain stability.

9.2.5 Summary

- Use `throw` to explicitly signal that an error has occurred.
- Unchecked exceptions like `IllegalArgumentException` can be thrown without declaration.
- Use `try-catch` blocks to handle exceptions and prevent program termination.
- Input validation and exception throwing improve program robustness by enforcing rules clearly.

Mastering throwing and catching exceptions is essential to writing reliable Java programs that handle errors gracefully and communicate problems clearly.

9.3 Creating Custom Exceptions

Java provides many built-in exceptions, but sometimes you need to represent **domain-specific errors** that are not covered by the standard ones. In these cases, creating your own **custom exceptions** helps make your code clearer and more expressive.

9.3.1 Why Create Custom Exceptions?

- To communicate **specific error conditions** in your application clearly.
- To separate **business logic errors** from general exceptions.
- To allow precise handling of different error types by catching specific exceptions.
- To improve **code readability** and maintainability by giving meaningful names to errors.

9.3.2 How to Create a Custom Exception

Custom exceptions are classes that extend either:

- `Exception` (checked exceptions, must be declared or caught), or
- `RuntimeException` (unchecked exceptions, optional declaration and handling).

9.3.3 Example: Creating an `InvalidAgeException`

Suppose your program needs to validate age input, and you want a custom exception to represent invalid age values:

```java
// Checked exception (extends Exception)
public class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

Or, as an unchecked exception:

```java
// Unchecked exception (extends RuntimeException)
public class InvalidAgeException extends RuntimeException {
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

9.3.4 Throwing and Catching Custom Exceptions

Here is how you might use `InvalidAgeException` in your code:

```java
public class User {
    private int age;

    // For checked exceptions, declare with throws
    public void setAge(int age) throws InvalidAgeException {
        if (age < 0 || age > 150) {
            throw new InvalidAgeException("Age " + age + " is not valid.");
        }
        this.age = age;
    }
}
```

To use this method and handle exceptions:

```java
public class Main {
    public static void main(String[] args) {
        User user = new User();
        try {
            user.setAge(200);
        } catch (InvalidAgeException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }
}
```

If `InvalidAgeException` extends `RuntimeException`, you can omit the `throws` declaration and `try-catch` block but still choose to catch it if desired.

Full runnable code:

```java
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

class User {
    private int age;

    public void setAge(int age) throws InvalidAgeException {
        if (age < 0 || age > 150) {
            throw new InvalidAgeException("Age " + age + " is not valid.");
        }
        this.age = age;
    }
}

public class Main {
    public static void main(String[] args) {
        User user = new User();
        try {
            user.setAge(200);
        } catch (InvalidAgeException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }
}
```

9.3.5 Benefits of Custom Exceptions

- **Clarity:** Your exceptions convey specific meaning related to your domain.
- **Control:** You can catch and handle different exceptions separately.
- **Extensibility:** Custom exceptions can have additional fields or methods providing extra error details.
- **Debugging:** Meaningful exception types make it easier to trace and fix problems.

9.3.6 Best Practices for Custom Exceptions

- Name your exception clearly with the suffix `Exception` (e.g., `InvalidAgeException`).
- Extend `Exception` if you need the exception to be **checked**, forcing callers to handle it.
- Extend `RuntimeException` for **unchecked exceptions** that indicate programming errors or conditions unlikely to be recovered from.
- Provide constructors that accept error messages and optionally causes (`Throwable`), e.g.:

```java
public class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
    public InvalidAgeException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

9.3.7 Summary

Creating custom exceptions allows you to design error handling that fits your program's needs perfectly. They provide a way to:

- Signal domain-specific problems explicitly.
- Separate error handling logic cleanly.
- Improve code readability by making error types descriptive.

By defining, throwing, and catching custom exceptions, your Java applications become easier to maintain and more robust.

9.4 Checked vs Unchecked Exceptions

In Java, exceptions are broadly categorized into **checked** and **unchecked** exceptions. Understanding the difference is essential for writing robust programs that handle errors properly while keeping the code clean and maintainable.

9.4.1 What Are Checked Exceptions?

Checked exceptions are exceptions that the Java compiler **forces you to handle explicitly**. This means any method that can throw a checked exception must either:

- Catch it using a `try-catch` block, or
- Declare it in its method signature using the `throws` keyword.

If neither is done, your program will **fail to compile**.

9.4.2 Common Checked Exceptions

- `IOException` — for input/output errors (e.g., file not found).
- `SQLException` — when database access errors occur.
- `ClassNotFoundException` — if a class cannot be found during runtime.

9.4.3 Example: Checked Exception Handling

Full runnable code:

```
import java.io.*;

public class CheckedExample {
    public static void readFile(String filename) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(filename));
        System.out.println(reader.readLine());
        reader.close();
    }

    public static void main(String[] args) {
        try {
            readFile("data.txt");
        } catch (IOException e) {
            System.out.println("File error: " + e.getMessage());
        }
    }
}
```

Because `readFile` declares `throws IOException`, callers must handle or propagate it.

9.4.4 What Are Unchecked Exceptions?

Unchecked exceptions are exceptions that **the compiler does not require you to handle or declare**. They usually indicate programming errors or unexpected runtime conditions.

Unchecked exceptions are subclasses of `RuntimeException`.

9.4.5 Common Unchecked Exceptions

- `NullPointerException` — when you try to use a `null` reference.
- `ArithmeticException` — such as division by zero.
- `IndexOutOfBoundsException` — accessing invalid array or list indices.

9.4.6 Example: Unchecked Exception

Full runnable code:

```java
public class UncheckedExample {
    public static void divide(int a, int b) {
        int result = a / b;  // May throw ArithmeticException
        System.out.println("Result: " + result);
    }

    public static void main(String[] args) {
        divide(10, 0);  // Will throw exception at runtime
    }
}
```

The compiler won't force you to catch `ArithmeticException`, but if it occurs, the program crashes unless handled.

9.4.7 Pros and Cons of Checked vs Unchecked Exceptions

Aspect	Checked Exceptions	Unchecked Exceptions
Compiler Enforcement	Must be handled or declared	No compile-time requirement
Typical Use Case	Recoverable conditions (e.g., I/O errors)	Programming errors (e.g., null pointer)
Verbosity	Can lead to more verbose code due to mandatory handling	Cleaner code, but can mask errors
Error Propagation	Explicit propagation via method signatures	Propagates unchecked, may cause runtime failure
Flexibility	Safer in critical systems needing explicit error handling	Faster development, but riskier if misused

9.4.8 When to Use Custom Checked or Unchecked Exceptions

- Use **checked exceptions** when the caller is expected to **recover or handle** the error meaningfully (e.g., file read errors, network failures).
- Use **unchecked exceptions** to indicate **programmer mistakes** or conditions that usually cannot be recovered from (e.g., invalid arguments, null pointers).
- Many Java APIs use unchecked exceptions for illegal arguments (`IllegalArgumentException`) and checked exceptions for external system issues (`IOException`).

9.4.9 Summary

- **Checked exceptions** require explicit handling or declaration and represent recoverable conditions.
- **Unchecked exceptions** do not require explicit handling and usually indicate bugs or programming errors.
- Understanding the distinction helps you design APIs and error handling strategies that balance safety and code clarity.
- Use checked exceptions for expected errors callers can handle, and unchecked exceptions for programming faults.

By grasping checked vs unchecked exceptions, you'll write Java programs that are both reliable and maintainable, handling errors effectively without cluttering your code.

Chapter 10.

Java Collections Framework

1. List, Set, and Map Interfaces
2. ArrayList, LinkedList, HashSet, TreeSet, HashMap
3. Iterators and Enhanced for Loop
4. Sorting and Searching Collections

10 Java Collections Framework

10.1 List, Set, and Map Interfaces

Java's Collections Framework provides powerful, flexible interfaces to manage groups of objects. Among the core interfaces, **`List`**, **`Set`**, and **`Map`** stand out as fundamental building blocks for organizing and accessing data. Understanding their differences helps you choose the right structure based on your needs.

10.1.1 The `List` Interface

A **`List`** is an **ordered collection** that allows **duplicate elements**. It maintains the insertion order, and elements can be accessed by their **index**.

10.1.2 Key characteristics:

- **Order matters:** The order in which you add elements is preserved.
- **Duplicates allowed:** You can add the same object multiple times.
- **Indexed access:** You can get, add, or remove elements using an integer index.

10.1.3 Use cases:

- When you need to maintain a sequence of elements.
- When you want fast random access via indices.
- Examples: a playlist of songs, a list of user names in order.

10.1.4 Conceptual example:

```java
List<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Apple");  // Duplicate allowed
System.out.println(fruits.get(1));  // Outputs: Banana
```

10.1.5 The `Set` Interface

A **`Set`** is a collection that **does not allow duplicates** and, depending on the implementation, may or may not maintain order.

10.1.6 Key characteristics:

- **No duplicates:** Each element is unique.
- **Order varies:** Some `Set` types (like `HashSet`) do not guarantee order, while others (`LinkedHashSet`, `TreeSet`) maintain insertion or sorted order.
- **No indexing:** You cannot access elements by position.

10.1.7 Use cases:

- When uniqueness of elements is important.
- When you want to quickly test if an element exists.
- Examples: storing unique user IDs, tags, or keywords.

10.1.8 Conceptual example:

```java
Set<String> uniqueColors = new HashSet<>();
uniqueColors.add("Red");
uniqueColors.add("Green");
uniqueColors.add("Red");  // Duplicate ignored
System.out.println(uniqueColors.contains("Green"));  // Outputs: true
```

10.1.9 The `Map` Interface

A **`Map`** stores data as **key-value pairs**, where each **key** maps to exactly one **value**. Keys are unique, but values can be duplicated.

10.1.10 Key characteristics:

- **Key uniqueness:** Each key appears once.
- **Value duplication:** Multiple keys can have the same value.

- **No order guarantee:** Depending on the implementation, order may or may not be maintained (`HashMap` vs. `LinkedHashMap`).
- **Efficient key-based lookup:** Fast access to values via keys.

10.1.11 Use cases:

- When associating related data, like a dictionary or phonebook.
- When quick retrieval by key is required.
- Examples: mapping usernames to passwords, product IDs to descriptions.

10.1.12 Conceptual example:

