Chapter 1: Java and Object-Oriented Programming (OOP)

Welcome to the **Java and OOP textbook** template created with **Marp**.
This template is designed for printed-style documents using **A4 portrait pages** with automatic pagination.

1.1 Introduction

Java is one of the most popular programming languages in the world.
It powers web servers, Android apps, robots, and even banking systems.
But the most important thing that makes Java powerful is its **Object-Oriented Programming (OOP)** structure.

OOP is a way of designing programs that models the real world — objects, actions, and relationships — in code.
In this chapter, you'll review the key OOP ideas using Java, compare them with other languages, and learn how to handle errors safely.

1.2 What is Object-Oriented Programming?

OOP stands for **Object-Oriented Programming**.
It helps you think of software as a world of **objects** that have data (attributes) and behavior (methods).

💡 Core Concepts of OOP

Concept	Description	Example
Class	A blueprint for creating objects	`class Student { ... }`
Object	A real instance made from a class	`Student s = new Student();`
Encapsulation	Hiding internal data and protecting access	Using `private` and `getter/setter`
Inheritance	Reusing and extending another class	`class Dog extends Animal`

1.3 Java Recap: Structure of a Program

Every Java program must have a class and a `main()` method.
That's where the program begins to run.

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

Compile and Run

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

1.4 Classes and Objects

A **class** defines what an object knows (its variables) and what it can do (its methods).
An **object** is created from that class — like building a real car from a blueprint.

```java
class Student {
    String name;
    int score;

    void showInfo() {
        System.out.println(name + " got " + score + " points");
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.name = "Alice";
        s.score = 95;
        s.showInfo();
    }
}
```

Output

```
Alice got 95 points
```

Here:

- `Student` is the class.
- `s` is the object (instance) of that class.

1.5 Encapsulation — Protecting Data

Encapsulation means **hiding the details** of how data is stored or changed.
Instead of letting users directly change a variable, we control access using **getters** and **setters**.

```java
class Student {
    private int score;

    public void setScore(int s) {
        if (s >= 0 && s <= 100)
            score = s;
        else
            System.out.println("Invalid score!");
    }

    public int getScore() {
        return score;
    }
}
```

Why it matters:

- Prevents invalid data
- Keeps internal design flexible
- Encourages clean, safe code

1.6 Inheritance — Reusing and Extending Classes

Inheritance allows one class to **reuse** another's code.
You define a *parent class* (or *superclass*), and a *child class* (or *subclass*) that extends it.

```java
class Animal {
    void speak() {
        System.out.println("Some sound...");
    }
}

class Dog extends Animal {
    void speak() {
        System.out.println("Woof!");
    }
}
```

Example Run

```java
Dog d = new Dog();
d.speak();
```

Output:

```
Woof!
```

Analogy:

Just as a dog **is a type of** animal, a subclass inherits the traits of its parent class.

1.7 Polymorphism — One Interface, Many Behaviors

“Polymorphism” literally means “many forms.”
It lets a variable of a parent type refer to any child type — and run the right method automatically.

```java
Animal a = new Dog();
a.speak();
```

Even though `a` is declared as an `Animal`, it speaks like a `Dog`.
This makes code flexible — new types can be added without changing the main program.

1.8 Interface — Defining a Contract

An **interface** describes a list of methods that a class *must implement*, but it doesn't say how they work.

Think of it like a "promise" that a class makes.

```java
interface Playable {
    void play();
}

class Dog implements Playable {
    public void play() {
        System.out.println("Dog plays fetch!");
    }
}
```

Any class that implements `Playable` must have a `play()` method.

✅ Interfaces are great for connecting unrelated classes through shared behavior.

1.9 Package — Organizing Your Code

As your program grows, it's better to group related classes in **packages** (folders).
This helps manage large projects and avoid naming conflicts.

Example folder:

```
src/
 └── animals/
      ├── Animal.java
      └── Dog.java
```

Inside `Dog.java`:

```java
package animals;

public class Dog extends Animal { ... }
```

Then in another file:

```java
import animals.Dog;
```

🗂️ Packages = organized, readable projects.

1.10 Exception Handling — Catching Errors Gracefully

Sometimes things go wrong — a file is missing, a number divides by zero, etc.
Instead of letting your program crash, Java lets you **catch and handle** exceptions.

Example

```java
public class Demo {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        } finally {
            System.out.println("Program finished.");
        }
    }
}
```

Output

```
Cannot divide by zero!
Program finished.
```

Two Kinds of Exceptions

Type	Example	Must Handle?

1.11 Full Program Example

Below is a complete Java program combining the OOP ideas you've learned.

```java
interface Playable {
    void play();
}
class Animal {
    public void speak() { System.out.println("..."); }
}

class Dog extends Animal implements Playable {
    private String name;
    Dog(String name) { this.name = name; }

    @Override
    public void speak() { System.out.println(name + " says Woof!"); }
    public void play() { System.out.println(name + " plays fetch!"); }
}
```

```java
public class Main {
    public static void main(String[] args) {
        try {
            Dog d = new Dog("Bobby");
            d.speak();
            d.play();
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Output

```
Bobby says Woof!
Bobby plays fetch!
```

1.12 Mini Project: School Registration System 🏫

Challenge:
Design a simple OOP model for a school's course registration system.

Requirements:

- Classes: `Student`, `Course`, `Registration`
- Use **encapsulation** for student data
- Use **inheritance** if needed (e.g., `OnlineCourse` extends `Course`)
- Throw an **exception** if the course is full

Goal:
Apply all OOP principles in a meaningful context.

1.13 OOP Summary

Concept	What It Does
Class & Object	Define and create things in code
Encapsulation	Protect internal data
Inheritance	Reuse and extend behavior
Polymorphism	One method, many outcomes
Interface	Define shared contracts
Exception Handling	Catch and fix runtime errors safely

1.14 Review Questions

1. What is the difference between a **class** and an **object**?
2. Why is **encapsulation** important?
3. What Java keyword allows one class to inherit another?
4. How does **polymorphism** make code flexible?
5. When would you use an **interface** instead of inheritance?
6. What happens when a program divides by zero in Java?

1.15 Reflection

> “OOP teaches us not just to write code,
> but to think in **systems** — objects that work together.”

By mastering OOP in Java, you’re building skills that apply to almost every modern programming language — Python, C#, Swift, and more.

✅ Key Takeaways

- Java uses **OOP** as its foundation.
- Understanding OOP makes large, complex programs easier to design.
- Always think:

  > *What objects exist in my problem?*
  > *What do they do?*
  > *How do they interact?*

🧩 Practice Exercise (Optional)

Write a small program for a **Library System**:

- Create `Book`, `Member`, and `Library` classes.
- Use Encapsulation to protect book details.
- Use Inheritance for `EBook` vs `PrintedBook`.
- Use an Interface `Borrowable`.
- Use Exception Handling when a member borrows more than 3 books.

📖 Further Reading

- *Head First Java* by Kathy Sierra & Bert Bates
- *Effective Java* by Joshua Bloch
- *Java Programming for Beginners* (Oracle Academy)