Chapter 11: Object-Oriented Programming Concepts

Introduction

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code that manipulates the data. It has become the dominant programming approach for designing robust, modular, and reusable software systems.

This chapter explores the core principles, components, and benefits of OOP. It provides a deep dive into the four pillars of OOP, various terminologies, and advanced concepts, using examples from languages like Java, C++, and Python.

11.1 Basics of Object-Oriented Programming

OOP models the real world more closely than procedural programming. Instead of writing functions and procedures that operate on data, OOP organizes both data and behavior into objects.

Key Definitions:

- **Object**: An instance of a class. It contains state (attributes) and behavior (methods).
- **Class**: A blueprint for creating objects. It defines the structure and behaviors that the objects created from it will have.

Example (Java):

```
class Car {
    String color;
    void drive() {
        System.out.println("Car is driving");
    }
}
Car myCar = new Car();  // Object creation
```

11.2 Four Pillars of OOP

11.2.1 Encapsulation

Encapsulation is the process of bundling data and methods that operate on the data within a single unit (class), and restricting access to some of the object's components.

- **Access Modifiers**: `private`, `public`, `protected`
- **Getters and Setters**: Used to access private variables

Benefits:

- Data hiding
- Improved code maintainability

Example (Java):

```
class Employee {
    private int salary;

    public void setSalary(int s) {
        salary = s;
    }

    public int getSalary() {
        return salary;
    }
}
```

11.2.2 Inheritance

Inheritance allows a class (subclass/derived class) to inherit fields and methods from another class (superclass/base class).

- Promotes code reusability
- Supports hierarchical classification

Example (Java):

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

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

11.2.3 Polymorphism

Polymorphism allows objects to be treated as instances of their parent class rather than their actual class.

- **Compile-time Polymorphism** (Method Overloading)

- **Runtime Polymorphism** (Method Overriding)

Example (Method Overriding in Java):

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

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

11.2.4 Abstraction

Abstraction means hiding internal implementation and showing only the essential features.

- Achieved using abstract classes or interfaces
- Helps reduce complexity

Example (Java):

```
abstract class Shape {
    abstract void draw();
}

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

11.3 OOP Terminology and Concepts

11.3.1 Constructor

A special method that is automatically called when an object is created.

```
class Person {
    Person() {
        System.out.println("Constructor called");
    }
}
```

11.3.2 `this` Keyword

Refers to the current object instance.

11.3.3 Static Keyword

Belongs to the class rather than instance.

11.3.4 Final Keyword

- `final` variable → constant
- `final` method → cannot be overridden
- `final` class → cannot be inherited

11.4 Object Class and Method Overriding (Java-Specific)

Object Class

All classes in Java inherit from the `Object` class by default. Key methods:

- `toString()`
- `equals()`
- `hashCode()`

Method Overriding

Allows a subclass to provide a specific implementation of a method already defined in its superclass.

11.5 Aggregation vs Composition

- **Aggregation**: "Has-a" relationship where both entities can exist independently.
- **Composition**: "Has-a" relationship where the child cannot exist without the parent.

Example (Aggregation):

```
class Engine {}

class Car {
    Engine engine;  // Car has-a Engine
}
```

11.6 Interface vs Abstract Class

Feature	Interface	Abstract Class
Methods	Only abstract (Java 7); default & static (Java 8+)	Can have both abstract and concrete methods
Fields	Public static final only	Can have any type of field
Multiple Inheritance	Yes	No

11.7 SOLID Principles in OOP

The SOLID principles help design scalable and maintainable OOP systems.

1. **S** – Single Responsibility Principle
2. **O** – Open/Closed Principle
3. **L** – Liskov Substitution Principle
4. **I** – Interface Segregation Principle
5. **D** – Dependency Inversion Principle

11.8 Advanced OOP Concepts

11.8.1 Object Cloning

Creating a copy of an object using `clone()` method in Java.

11.8.2 Inner Classes

Classes defined within another class.

11.8.3 Anonymous Classes

A class without a name used for instantiating objects with certain "extras".

11.8.4 Lambda Expressions (Java 8+)

Used to implement functional programming within OOP.

11.9 Object-Oriented Design Patterns

OOP enables use of reusable patterns in software engineering:

- **Creational Patterns**: Singleton, Factory, Builder
- **Structural Patterns**: Adapter, Composite, Proxy
- **Behavioral Patterns**: Observer, Strategy, Command

Summary

Object-Oriented Programming is a foundational concept in modern software development. Its key principles—encapsulation, inheritance, polymorphism, and abstraction—enable modular, reusable, and maintainable code. Through a deep understanding of classes, objects, access control, and design patterns, developers can create robust software systems tailored to real-world problems.