```java
Map<String, String> countryCapitals = new HashMap<>();
countryCapitals.put("Canada", "Ottawa");
countryCapitals.put("France", "Paris");
System.out.println(countryCapitals.get("France"));  // Outputs: Paris
```

10.1.13 Choosing Between List, Set, and Map

Scenario	Recommended Interface	Reason
Need ordered collection with duplicates	`List`	Keeps order and allows duplicates
Need unique elements only	`Set`	Prevents duplicates, useful for uniqueness
Need key-value pairs	`Map`	Stores data as pairs with unique keys for quick lookup

Full runnable code:

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // List example: allows duplicates, maintains order
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Apple");  // Duplicate allowed
        System.out.println("List example:");
        System.out.println("Second fruit: " + fruits.get(1));  // Outputs: Banana
        System.out.println("All fruits: " + fruits);
        System.out.println();

        // Set example: disallows duplicates, no indexing
        Set<String> uniqueColors = new HashSet<>();
        uniqueColors.add("Red");
        uniqueColors.add("Green");
        uniqueColors.add("Red");  // Duplicate ignored
        System.out.println("Set example:");
        System.out.println("Contains 'Green'? " + uniqueColors.contains("Green"));  // Outputs: true
        System.out.println("All unique colors: " + uniqueColors);
        System.out.println();

        // Map example: key-value pairs, keys must be unique
        Map<String, String> countryCapitals = new HashMap<>();
        countryCapitals.put("Canada", "Ottawa");
        countryCapitals.put("France", "Paris");
        System.out.println("Map example:");
        System.out.println("Capital of France: " + countryCapitals.get("France"));  // Outputs: Paris
        System.out.println("All country-capital pairs: " + countryCapitals);
    }
}
```

10.1.14 Summary

- **`List`**: Ordered, indexed, allows duplicates. Use when sequence and position matter.
- **`Set`**: Unordered (or ordered depending on implementation), no duplicates. Use when uniqueness is key.
- **`Map`**: Key-value pairs with unique keys. Use for associative data and fast retrieval.

Each interface represents a different way to organize and access data, enabling you to model real-world scenarios effectively in your Java programs.

10.2 ArrayList, LinkedList, HashSet, TreeSet, HashMap

In the previous section, we explored the core interfaces: `List`, `Set`, and `Map`. Now let's dive deeper into some of the most commonly used **concrete implementations** of these interfaces: `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, and `HashMap`. Understanding their underlying data structures and performance trade-offs helps you choose the right collection for your program.

10.2.1 `ArrayList`

Underlying Structure:

- Backed by a **resizable array**.
- When the internal array fills up, it creates a new, larger array and copies elements over.

Performance:

- **Fast random access** (`get(index)`) — O(1).
- **Adding at end** is usually O(1) amortized.
- **Inserting/removing in the middle** requires shifting elements — O(n).

Use cases:

- When you need **fast indexed access**.
- When insertions/deletions happen mostly at the end.

Example:

```java
import java.util.ArrayList;

ArrayList<String> colors = new ArrayList<>();
colors.add("Red");
colors.add("Blue");
colors.add("Green");

System.out.println(colors.get(1)); // Outputs: Blue

colors.remove("Blue"); // Remove element by value

for (String color : colors) {
    System.out.println(color);
}
```

10.2.2 `LinkedList`

Underlying Structure:

- Implements a **doubly linked list**.
- Each element (node) stores references to previous and next elements.

Performance:

- **Fast insertions and removals** at both ends — O(1).
- Access by index requires traversing the list — O(n).
- Not ideal for random access but great for frequent add/remove operations.

Use cases:

- When you frequently add or remove elements at the beginning or middle.
- When you don't require fast indexed access.

Example:

```java
import java.util.LinkedList;

LinkedList<String> queue = new LinkedList<>();
queue.add("First");
queue.add("Second");
queue.addFirst("Zero");  // Add to the front

System.out.println(queue.get(1)); // Outputs: First

queue.removeLast(); // Remove from end

for (String item : queue) {
    System.out.println(item);
}
```

10.2.3 HashSet

Underlying Structure:

- Uses a **hash table**.
- Hashes elements to compute storage location.

Performance:

- **Add, remove, and contains** operations typically O(1).
- Does **not maintain order**.
- No duplicates allowed.

Use cases:

- When you need a collection of **unique elements** with fast lookup.
- When order is unimportant.

Example:

```java
import java.util.HashSet;

HashSet<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Apple");  // Duplicate ignored

System.out.println(set.contains("Banana")); // true
```

```java
for (String fruit : set) {
    System.out.println(fruit);
}
```

10.2.4 TreeSet

Underlying Structure:

- Uses a **Red-Black tree**, a self-balancing binary search tree.
- Maintains elements in **sorted order** (natural or via a Comparator).

Performance:

- Operations like add, remove, contains run in **O(log n)** time.
- Sorted traversal is easy and efficient.

Use cases:

- When you need **sorted, unique elements**.
- When order matters and you want fast range queries.

Example:

```java
import java.util.TreeSet;

TreeSet<Integer> numbers = new TreeSet<>();
numbers.add(50);
numbers.add(10);
numbers.add(30);

System.out.println(numbers.first());  // Outputs: 10
System.out.println(numbers.last());   // Outputs: 50

for (int num : numbers) {
    System.out.println(num);
}
```

10.2.5 HashMap

Underlying Structure:

- Implements the `Map` interface using a **hash table**.
- Stores key-value pairs.
- Keys are hashed for fast retrieval.

Performance:

- **Put, get, and remove** operations average O(1).
- Does not guarantee order (use `LinkedHashMap` if order matters).

Use cases:

- When you want to associate keys with values.
- When fast lookup by key is important.

Example:

```java
import java.util.HashMap;

HashMap<String, Integer> ages = new HashMap<>();
ages.put("Alice", 30);
ages.put("Bob", 25);

System.out.println(ages.get("Alice"));  // Outputs: 30

ages.remove("Bob");

for (String name : ages.keySet()) {
    System.out.println(name + " is " + ages.get(name) + " years old");
}
```

10.2.6 Summary and Comparison

Collection	Data Structure	Access Speed	Insertion/Deletion	Order Maintained?	Allows Duplicates?
`ArrayList`	Resizable array	Fast (O(1) get)	Slow if in middle (O(n))	Yes (insertion order)	Yes
`LinkedList`	Doubly linked list	Slow (O(n) get)	Fast at ends (O(1))	Yes (insertion order)	Yes
`HashSet`	Hash table	Fast (O(1))	Fast (O(1))	No	No
`TreeSet`	Balanced tree	Moderate (O(log n))	Moderate (O(log n))	Yes (sorted order)	No
`HashMap`	Hash table (key-value)	Fast (O(1))	Fast (O(1))	No	Keys: No; Values: Yes

Full runnable code:

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // ArrayList example
```

```java
ArrayList<String> colors = new ArrayList<>();
colors.add("Red");
colors.add("Blue");
colors.add("Green");
System.out.println("ArrayList get(1): " + colors.get(1)); // Blue
colors.remove("Blue");
System.out.println("ArrayList after removal:");
for (String color : colors) {
    System.out.println(color);
}

System.out.println();

// LinkedList example
LinkedList<String> queue = new LinkedList<>();
queue.add("First");
queue.add("Second");
queue.addFirst("Zero");
System.out.println("LinkedList get(1): " + queue.get(1)); // First
queue.removeLast();
System.out.println("LinkedList after removal:");
for (String item : queue) {
    System.out.println(item);
}

System.out.println();

// HashSet example
HashSet<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Apple");
System.out.println("HashSet contains 'Banana': " + set.contains("Banana"));
System.out.println("HashSet elements:");
for (String fruit : set) {
    System.out.println(fruit);
}

System.out.println();

// TreeSet example
TreeSet<Integer> numbers = new TreeSet<>();
numbers.add(50);
numbers.add(10);
numbers.add(30);
System.out.println("TreeSet first: " + numbers.first());
System.out.println("TreeSet last: " + numbers.last());
System.out.println("TreeSet sorted elements:");
for (int num : numbers) {
    System.out.println(num);
}

System.out.println();

// HashMap example
HashMap<String, Integer> ages = new HashMap<>();
ages.put("Alice", 30);
ages.put("Bob", 25);
```

```
        System.out.println("HashMap get('Alice'): " + ages.get("Alice"));
        ages.remove("Bob");
        System.out.println("HashMap entries:");
        for (String name : ages.keySet()) {
            System.out.println(name + " is " + ages.get(name) + " years old");
        }
    }
}
```

10.2.7 Choosing the Right Implementation

- Use **`ArrayList`** for fast indexed access and mostly append operations.
- Use **`LinkedList`** for frequent insertions/removals at the beginning or middle.
- Use **`HashSet`** to store unique items with fast lookup, no ordering needed.
- Use **`TreeSet`** to keep unique items sorted automatically.
- Use **`HashMap`** when you need to map keys to values with fast access.

Understanding these implementations equips you with the knowledge to select the best collection for your task — balancing speed, order, and data uniqueness according to your program's requirements.

10.3 Iterators and Enhanced for Loop

Traversing collections is a fundamental operation in Java programming. The Collections Framework provides two main ways to iterate through elements: the **Iterator** interface and the **enhanced for loop** (also called the *for-each* loop). Both methods allow you to access each element in a collection, but they differ in flexibility and use cases.

10.3.1 The Iterator Interface

The **Iterator** interface provides a standardized way to safely traverse any collection, such as `List`, `Set`, or `Map` (via key or entry sets).

10.3.2 Key Methods of Iterator:

- `boolean hasNext()` — Returns `true` if there are more elements to iterate over.
- `E next()` — Returns the next element in the iteration.

- `void remove()` — Removes from the underlying collection the last element returned by this iterator. This method is optional but useful for safe removal during iteration.

10.3.3 Advantages of Using Iterator:

- Allows **safe removal** of elements during iteration without causing `ConcurrentModificationExceptio`
- Works uniformly with all collections.
- Provides explicit control over the iteration process.

10.3.4 Example: Iterating over an ArrayList with Iterator

Full runnable code:

