📚 Java Learning Journey

Chapter 9 - Objects and Classes

✍️ **Prepared by:** Ahmed Elsifi

🧠 Procedural Programming vs. Object-Oriented Programming (OOP)

Procedural Programming	**Object-Oriented Programming (OOP)**
Focuses on procedures/functions.	Focuses on objects and their interactions.
Data and functions are separate.	Data and behavior are bundled into objects.
Top-down approach.	Bottom-up or modular approach.
Harder to maintain for large systems.	Better for GUI, large-scale, reusable software.

Example:
To create a GUI (like buttons, text fields), procedural code would be messy. OOP allows you to create objects like `Button`, `TextField`, each with their own properties and behaviors.

🧱 What Are Objects, Properties, Methods, and Constructors?

Object

- An instance of a class.
- Represents a real-world entity (e.g., a circle, a student, a button).
- Has:
  - **State** (properties/data fields)
  - **Behavior** (methods)

Properties (Data Fields)

- Attributes that define the object's state.
- Example: `radius` of a `Circle`.

Methods

- Functions that define the object's behavior.
- Example: `getArea()`, `setRadius()`.

Constructor

- A special method used to **initialize objects**.
- Has the **same name as the class**.
- No return type (not even `void`).
- Can be overloaded.

🧪 Java OOP Syntax: Creating Objects and Constructors

Defining a Class

```java
public class Circle {
    // Property (data field)
    private double radius;

    // Constructor
    public Circle() {
        radius = 1.0;
    }

    // Overloaded constructor
    public Circle(double newRadius) {
        radius = newRadius;
    }

    // Method
    public double getArea() {
        return radius * radius * Math.PI;
    }
}
```

Creating an Object

```java
// Using the default constructor
Circle circle1 = new Circle();

// Using the overloaded constructor
Circle circle2 = new Circle(5.0);
```

Accessing Members

```java
// Call a method
double area = circle2.getArea();

// Access a field (if public, but usually private)
double r = circle2.radius; // Not recommended without getter
```

❓ Why Use Constructors?

- To **initialize an object's state** when it's created.
- To ensure the object is in a **valid initial state**.
- To provide flexibility through **overloading**.

Why Overload Constructors?

- To allow objects to be initialized in **different ways**.
- Example:

```java
public class Circle {
    private double radius;

    public Circle() {
        radius = 1.0; // Default
    }

    public Circle(double radius) {
        this.radius = radius; // Custom
    }
}
```

- Now users can create circles with default radius or a custom one.

🧠 Key Concepts to Remember

- A **class** is a blueprint; an **object** is an instance of that class.
- Use `new` to create objects.
- Use the **dot operator** (`.`) to access methods and fields.
- **Constructors** initialize objects. They can be overloaded.
- **Data encapsulation**: Make fields `private`, provide `public` getters/setters.
- `this` refers to the current object.
- Objects are passed by **reference**, primitives by **value**.
- Use `static` for class-level (shared) variables/methods.

✅ Example to Tie It All Together

```java
// Define class
public class Student {
    private String name;
    private int id;

    // Constructor
    public Student(String name, int id) {
        this.name = name;
        this.id = id;
    }

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

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

// Create object
Student s1 = new Student("Alice", 101);

// Use object
System.out.println(s1.getName()); // Output: Alice
```

🧠 Final Note

OOP helps you model the real world in code.
Remember:

- **Classes** define types.
- **Objects** are instances.
- **Constructors** initialize.
- **Methods** define behavior.
- **Encapsulation** protects data.