```java
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorExample {
    public static void main(String[] args) {
        ArrayList<String> animals = new ArrayList<>();
        animals.add("Cat");
        animals.add("Dog");
        animals.add("Rabbit");

        Iterator<String> it = animals.iterator();
        while (it.hasNext()) {
            String animal = it.next();
            System.out.println(animal);
            if (animal.equals("Dog")) {
                it.remove();  // Remove "Dog" safely during iteration
            }
        }

        System.out.println("After removal: " + animals);
    }
}
```

Output:

```
Cat
Dog
Rabbit
After removal: [Cat, Rabbit]
```

10.3.5 The Enhanced For Loop (For-each Loop)

Introduced in Java 5, the **enhanced for loop** simplifies iteration syntax and improves code readability.

10.3.6 Syntax:

```
for (ElementType element : collection) {
    // Use element
}
```

10.3.7 Advantages:

- Cleaner, more concise syntax.
- Eliminates manual use of `Iterator`.
- Less prone to errors like forgetting to call `next()` or `hasNext()`.

10.3.8 Limitations:

- You **cannot remove** elements safely during iteration.
- Does not provide access to the iterator itself, so no fine-grained control.

10.3.9 Example: Iterating over a Set with enhanced for loop

Full runnable code:

```
import java.util.HashSet;

public class ForEachExample {
    public static void main(String[] args) {
        HashSet<String> fruits = new HashSet<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        for (String fruit : fruits) {
            System.out.println(fruit);
        }
    }
}
```

10.3.10 When to Use Iterator vs. Enhanced For Loop

Feature	Iterator	Enhanced For Loop
Ability to remove elements during iteration	Yes (`remove()` method)	No
Syntax complexity	More verbose	Simple and concise
Use with all collections	Yes	Yes
Suitable for custom iteration	Yes	No
Control over iteration	Fine-grained	Limited

Use **Iterator** when you need to modify the collection during iteration or want explicit control. Use the **enhanced for loop** for straightforward traversal without modification.

10.3.11 Summary

- The **Iterator** interface offers flexible, safe traversal with the ability to remove elements during iteration.
- The **enhanced for loop** provides concise syntax, improving readability for simple iteration.
- Both can be used on any Java collection implementing `Iterable`.
- Choosing between them depends on whether you need modification during iteration and how much control you want over the iteration process.

Understanding these iteration techniques empowers you to write clearer, safer, and more effective Java code when working with collections.

10.4 Sorting and Searching Collections

Sorting and searching are fundamental operations when working with collections in Java. The Java Collections Framework provides built-in tools to perform these efficiently. In this section, we'll explore how to **sort collections** using `Collections.sort()`, the `Comparable` and `Comparator` interfaces for custom sorting, and how to **search** collections using `Collections.binarySearch()`.

10.4.1 Sorting Collections with `Collections.sort()`

The `Collections` class includes the static method `sort()`, which sorts lists in natural order or using a custom comparator.

10.4.2 Sorting Lists of Primitive Wrapper Types

Java's wrapper classes for primitives, like `Integer`, `Double`, and `String`, implement the `Comparable` interface, so their natural order is defined (e.g., numbers sorted ascending, strings lexicographically).

Example:

Full runnable code:

```java
import java.util.ArrayList;
import java.util.Collections;

public class SortExample {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(42);
        numbers.add(15);
        numbers.add(8);
        numbers.add(23);

        System.out.println("Before sorting: " + numbers);

        Collections.sort(numbers);

        System.out.println("After sorting: " + numbers);
    }
}
```

Output:

```
Before sorting: [42, 15, 8, 23]
After sorting: [8, 15, 23, 42]
```

10.4.3 The `Comparable` Interface

To sort a list of custom objects, the objects must define a natural order by implementing the `Comparable` interface. This interface requires implementing the `compareTo()` method.

Example: Sorting `Person` objects by age

Full runnable code:

```java
import java.util.ArrayList;
import java.util.Collections;

class Person implements Comparable<Person> {
    String name;
    int age;

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

    @Override
    public int compareTo(Person other) {
        return this.age - other.age; // ascending order by age
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class ComparableExample {
    public static void main(String[] args) {
        ArrayList<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 35));

        System.out.println("Before sorting: " + people);

        Collections.sort(people);

        System.out.println("After sorting: " + people);
    }
}
```

Output:

```
Before sorting: [Alice (30), Bob (25), Charlie (35)]
After sorting: [Bob (25), Alice (30), Charlie (35)]
```

10.4.4 Using a Comparator for Custom Sorting

If you need to sort objects by different criteria without changing their natural order, use the `Comparator` interface.

Example: Sorting the same `Person` objects by name alphabetically

```java
import java.util.Comparator;

Comparator<Person> nameComparator = new Comparator<Person>() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.name.compareTo(p2.name);
    }
};
Collections.sort(people, nameComparator);
System.out.println("Sorted by name: " + people);
```

With Java 8+, you can simplify using lambda expressions:

```java
Collections.sort(people, (p1, p2) -> p1.name.compareTo(p2.name));
```

10.4.5 Searching Collections with `Collections.binarySearch()`

The `binarySearch()` method performs a fast search on **sorted lists** by repeatedly dividing the search interval in half.

10.4.6 Important:

- The list **must be sorted** according to the same ordering used for the search.
- Otherwise, the result is undefined.

Example: Binary search on sorted integers

```java
int index = Collections.binarySearch(numbers, 23);
System.out.println("Index of 23: " + index);
```

Example: Binary search on custom objects

```java
int pos = Collections.binarySearch(people, new Person("Dummy", 30));
System.out.println("Position of person aged 30: " + pos);
```

Note: The search uses the natural ordering (`Comparable`) or a provided `Comparator`.

Full runnable code:

```java
import java.util.*;

public class Main {
    static class Person {
        String name;
        int age;

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

        // Override toString for readable printout
        @Override
        public String toString() {
            return name + "(" + age + ")";
        }
    }

    public static void main(String[] args) {
        List<Person> people = new ArrayList<>(Arrays.asList(
            new Person("Alice", 25),
            new Person("Bob", 30),
            new Person("Charlie", 20),
            new Person("Diana", 30)
        ));

        // Sort using anonymous Comparator class
        Comparator<Person> nameComparator = new Comparator<Person>() {
            @Override
            public int compare(Person p1, Person p2) {
                return p1.name.compareTo(p2.name);
            }
        };
        Collections.sort(people, nameComparator);
        System.out.println("Sorted by name (anonymous class): " + people);

        // Sort using lambda expression
        Collections.sort(people, (p1, p2) -> p1.name.compareTo(p2.name));
        System.out.println("Sorted by name (lambda): " + people);

        // Binary search requires list sorted by the same comparator
        // Let's search for a Person named "Charlie"
        Person searchPerson = new Person("Charlie", 0);

        // Because binarySearch uses compareTo of Person or comparator,
        // we must pass the same comparator
        int pos = Collections.binarySearch(people, searchPerson, nameComparator);
        System.out.println("Position of 'Charlie': " + pos);

        // If found, print the found person
        if (pos >= 0) {
            System.out.println("Found: " + people.get(pos));
        } else {
            System.out.println("Person not found");
        }
    }
}
```

10.4.7 Why Proper Ordering Matters

- Sorting ensures **predictable order**, allowing efficient searching algorithms like binary search.
- Without sorting, searching requires linear time, which is slower.
- Implementing consistent and correct `compareTo()` or `Comparator` logic is crucial for sorting to work correctly.

- Mixing different sorting criteria can lead to unexpected behavior.

10.4.8 Summary

Concept	Description	Example Use Case
`Collections.sort(List)`	Sorts a list according to natural order or comparator	Sorting integers, strings, or custom objects
`Comparable`	Defines natural ordering for a class	Sorting `Person` objects by age
`Comparator`	Defines alternative sorting logic	Sorting `Person` objects by name
`Collections.binarySearch`	Performs binary search on a sorted list	Finding the index of a number or object

Mastering sorting and searching will make your Java programs more efficient and versatile when handling data collections. Experiment by creating your own classes, implementing these interfaces, and performing searches on sorted lists!

Chapter 11.
Working with Files and I/O

1. Reading and Writing Files (java.io, java.nio)
2. BufferedReader and BufferedWriter
3. Serialization and Deserialization
4. File Paths and Directories

11 Working with Files and I/O

11.1 Reading and Writing Files (java.io, java.nio)

Working with files is a common requirement in many Java applications, whether you want to read data, write logs, or store configurations. Java provides two primary approaches for file input/output (I/O): the classic **`java.io`** package and the newer, more efficient **`java.nio`** (New I/O) package introduced in Java 7.

This section introduces both approaches, explaining their differences, and demonstrates how to read from and write to text files using practical examples.

11.1.1 Stream-Based I/O with `java.io`

The `java.io` package uses **streams** to handle data flow—either bytes (`InputStream`/`OutputStream`) or characters (`Reader`/`Writer`). For text files, character streams are most convenient.

11.1.2 Key Classes for Text File I/O:

- `FileReader` — Reads characters from a file.
- `FileWriter` — Writes characters to a file.
- `BufferedReader` — Wraps `FileReader` for efficient reading line-by-line.
- `BufferedWriter` — Wraps `FileWriter` for efficient writing.

11.1.3 Example: Reading and Writing Text Files Using `java.io`

Full runnable code:

```java
import java.io.*;

public class FileIOExample {
    public static void main(String[] args) {
        String inputFile = "input.txt";
        String outputFile = "output.txt";

        // Writing to a file using FileWriter and BufferedWriter
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {
            writer.write("Hello, Java I/O!");
            writer.newLine();
            writer.write("This is a sample file.");
        } catch (IOException e) {
            System.err.println("Error writing file: " + e.getMessage());
        }

        // Reading from a file using FileReader and BufferedReader
        try (BufferedReader reader = new BufferedReader(new FileReader(outputFile))) {
            String line;
            System.out.println("Reading file content:");
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
        }
    }
}
```

This program writes two lines to `output.txt`, then reads and prints them.

11.1.4 Buffer/Channel-Based I/O with `java.nio`

The `java.nio` package is designed for **non-blocking, buffer-oriented** I/O, offering better performance and scalability. Instead of streams, it uses **buffers** (containers for data) and **channels** (connections to entities like files or sockets).

11.1.5 Key Classes for File I/O in `java.nio`:

- `Paths` — To locate files or directories.
- `Files` — Contains utility methods for file operations (reading, writing).
- `ByteBuffer` and `CharBuffer` — Buffers to hold bytes or characters.
- `FileChannel` — For advanced file I/O with channels and buffers.

11.1.6 Simpler Reading and Writing with `Files`

Java 7 introduced convenient static methods in `Files` for reading and writing small files in one go.

Full runnable code:

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;
import java.util.List;
```

```java
public class NIOExample {
    public static void main(String[] args) {
        Path filePath = Paths.get("nio_output.txt");

        // Writing lines to a file
        List<String> linesToWrite = List.of("Hello from NIO!", "Using Files and Paths.");
        try {
            Files.write(filePath, linesToWrite);
        } catch (IOException e) {
            System.err.println("Error writing file: " + e.getMessage());
        }

        // Reading lines from a file
        try {
            List<String> linesRead = Files.readAllLines(filePath);
            System.out.println("Contents of nio_output.txt:");
            for (String line : linesRead) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
        }
    }
}
```

This approach is very concise and suitable for small to medium-sized text files.

11.1.7 Differences Between `java.io` and `java.nio`

Feature	`java.io` (Stream-based)	`java.nio` (Buffer/Channel-based)
Data Handling	Reads/writes data one byte or character at a time	Reads/writes data in chunks using buffers
Blocking I/O	Blocking calls (waits for operation to complete)	Supports non-blocking I/O (better for scalability)
Convenience	Simple for basic file operations	More powerful, requires more setup
Resource Management	Needs explicit closing (try-with-resources helps)	Uses channels and buffers, also requires closing
Performance	Generally slower for large files	Better performance and scalability

11.1.8 Exception Handling and Resource Management

File operations are prone to errors like missing files, permission issues, or I/O failures. Java enforces checked exceptions for many I/O methods, so you must handle or declare them.

The **try-with-resources** statement (Java 7+) automatically closes streams and readers after use, which is a best practice to avoid resource leaks.

Example of try-with-resources:

```
try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {
    // read file
} catch (IOException e) {
    // handle exception
}
// reader is closed automatically here
```

11.1.9 Summary

- Use **`java.io` streams** (`FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`) for simple, line-oriented text file I/O.
- Use **`java.nio` classes** (`Files`, `Paths`) for efficient, modern file I/O with cleaner APIs.
- Remember to always handle exceptions and close resources safely—try-with-resources is the recommended way.
- For large files or advanced needs, `java.nio`'s buffer and channel model offers better performance.

Mastering both approaches gives you the flexibility to handle a wide range of file I/O scenarios in Java, from small configuration files to large data streams.

11.2 BufferedReader and BufferedWriter

When working with files in Java, efficiency is key—especially when reading or writing large amounts of data. Using buffered streams like `BufferedReader` and `BufferedWriter` helps improve performance by minimizing expensive I/O operations. In this section, we'll explore what buffering means, why buffered streams matter, and how to use these classes properly with practical examples.

11.2.1 What Is Buffering and Why Use It?

File I/O operations interact with external storage devices, which are much slower than accessing memory. Each read or write operation that directly accesses a file can be time-consuming. Buffering introduces an intermediate memory area—called a **buffer**—that temporarily holds data to reduce the number of physical I/O operations.

How buffering improves efficiency:

- When reading, the buffered reader reads a large block of data into memory at once,

then serves it piece by piece on demand.
- When writing, the buffered writer collects data in a buffer and writes it all at once, reducing the number of write calls.

This leads to fewer costly disk accesses and better performance.

11.2.2 `BufferedReader` Reading Text Efficiently

`BufferedReader` wraps a `Reader` (commonly `FileReader`) and provides methods like `readLine()`, which reads text **line by line**—a common requirement for processing text files.

11.2.3 Example: Reading a File Line-by-Line

Full runnable code:

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class BufferedReaderExample {
    public static void main(String[] args) {
        String fileName = "example.txt";

        try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {
            String line;
            System.out.println("File contents:");
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
        }
    }
}
```

Explanation:

- The `try-with-resources` statement automatically closes the `BufferedReader` after use, avoiding resource leaks.
- `readLine()` reads one line at a time, returning `null` when the end of file is reached.
- The example prints each line to the console.

11.2.4 `BufferedWriter` Writing Text Efficiently

`BufferedWriter` wraps a `Writer` (commonly `FileWriter`) and buffers output, improving write efficiency and providing useful methods like `newLine()` to write platform-independent newline characters.

11.2.5 Example: Writing Lines to a File

Full runnable code:

```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class BufferedWriterExample {
    public static void main(String[] args) {
        String outputFile = "output.txt";

        try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {
            writer.write("First line of text");
            writer.newLine();  // writes a newline character
            writer.write("Second line of text");
        } catch (IOException e) {
            System.err.println("Error writing file: " + e.getMessage());
        }
    }
}
```

Explanation:

- The buffered writer collects the text in memory and writes it efficiently to disk.
- `newLine()` ensures the newline is compatible with the operating system.
- The `try-with-resources` block safely closes the writer.

11.2.6 Benefits Over Unbuffered Readers and Writers

Without buffering, every call to `read()` or `write()` might trigger a costly disk access. Buffered streams reduce this overhead by grouping operations:

Feature	Unbuffered (`FileReader`/`FileWriter`)	Buffered (`BufferedReader`/`BufferedWriter`)
Performance	Slower, many disk accesses	Faster, fewer disk accesses
Convenience Methods	No `readLine()` or `newLine()`	Provides useful line-oriented methods

Feature	Unbuffered (`FileReader/FileWriter`)	Buffered (`BufferedReader/BufferedWriter`)
Resource Usage	Less memory used (no buffer)	Uses a buffer in memory to improve speed

11.2.7 Best Practices When Using Buffered Streams

- Always use **try-with-resources** to ensure streams are closed automatically, preventing resource leaks.
- Prefer **buffered streams** when reading/writing large files or multiple lines.
- Use `readLine()` to process text files line-by-line instead of reading single characters.
- When writing, call `flush()` if you need to ensure data is immediately written to disk (usually done automatically when closing).

11.2.8 Combined Example: Copy a File Line-by-Line

Here's a practical example reading from one file and writing to another using buffered streams:

Full runnable code:

```java
import java.io.*;

public class FileCopyBuffered {
    public static void main(String[] args) {
        String sourceFile = "input.txt";
        String destinationFile = "copy.txt";

        try (BufferedReader reader = new BufferedReader(new FileReader(sourceFile));
             BufferedWriter writer = new BufferedWriter(new FileWriter(destinationFile))) {

            String line;
            while ((line = reader.readLine()) != null) {
                writer.write(line);
                writer.newLine();
            }

            System.out.println("File copied successfully.");
        } catch (IOException e) {
            System.err.println("I/O error: " + e.getMessage());
        }
    }
}
```

This program copies all lines from `input.txt` to `copy.txt` efficiently using buffered streams.

11.2.9 Summary

- **BufferedReader** and **BufferedWriter** improve file I/O performance by reducing the number of physical read/write operations.
- They provide convenient methods (`readLine()`, `newLine()`) that simplify text processing.
- Always use **try-with-resources** for safe resource management.
- Buffered streams are preferred for reading/writing large files or working with line-oriented text.

Understanding and using buffered streams effectively will help your Java applications handle file input and output more efficiently and reliably.

11.3 Serialization and Deserialization

In Java, **serialization** is the process of converting an object into a sequence of bytes so it can be saved to a file, sent over a network, or stored in memory. The reverse process, **deserialization**, reconstructs the original object from those bytes. This mechanism allows Java programs to persist and transfer objects easily.

11.3.1 Why Serialization?

Sometimes, you want to store the state of an object permanently or share it between programs, possibly running on different machines. Serialization provides a standard way to:

- Save objects to disk (persistence).
- Send objects through a network (communication).
- Cache objects for later use.
- Create deep copies of objects.

11.3.2 The `Serializable` Interface

In Java, to make an object serializable, its class must implement the marker interface `java.io.Serializable`. This interface does **not** declare any methods; it simply signals to the Java Virtual Machine (JVM) that the class supports serialization.

Example:

```java
import java.io.Serializable;

public class Person implements Serializable {
    private String name;
    private int age;

    // Constructor, getters, setters omitted for brevity
}
```

11.3.3 The `transient` Keyword

Sometimes, a field should **not** be serialized—for example, sensitive information like passwords or data that can be recalculated. The `transient` keyword marks such fields to be skipped during serialization.

```
private transient String password;
```

When deserialized, transient fields are set to their default values (`null` for objects, `0` for numbers, `false` for booleans).

11.3.4 Serializing and Deserializing Objects Example

Let's see a complete example that serializes a `Person` object to a file and then deserializes it.

Full runnable code:

```
import java.io.*;

public class SerializationDemo {

    public static void main(String[] args) {
        String filename = "person.ser";

        // Create a Person object
        Person person = new Person("Alice", 30);

        // Serialize the object to a file
        try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(filename))) {
            out.writeObject(person);
            System.out.println("Person object serialized.");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialize the object from the file
        try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename))) {
            Person deserializedPerson = (Person) in.readObject();
            System.out.println("Deserialized Person: " + deserializedPerson);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

class Person implements Serializable {
    private String name;
    private int age;

    // transient example: this field won't be serialized
    private transient String secretCode = "XYZ123";

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

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + ", secretCode='" + secretCode + "'}";
    }
}
```

11.3.5 Explanation:

- `ObjectOutputStream` and `ObjectInputStream` handle writing and reading serialized objects.
- We serialize the `person` object into `person.ser` file.
- We then deserialize it back into a new `Person` instance.
- Notice that `secretCode`, marked `transient`, is `null` when printed after deserialization because it was not saved.

11.3.6 Use Cases for Serialization

- **Caching:** Store complex objects to disk and reload later without recomputing.
- **Deep Copying:** Serialize an object and deserialize it to create a full copy without sharing references.
- **Remote Communication:** Java RMI (Remote Method Invocation) uses serialization to transfer objects between JVMs.
- **Persistence:** Save user settings or application state between sessions.

11.3.7 Important Considerations

- Classes must maintain **compatibility** by defining a `serialVersionUID` to avoid `InvalidClassException` when class definitions change.
- Be cautious about **security risks**: deserialization of untrusted data can lead to vulnerabilities.
- Serialization works best for **simple objects**; for complex graphs or large data, other mechanisms (like JSON, XML, or custom protocols) may be preferable.

11.3.8 Summary

- Serialization converts Java objects into byte streams for storage or transmission.
- Implement the `Serializable` interface to enable serialization.
- Use `transient` to exclude fields from serialization.
- Java provides `ObjectOutputStream` and `ObjectInputStream` for writing and reading serialized objects.
- Common use cases include caching, deep copying, and network communication.
- Proper handling of serialization versioning and security is important.

Mastering serialization equips you to save and transfer complex Java objects easily, extending the power of your applications beyond runtime memory.

11.4 File Paths and Directories

Managing file paths and directories is a fundamental part of file I/O in Java. Whether you're creating folders, navigating paths, or listing directory contents, Java provides both the traditional `java.io.File` class and the modern, more powerful `java.nio.file.Path` interface with supporting classes to handle these tasks efficiently and safely.

11.4.1 The Legacy `File` Class

The `File` class (in `java.io`) represents file and directory pathnames. You can create, delete, and inspect files or directories using its methods.

11.4.2 Example: Creating and Deleting a Directory

Full runnable code:

```java
import java.io.File;

public class FileExample {
    public static void main(String[] args) {
        File dir = new File("testDir");

        // Create directory if it does not exist
        if (!dir.exists()) {
            boolean created = dir.mkdir();
            System.out.println("Directory created: " + created);
        }

        // Delete directory (only if empty)
        boolean deleted = dir.delete();
        System.out.println("Directory deleted: " + deleted);
    }
}
```

Notes:

- `mkdir()` creates a single directory.
- To create nested directories, use `mkdirs()`.
- `delete()` removes the directory, but it must be empty.

11.4.3 Listing Directory Contents

```java
File dir = new File("someDirectory");
if (dir.isDirectory()) {
    String[] files = dir.list();
    System.out.println("Contents:");
    for (String fileName : files) {
        System.out.println(fileName);
    }
}
```

11.4.4 The Modern `Path` Interface and `Files` Utility

Since Java 7, `java.nio.file.Path` and the utility class `java.nio.file.Files` provide a more flexible way to work with file paths and operations.

11.4.5 Creating Paths

```java
import java.nio.file.Path;
import java.nio.file.Paths;

Path relativePath = Paths.get("testDir");
Path absolutePath = relativePath.toAbsolutePath();

System.out.println("Relative Path: " + relativePath);
System.out.println("Absolute Path: " + absolutePath);
```

11.4.6 Creating Directories

```java
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;

Path dirPath = Paths.get("newDir");
try {
    if (!Files.exists(dirPath)) {
        Files.createDirectory(dirPath);
        System.out.println("Directory created");
    }
} catch (IOException e) {
    System.err.println("Error creating directory: " + e.getMessage());
}
```

11.4.7 Deleting Directories

```java
try {
    Files.deleteIfExists(dirPath);
    System.out.println("Directory deleted if it existed.");
} catch (IOException e) {
    System.err.println("Error deleting directory: " + e.getMessage());
}
```

11.4.8 Listing Directory Contents with Streams

```java
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;

try (DirectoryStream<Path> stream = Files.newDirectoryStream(dirPath)) {
    for (Path entry : stream) {
        System.out.println(entry.getFileName());
    }
} catch (IOException e) {
    System.err.println("Error reading directory: " + e.getMessage());
}
```

11.4.9 Relative vs Absolute Paths and Cross-Platform Best Practices

- **Relative paths** are relative to the current working directory. They are useful for portability within projects.
- **Absolute paths** specify the full path from the root of the filesystem.
- To avoid platform-specific issues with path separators (`/` on Unix/Linux/macOS vs `\` on Windows), always use `Path` and `Paths.get()` rather than hardcoding separators.
- The `Path` interface normalizes paths and handles these differences transparently.

Example:

```
Path path = Paths.get("folder", "subfolder", "file.txt");
System.out.println(path.toString());  // Automatically uses correct separators
```

Full runnable code:

```
import java.io.File;
import java.io.IOException;
import java.nio.file.*;

public class Main {
    public static void main(String[] args) {
        // Using File class to list contents
        File dir = new File("someDirectory");
        if (!dir.exists()) {
            dir.mkdir();  // Create directory if it doesn't exist
        }

        System.out.println("Listing using java.io.File:");
        if (dir.isDirectory()) {
            String[] files = dir.list();
            System.out.println("Contents:");
            for (String fileName : files) {
                System.out.println(fileName);
            }
        }

        // Using Path and Files
        Path relativePath = Paths.get("testDir");
        Path absolutePath = relativePath.toAbsolutePath();

        System.out.println("\nPath information:");
        System.out.println("Relative Path: " + relativePath);
        System.out.println("Absolute Path: " + absolutePath);
        // Create a new directory if it doesn't exist
        try {
            if (!Files.exists(relativePath)) {
                Files.createDirectory(relativePath);
                System.out.println("Directory 'testDir' created.");
            }
        } catch (IOException e) {
            System.err.println("Error creating directory: " + e.getMessage());
        }

        // List contents using DirectoryStream
        System.out.println("\nListing using java.nio.file.DirectoryStream:");
        try (DirectoryStream<Path> stream = Files.newDirectoryStream(relativePath)) {
            for (Path entry : stream) {
                System.out.println(entry.getFileName());
            }
        } catch (IOException e) {
            System.err.println("Error reading directory: " + e.getMessage());
        }

        // Demonstrating relative vs absolute path resolution
        Path smartPath = Paths.get("folder", "subfolder", "file.txt");
        System.out.println("\nCross-platform Path: " + smartPath.toString());

        // Clean up
        try {
            Files.deleteIfExists(relativePath);
            System.out.println("Deleted 'testDir' if it existed.");
        } catch (IOException e) {
            System.err.println("Error deleting directory: " + e.getMessage());
        }
    }
}
```

11.4.10 Summary

- Use `File` for basic file and directory operations, but prefer `Path` and `Files` for modern, flexible file handling.
- Create, delete, and list directories with both APIs.
- Handle exceptions properly when performing file operations.
- Use `Path` and related utilities for cross-platform compatibility and easier path manipulation.
- Prefer `try-with-resources` when working with directory streams to ensure resources are closed.

By mastering both legacy and modern Java file path and directory tools, you'll write code that's both powerful and portable across platforms.

Chapter 12.

Java 8 and Beyond: Modern Features

12 Java 8 and Beyond: Modern Features

12.1 Lambda Expressions in Depth

Java 8 introduced **lambda expressions** — a powerful feature that brings functional programming capabilities to Java. Lambdas enable you to write more concise, readable, and flexible code, especially when working with functional interfaces. In this section, we'll explore lambda syntax, target types, scoping rules, and compare lambdas with anonymous inner classes through practical examples.

12.1.1 What Is a Lambda Expression?

A **lambda expression** is essentially an anonymous function — a block of code you can pass around as an object. Unlike traditional methods, lambdas don't have names and can be used wherever a functional interface is expected.

12.1.2 Lambda Syntax Variations

The basic syntax is:

```
(parameters) -> expression
```

or

```
(parameters) -> { statements; }
```

12.1.3 Examples:

- No parameters:

```
() -> System.out.println("Hello World");
```

- One parameter (parentheses optional):

```
x -> x * 2
```

- Multiple parameters:

```
(x, y) -> x + y
```

- Multiple statements in a block:

```
(x, y) -> {
    int sum = x + y;
```

```java
    return sum;
}
```

12.1.4 Target Types and Functional Interfaces

A lambda expression must match the **target type** — usually a **functional interface** (an interface with exactly one abstract method). The lambda's parameters and return type must be compatible with that method.

12.1.5 Example of a functional interface:

```java
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}
```

You can assign a lambda to a `Calculator`:

```java
Calculator add = (a, b) -> a + b;
System.out.println(add.calculate(5, 3)); // Outputs 8
```

Java 8 provides many built-in functional interfaces like `Runnable`, `Callable`, `Comparator`, `Consumer`, `Supplier`, and `Function`.

12.1.6 Scope Rules in Lambdas

Lambdas capture variables from their enclosing scope similarly to anonymous classes, but with some differences:

- You can use **final** or **effectively final** variables (variables not modified after initialization).
- The `this` keyword inside a lambda refers to the enclosing class instance, not the lambda itself.

Example:

Full runnable code:

```java
public class LambdaScope {
    private int value = 10;

    public void demonstrate() {
        int localVar = 20;
        Runnable r = () -> {
            System.out.println("value = " + value);
            System.out.println("localVar = " + localVar);
            System.out.println("this.value = " + this.value);
        };
        r.run();
    }
}
```

12.1.7 Comparing Lambdas with Anonymous Inner Classes

Before Java 8, anonymous inner classes were used to pass behavior. Lambdas simplify this pattern:

12.1.8 Anonymous Inner Class Example:

```
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running with anonymous class");
    }
};
r1.run();
```

12.1.9 Equivalent Lambda:

```
Runnable r2 = () -> System.out.println("Running with lambda");
r2.run();
```

Benefits of Lambdas:

- **Conciseness:** Less boilerplate code.
- **Readability:** Clearer intent and reduced clutter.
- **Flexibility:** Easier to write inline functions for small operations.

12.1.10 Practical Examples

Example 1: Event Listener

```java
button.addActionListener(e -> System.out.println("Button clicked!"));
```

This replaces verbose anonymous inner classes in GUI programming.

Example 2: Runnable Task

```java
new Thread(() -> {
    for (int i = 0; i < 5; i++) {
        System.out.println("Thread count: " + i);
    }
}).start();
```

Example 3: Functional Interface Implementation

Full runnable code:

```java
@FunctionalInterface
interface StringChecker {
    boolean check(String s);
}

public class LambdaExample {
    public static void main(String[] args) {
        StringChecker isEmpty = s -> s.isEmpty();
        StringChecker startsWithA = s -> s.startsWith("A");

        System.out.println(isEmpty.check(""));      // true
        System.out.println(startsWithA.check("Apple")); // true
    }
}
```

12.1.11 Summary

- Lambda expressions introduce anonymous functions with a compact syntax.
- They require a target functional interface type.
- Lambdas improve code conciseness and clarity, especially for inline behavior.
- Scope rules are similar to anonymous classes but `this` behaves differently.
- Practical use cases include event handling, threading, and functional programming.
- Lambdas are foundational for Java 8's Stream API and functional style.

Mastering lambda expressions opens the door to modern, expressive Java programming, making your code cleaner and more maintainable.

12.2 Stream API Basics and Pipelines

Java 8 introduced the **Stream API**, a powerful tool for processing sequences of elements in a functional style. Streams make it easy to write concise, readable, and expressive code for manipulating collections and other data sources. In this section, we will explore how streams work, the distinction between intermediate and terminal operations, lazy evaluation, and how to build pipelines for efficient data processing.

12.2.1 What Is a Stream?

A **stream** represents a sequence of elements supporting various operations to perform computations in a declarative way. Unlike collections, streams don't store data—they convey data from a source (like a `List`) through a pipeline of operations.

Streams allow operations like filtering, mapping, sorting, and reducing, typically expressed as a chain of method calls.

12.2.2 Intermediate vs Terminal Operations

Intermediate Operations

These operations transform a stream into another stream and are **lazy**, meaning they don't execute until a terminal operation is invoked. Examples include:

- `filter()` — selects elements based on a condition
- `map()` — transforms elements
- `sorted()` — sorts elements
- `distinct()` — removes duplicates

Because they return a stream, intermediate operations can be chained together.

Terminal Operations

These produce a result or side-effect and trigger the execution of the entire pipeline. Examples include:

- `collect()` — gathers elements into a collection or other container
- `forEach()` — performs an action for each element
- `reduce()` — combines elements into a single value
- `count()` — counts elements

Once a terminal operation runs, the stream pipeline processes all intermediate steps.

12.2.3 Lazy Evaluation and Pipelines

Streams use **lazy evaluation**—intermediate operations are not executed until a terminal operation is called. This allows optimization, such as short-circuiting (e.g., stopping processing early when possible).

A **pipeline** is a sequence of stream operations: a data source, followed by zero or more intermediate operations, and terminated by a terminal operation.

12.2.4 Practical Examples

Consider a list of `Person` objects:

```java
import java.util.*;
import java.util.stream.*;

class Person {
    String name;
    int age;

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

    public String getName() { return name; }
    public int getAge() { return age; }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}
```

12.2.5 Example 1: Filtering and Collecting

Find all people older than 18 and collect their names:

```java
List<Person> people = Arrays.asList(
    new Person("Alice", 23),
    new Person("Bob", 17),
    new Person("Charlie", 19)
);

List<String> adults = people.stream()
    .filter(p -> p.getAge() > 18)      // Intermediate operation
    .map(Person::getName)               // Intermediate operation
    .collect(Collectors.toList());     // Terminal operation

System.out.println(adults);  // Output: [Alice, Charlie]
```

12.2.6 Example 2: Sorting and Printing

Sort people by age and print each:

```
people.stream()
    .sorted(Comparator.comparingInt(Person::getAge))  // Intermediate
    .forEach(System.out::println);                     // Terminal
```

Output:

```
Bob (17)
Charlie (19)
Alice (23)
```

12.2.7 Example 3: Combining Operations with Reduction

Calculate the average age:

```
OptionalDouble avgAge = people.stream()
    .mapToInt(Person::getAge)
    .average();

avgAge.ifPresent(avg -> System.out.println("Average age: " + avg));
```

Full runnable code:

```
import java.util.*;
import java.util.stream.*;

public class Main {
    static class Person {
        String name;
        int age;

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

        public String getName() { return name; }
        public int getAge() { return age; }

        @Override
        public String toString() {
            return name + " (" + age + ")";
        }
    }

    public static void main(String[] args) {
        List<Person> people = Arrays.asList(
            new Person("Alice", 23),
            new Person("Bob", 17),
            new Person("Charlie", 19)
        );

        // Example 1: Filtering and collecting names of adults
        List<String> adults = people.stream()
            .filter(p -> p.getAge() > 18)
            .map(Person::getName)
            .collect(Collectors.toList());
        System.out.println("Adults: " + adults);

        // Example 2: Sorting by age and printing each person
        System.out.println("\nSorted by age:");
        people.stream()
            .sorted(Comparator.comparingInt(Person::getAge))
            .forEach(System.out::println);

        // Example 3: Average age
        OptionalDouble avgAge = people.stream()
            .mapToInt(Person::getAge)
            .average();
        avgAge.ifPresent(avg -> System.out.println("\nAverage age: " + avg));
    }
}
```

12.2.8 Why Use Streams?

Streams offer several advantages over traditional loops and collection manipulation:

- **Declarative style:** Focus on *what* you want, not *how* to do it.
- **Conciseness:** Less boilerplate code compared to loops.
- **Composability:** Chain multiple operations fluently.
- **Potential for parallelism:** Easily switch to parallel streams for performance.

12.2.9 Summary

- The Stream API processes data through a pipeline of intermediate and terminal operations.
- Intermediate operations are lazy and return streams, enabling chaining.
- Terminal operations trigger execution and produce results or side effects.
- Streams simplify filtering, mapping, sorting, and collecting data.
- Using streams leads to cleaner, more maintainable code.

Try experimenting by modifying filters, sorting criteria, or collecting into different data structures like sets or maps. Streams provide a modern approach to data processing that's both powerful and intuitive.

12.3 Method References and Optional

Java 8 introduced many modern features to write more concise, readable, and safer code. Two such features are **method references** and the **Optional** class. Method references provide a shorthand way to write lambdas when you just want to call an existing method. Optional helps safely handle potentially null values, reducing the risk of `NullPointerException`.

12.3.1 Method References: A Shortcut for Lambdas

Lambda expressions are great for passing behavior, but sometimes they just call an existing method without adding any extra logic. Method references simplify this by letting you refer to methods directly.

12.3.2 Syntax Variants of Method References

There are **three main types**:

- **Static method references:** `ClassName::staticMethod`
- **Instance method references of a particular object:** `instance::instanceMethod`
- **Instance method references of an arbitrary object of a particular type:** `ClassName::instanceMethod`
- **Constructor references:** `ClassName::new`

12.3.3 Example 1: Static Method Reference

Suppose you want to print elements of a list:

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Using lambda
names.forEach(name -> System.out.println(name));

// Using method reference
names.forEach(System.out::println);
```

Here, `System.out::println` is a reference to the static method `println` of the `PrintStream` instance.

12.3.4 Example 2: Instance Method Reference (Particular Object)

If you have an instance of a class and want to refer to its method:

```java
class Greeter {
    public void greet(String name) {
        System.out.println("Hello, " + name);
    }
}

Greeter greeter = new Greeter();

List<String> names = Arrays.asList("Alice", "Bob");
names.forEach(greeter::greet);
```

Here, `greeter::greet` calls the `greet` method on the specific `greeter` object for each name.

12.3.5 Example 3: Instance Method Reference (Arbitrary Object)

This form is used when you want to call an instance method on each element of the stream or collection:

```java
List<String> words = Arrays.asList("apple", "banana", "cherry");

words.stream()
    .map(String::toUpperCase)  // Calls toUpperCase() on each string
    .forEach(System.out::println);
```

Here, `String::toUpperCase` means for each `String` object, invoke its `toUpperCase` method.

12.3.6 Example 4: Constructor Reference

Constructor references create new objects:

```java
Supplier<List<String>> listSupplier = ArrayList::new;
List<String> newList = listSupplier.get();
```

This is equivalent to:

```java
Supplier<List<String>> listSupplier = () -> new ArrayList<>();
```

12.3.7 Optional: Handling Nullable Values Safely

In Java, `null` can cause frustrating `NullPointerExceptions`. The `Optional<T>` class is a container that **may or may not hold a non-null value**. It encourages explicit handling of the presence or absence of values.

12.3.8 Creating Optionals

- `Optional.of(value)` — creates an Optional containing a non-null value (throws if value is null).
- `Optional.ofNullable(value)` — creates an Optional that may hold null safely.
- `Optional.empty()` — creates an empty Optional (no value).

12.3.9 Example 1: Creating and Using Optional

```
Optional<String> optionalName = Optional.of("Alice");
optionalName.ifPresent(name -> System.out.println("Name is " + name));
// Prints: Name is Alice
```

12.3.10 Example 2: Avoiding Null Checks with `orElse()`

```
Optional<String> emptyOptional = Optional.empty();

String name = emptyOptional.orElse("Default Name");
System.out.println(name);  // Prints: Default Name
```

12.3.11 Example 3: Chaining Optional Operations

```
Optional<String> optional = Optional.ofNullable(getUserInput());

optional
    .map(String::trim)          // Transform the value if present
    .filter(s -> !s.isEmpty()) // Filter out empty strings
    .ifPresent(System.out::println); // Print if a valid non-empty string
```

This avoids manual null and empty checks by chaining methods.

Full runnable code:

```
import java.util.*;
import java.util.function.Supplier;
import java.util.stream.*;
import java.util.Optional;

public class Main {

    // Example 2: Greeter class for method reference
    static class Greeter {
        public void greet(String name) {
            System.out.println("Hello, " + name);
        }
    }

    public static void main(String[] args) {
        // Example 1: Static method reference
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        System.out.println("Example 1: Static method reference");
        names.forEach(System.out::println);

        // Example 2: Instance method reference (particular object)
        System.out.println("\nExample 2: Instance method reference (particular object)");
        Greeter greeter = new Greeter();
        names.forEach(greeter::greet);

        // Example 3: Instance method reference (arbitrary object)
        System.out.println("\nExample 3: Instance method reference (arbitrary object)");
        List<String> words = Arrays.asList("apple", "banana", "cherry");
        words.stream()
             .map(String::toUpperCase)
             .forEach(System.out::println);

        // Example 4: Constructor reference
        System.out.println("\nExample 4: Constructor reference");
        Supplier<List<String>> listSupplier = ArrayList::new;
        List<String> newList = listSupplier.get();
        newList.add("New");
        newList.add("List");
        newList.forEach(System.out::println);

        // Optional examples
        System.out.println("\nOptional example 1: Creating and using Optional");
        Optional<String> optionalName = Optional.of("Alice");
        optionalName.ifPresent(name -> System.out.println("Name is " + name));

        System.out.println("\nOptional example 2: Avoiding null checks with orElse()");
        Optional<String> emptyOptional = Optional.empty();
        String name = emptyOptional.orElse("Default Name");
        System.out.println(name);

        System.out.println("\nOptional example 3: Chaining operations");
        Optional<String> optionalInput = Optional.ofNullable(getUserInput());
        optionalInput
            .map(String::trim)
            .filter(s -> !s.isEmpty())
            .ifPresent(System.out::println);
    }

    // Dummy input simulation
    private static String getUserInput() {
        return "   Hello Optional!   ";
    }
}
```

12.3.12 Benefits of Method References and Optional

- **Method references** improve readability by removing boilerplate lambdas when simply delegating to existing methods.
- **Optional** reduces boilerplate null checks and makes it clear when a value might be missing, improving code safety and clarity.

12.3.13 Summary

- Method references are a concise way to reference existing methods or constructors, making lambda expressions cleaner.
- The Optional class provides a safe container to handle nullable values, offering methods like `of()`, `empty()`, `ifPresent()`, and `orElse()` to deal with presence or absence of data.
- Together, these features help write modern, readable, and null-safe Java code.

Try replacing some of your existing lambdas with method references for cleaner code, and experiment with Optional to handle potential nulls gracefully in your projects.

12.4 Date and Time API (`java.time`)

Before Java 8, handling dates and times in Java was mainly done using the `java.util.Date` and `java.util.Calendar` classes. These classes were often confusing, mutable, and not thread-safe, leading to bugs and complicated code. Java 8 introduced a completely new, modern **Date and Time API** under the package `java.time`, designed to overcome these issues with a clean, fluent, and immutable approach inspired by the popular Joda-Time library.

12.4.1 Core Classes in the `java.time` Package

The API provides a variety of classes tailored for different use cases. Here are the most commonly used ones:

- **`LocalDate`** – Represents a date (year, month, day) without time or timezone, e.g., `2025-06-21`.
- **`LocalTime`** – Represents a time of day (hour, minute, second, nanosecond) without date or timezone.
- **`LocalDateTime`** – Combines date and time without timezone.
- **`ZonedDateTime`** – A date and time with timezone information.

- **`Duration`** – Represents a time-based amount of time (seconds, nanoseconds), useful for measuring intervals.

12.4.2 Creating Dates and Times

You can create instances using factory methods:

```java
LocalDate today = LocalDate.now();  // Current date
LocalDate specificDate = LocalDate.of(2025, 6, 21); // June 21, 2025

LocalTime currentTime = LocalTime.now();
LocalTime specificTime = LocalTime.of(14, 30, 0); // 2:30 PM

LocalDateTime now = LocalDateTime.now();
LocalDateTime birthday = LocalDateTime.of(1990, 12, 15, 10, 0);

ZonedDateTime zonedNow = ZonedDateTime.now(); // Current date/time with timezone
```

12.4.3 Parsing and Formatting Dates and Times

You can easily parse date/time strings and format them back to strings using `DateTimeFormatter`.

```java
// Parsing a date string to LocalDate
String dateStr = "2025-06-21";
LocalDate date = LocalDate.parse(dateStr);

// Formatting LocalDate to a string
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd MMM yyyy");
String formattedDate = date.format(formatter);  // "21 Jun 2025"
System.out.println(formattedDate);
```

12.4.4 Manipulating Dates and Times

The API makes it simple to add or subtract time units:

```java
LocalDate tomorrow = today.plusDays(1);
LocalDate lastWeek = today.minusWeeks(1);

LocalTime oneHourLater = currentTime.plusHours(1);
LocalDateTime nextMonth = now.plusMonths(1);

System.out.println("Tomorrow: " + tomorrow);
System.out.println("One hour later: " + oneHourLater);
System.out.println("Next month: " + nextMonth);
```

12.4.5 Using Duration for Time Intervals

`Duration` measures time between two points or represents fixed time spans:

```java
LocalTime start = LocalTime.of(9, 0);
LocalTime end = LocalTime.of(17, 30);

Duration workDuration = Duration.between(start, end);
System.out.println("Work duration in hours: " + workDuration.toHours());  // 8 hours
```

Full runnable code:

```java
import java.time.*;
import java.time.format.DateTimeFormatter;

public class Main {
    public static void main(String[] args) {
        // Creating Dates and Times
        LocalDate today = LocalDate.now();
        LocalDate specificDate = LocalDate.of(2025, 6, 21);

        LocalTime currentTime = LocalTime.now();
        LocalTime specificTime = LocalTime.of(14, 30, 0);

        LocalDateTime now = LocalDateTime.now();
        LocalDateTime birthday = LocalDateTime.of(1990, 12, 15, 10, 0);

        ZonedDateTime zonedNow = ZonedDateTime.now();

        System.out.println("Today: " + today);
        System.out.println("Specific date: " + specificDate);
        System.out.println("Current time: " + currentTime);
        System.out.println("Specific time: " + specificTime);
        System.out.println("Now: " + now);
        System.out.println("Birthday: " + birthday);
        System.out.println("Zoned now: " + zonedNow);

        System.out.println("\n--- Parsing and Formatting ---");
        String dateStr = "2025-06-21";
        LocalDate parsedDate = LocalDate.parse(dateStr);
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd MMM yyyy");
        String formattedDate = parsedDate.format(formatter);
        System.out.println("Formatted: " + formattedDate);

        System.out.println("\n--- Manipulating Dates and Times ---");
        LocalDate tomorrow = today.plusDays(1);
        LocalDate lastWeek = today.minusWeeks(1);
        LocalTime oneHourLater = currentTime.plusHours(1);
        LocalDateTime nextMonth = now.plusMonths(1);
        System.out.println("Tomorrow: " + tomorrow);
        System.out.println("Last week: " + lastWeek);
        System.out.println("One hour later: " + oneHourLater);
        System.out.println("Next month: " + nextMonth);

        System.out.println("\n--- Duration Example ---");
        LocalTime start = LocalTime.of(9, 0);
        LocalTime end = LocalTime.of(17, 30);
        Duration workDuration = Duration.between(start, end);
        System.out.println("Work duration in hours: " + workDuration.toHours());
        System.out.println("In minutes: " + workDuration.toMinutes());
    }
}
```

12.4.6 Thread Safety and Immutability

All classes in the `java.time` API are **immutable**—once created, their values cannot change. Instead, any manipulation returns a new instance. This immutability makes them inherently **thread-safe**, meaning you can share date/time objects across threads without synchronization concerns.

This design dramatically improves reliability and simplifies multithreaded applications compared to the old `Date` and `Calendar` classes.

12.4.7 Summary

- The new Date and Time API (`java.time`) replaces the older, error-prone `Date` and `Calendar`.
- Key classes include `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, and `Duration`.
- You can easily create, parse, format, and manipulate dates and times with fluent, readable methods.
- The API is immutable and thread-safe, promoting safer and cleaner code.

Explore these classes and try formatting, parsing, and calculating dates and times in your programs to master Java's modern approach to date/time handling.

Chapter 13.

Inner and Nested Classes

1. Member and Static Nested Classes
2. Anonymous Inner Classes
3. Local Inner Classes and Use Cases

13 Inner and Nested Classes

13.1 Member and Static Nested Classes

Java allows you to define classes inside other classes, which are called **nested classes**. Nested classes help logically group classes that are only used in one place, increase encapsulation, and reduce namespace clutter. There are two main types of nested classes: **member inner classes** and **static nested classes**. Though both are nested inside another class, they behave quite differently.

13.1.1 Member Inner Classes

A **member inner class** is a non-static class defined inside another class. It is associated with an instance of the enclosing class and can access the enclosing class's members—including private fields and methods—directly.

13.1.2 Syntax and Characteristics:

```java
public class OuterClass {
    class InnerClass {
        void display() {
            System.out.println("Inside member inner class");
        }
    }
}
```

- Declared without the `static` keyword.
- Each instance of the inner class is tied to an instance of the outer class.
- Can access all fields and methods of the outer class, even private ones.
- To create an instance of a member inner class, you need an instance of the outer class.

13.1.3 Example: Member Inner Class Usage

Full runnable code:

```java
public class Car {
    private String model = "Tesla";

    // Member inner class
    class Engine {
        void showModel() {
            // Accessing outer class's private field
            System.out.println("Engine in car model: " + model);
        }
    }

    public static void main(String[] args) {
        Car car = new Car();                // Create outer class instance
        Car.Engine engine = car.new Engine();  // Create inner class instance
        engine.showModel();                 // Output: Engine in car model: Tesla
    }
}
```

13.1.4 Static Nested Classes

A **static nested class** is a nested class declared with the `static` keyword. Unlike member inner classes, it does **not** associate with an instance of the outer class. Instead, it behaves more like a regular top-level class but scoped inside the outer class.

13.1.5 Syntax and Characteristics:

```
public class OuterClass {
    static class StaticNestedClass {
        void display() {
            System.out.println("Inside static nested class");
        }
    }
}
```

- Declared with the `static` keyword.
- Cannot directly access non-static members of the outer class.
- Does not require an instance of the outer class to be instantiated.
- Commonly used to group classes that are only relevant to the outer class but do not need access to its instance data.

13.1.6 Example: Static Nested Class Usage

Full runnable code:

```
public class Computer {
    private static String brand = "Dell";

    // Static nested class
    static class USBPort {
        void showBrand() {
            // Can access static fields of outer class
            System.out.println("USB Port for brand: " + brand);
        }
    }

    public static void main(String[] args) {
        Computer.USBPort usb = new Computer.USBPort();  // No outer instance needed
        usb.showBrand();  // Output: USB Port for brand: Dell
    }
}
```

13.1.7 Key Differences and Use Cases

Feature	Member Inner Class	Static Nested Class
Declared with `static`?	No	Yes
Associated with outer instance?	Yes	No
Access to outer class members	Can access all (including non-static)	Can access only static members
Instantiation requirement	Needs an instance of outer class	No outer instance needed
Typical use case	When inner class needs outer instance data	When grouping helper classes logically

13.1.8 Why Use Static Nested Classes?

- **Namespace Management:** They help group related classes inside another class without cluttering the package namespace.
- **Performance:** Since static nested classes don't carry a reference to the outer instance, they are more memory-efficient.
- **Clearer Design:** Use static nested classes when the nested class doesn't need access to the outer class instance but logically belongs to the outer class.

13.1.9 Summary

- **Member inner classes** are tied to an instance of the outer class, can access all its members, and require an outer class instance to be created.
- **Static nested classes** behave more like regular classes, cannot access instance members

directly, and don't require an outer instance.
- Use member inner classes when the inner class needs to interact closely with its outer class's instance.
- Use static nested classes to group classes logically, reduce namespace pollution, and avoid unnecessary references to outer instances.

Try creating your own classes with both member and static nested classes to see how they help organize and encapsulate your code effectively!

13.2 Anonymous Inner Classes

Java provides a convenient way to declare and instantiate a class all at once without explicitly naming the class: this is called an **anonymous inner class**. Anonymous inner classes are useful when you need a one-time-use implementation of an interface or subclass, typically for event handling, callbacks, or simple method overrides.

13.2.1 What Are Anonymous Inner Classes?

An anonymous inner class is an **unnamed class** defined and instantiated in a single expression. You usually use it to provide an implementation of an interface or an abstract class on the spot, without cluttering your code with a separate class declaration.

13.2.2 Syntax Overview

```
InterfaceOrClassType instance = new InterfaceOrClassType() {
    // Class body: override methods or add new code
};
```

- The new class is created inline.
- It can override methods from the interface or superclass.
- No class name is provided.
- Typically assigned to a variable or passed as an argument.

13.2.3 Typical Use Cases

Before Java 8 introduced lambdas, anonymous inner classes were commonly used for:

- Event listeners in GUI programming.
- Runnable tasks for threads.
- Implementing interfaces with a single method.
- Quick one-off subclasses.

13.2.4 Example 1: Using Anonymous Inner Class for Runnable

Suppose you want to create a simple thread task:

Full runnable code:

```
public class Demo {
    public static void main(String[] args) {
        Runnable task = new Runnable() {
            @Override
            public void run() {
                System.out.println("Task running in a thread");
            }
        };

        Thread thread = new Thread(task);
        thread.start();
    }
}
```

Here, `new Runnable() { ... }` defines and creates an unnamed class implementing `Runnable` and overrides `run()` right away.

13.2.5 Example 2: Anonymous Inner Class vs Lambda Expression

Since Java 8, lambda expressions provide a more concise way to do the same thing when targeting functional interfaces (interfaces with a single abstract method).

```
// Using anonymous inner class
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running with anonymous inner class");
    }
};

// Using lambda expression (Java 8+)
Runnable r2 = () -> System.out.println("Running with lambda");
```

Both create a runnable task, but lambdas are shorter and often clearer.

13.2.6 More Practical Example: Action Listener in GUI

Before lambdas, adding an action listener to a button looked like this:

```java
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});
```

With Java 8 lambdas, it can be simplified:

```java
button.addActionListener(e -> System.out.println("Button clicked!"));
```

13.2.7 Key Points About Anonymous Inner Classes

- They can extend a class or implement an interface, but not both.
- Can access final or effectively final variables from the enclosing scope.
- Useful for quick, inline implementations without extra class files.
- They create a separate class file under the hood (with names like `OuterClass$1.class`).

13.2.8 When to Use Anonymous Inner Classes vs Lambdas

- Use **anonymous inner classes** when implementing interfaces with **multiple methods** or when you need to override multiple methods or add fields.
- Use **lambdas** for concise, functional-style code targeting interfaces with **a single abstract method**.
- Lambdas enhance readability and reduce boilerplate code compared to anonymous inner classes.

13.2.9 Summary

Anonymous inner classes provide a flexible way to create unnamed, one-off implementations for interfaces or classes directly within your code. They were widely used before lambdas arrived in Java 8, especially for GUI event handling and thread tasks. Although lambdas offer a cleaner syntax for many cases, understanding anonymous inner classes is crucial for maintaining and understanding legacy Java code.

Try rewriting some of your anonymous inner classes using lambda expressions where possible to see how they compare in conciseness and clarity!

13.3 Local Inner Classes and Use Cases

In Java, **local inner classes** are classes defined **inside a method**, rather than at the class level. These classes are local to the method where they are declared, meaning their scope and visibility are confined to that method. Local inner classes provide a neat way to encapsulate helper functionality that is only relevant within a single method, improving code organization and readability.

13.3.1 What Are Local Inner Classes?

A local inner class is declared within a method (or a block) and cannot be accessed from outside that method. They behave like any other inner class but have some specific rules due to their local scope:

- They can access **final** or **effectively final** variables from the enclosing method.
- Their instances can only be created within the method.
- They are hidden from the outside world, providing encapsulation.

13.3.2 Why Use Local Inner Classes?

Local inner classes are useful when:

- You want to keep helper logic tightly scoped to a method without polluting the wider class namespace.
- The class is short-lived and only meaningful during the execution of a specific method.
- You want to implement an interface or extend a class in a method, but the implementation doesn't deserve a full class at the outer level.

13.3.3 Example: Local Inner Class Inside a Method

Here's a simple example demonstrating a local inner class defined inside a method:

Full runnable code:

```
public class Calculator {

    public void calculateSquares(int[] numbers) {
        // Local inner class defined within the method
        class SquarePrinter {
            void printSquares() {
                for (int num : numbers) {
                    System.out.println(num + " squared is " + (num * num));
                }
            }
        }

        // Create an instance and use it
        SquarePrinter printer = new SquarePrinter();
        printer.printSquares();
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int[] values = {2, 3, 4};
        calc.calculateSquares(values);
    }
}
```

In this example:

- `SquarePrinter` is a local inner class inside `calculateSquares`.
- It has access to the method parameter `numbers` (which is effectively final — not modified).
- The class is only usable inside `calculateSquares`, encapsulating the functionality neatly.

13.3.4 Accessing Variables

Local inner classes can access:

- **Final variables** explicitly declared as `final`.
- **Effectively final variables** — variables that are not modified after initialization.

If you try to modify a variable accessed by the local class after declaration, the compiler will issue an error, ensuring safety and predictability.

13.3.5 Use Cases for Local Inner Classes

Some common scenarios where local inner classes shine:

- **Event handling:** Defining short event listener classes inside a method.
- **Helper operations:** Encapsulating small algorithmic steps without exposing them at the class level.
- **Building complex methods:** Breaking a method into logical parts without creating multiple outer classes.

13.3.6 Summary

Local inner classes provide a powerful way to encapsulate class definitions within methods. They promote cleaner code by limiting class visibility and keeping related logic bundled together. Understanding their scope rules and usage helps you write better-organized Java code that is easier to maintain.