CS 104 - Object-Oriented Programming (OOP)

Lecture 5: Encapsulation, Classes, Objects, Constructors

Department of Computer Science, UET Peshawar

---

1. Objects and Classes

Definition
An **object** in Java is an instance of a class. It represents real-world entities that have **states** (attributes) and **behaviors** (actions). For example, a car object may have the state (color, model) and behaviors (start, stop).
A **class** is a blueprint for creating objects. It defines the state and behavior that the objects created from it will have.

Why Use Classes and Objects?

- **Reusability**: Classes allow the reuse of code across multiple instances (objects).
- **Modularity**: Each object can be managed independently, leading to organized and maintainable code.
- **Memory Efficiency**: Objects are allocated memory dynamically at runtime, so Java manages memory efficiently.

Example:

```
class Car {
    private String model;
    private String color;

    // Constructor
    public Car(String model, String color) {
        this.model = model;
        this.color = color;
    }

    // Getter for model
    public String getModel() {
        return model;
    }

    // Setter for model with validation
    public void setModel(String model) {
        if(model.equals("Porche") || model.equals("Corolla")) {
            this.model = model;
        } else {
            this.model = "Unknown Model";
        }
    }

    // Getter for color
    public String getColor() {
        return color;
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Porche", "Red");
        System.out.println("Model: " + myCar.getModel());
        System.out.println("Color: " + myCar.getColor());
    }
}
```

Packages

Java **packages** allow you to group related classes and organize code. This structure helps prevent **namespace conflicts** and improves code maintenance. When dealing with large projects, packages organize different functionality into separate units.

Example of creating multiple classes in a package:

```
package vehicles;  // Declare package

public class Car {
    private String model;
    public Car(String model) {
        this.model = model;
    }
}
```

---

2. Encapsulation

Encapsulation is a principle of wrapping the data (variables) and methods (functions) together in a single unit (class) and restricting direct access to some of the object's components. This is done by **declaring class variables as private** and accessing them via **getter and setter methods**.

Why Make Variables Private?

- **Data Protection**: By keeping variables private, you ensure the data inside the object is safe and not accessible from outside the class.
- **Controlled Access**: Using setter methods, you can control how the data is modified (e.g., by adding validation).

Best Practice:

- Keep class variables **private** and provide **getter** and **setter** methods for controlled access.
- Use **validation** in setter methods to avoid invalid state.

---

3. The `this` Keyword

In Java, the `this` keyword refers to the **current object instance**. It is mainly used to differentiate between instance variables and method parameters when they have the same name.

Example:

```
public class Car {
    private String model;

    public Car(String model) {
        this.model = model;  // 'this' refers to the current object's model 
variable
    }
}
```

Why Use `this`?

- To resolve **variable shadowing** (when a method parameter has the same name as an instance variable).
- To call **other constructors** in the same class from a constructor.

---

4. Setters and Getters

Setters are methods used to assign values to private variables, and **getters** are used to retrieve those values. They provide a way to implement **validation logic** before updating the value of a variable.

Advantages:

- **Validation**: Setters allow checking if the data being passed is valid, ensuring an object always has a valid state.
- **Encapsulation**: Even though the variables are private, setters and getters expose a controlled interface for external classes.

---

5. Constructors

A **constructor** is a special method used to initialize objects. A constructor has the same name as the class and **does not have a return type**.
Constructors allow setting initial values when an object is created and ensure that the object starts with valid data.

Constructor Overloading
Java allows multiple constructors with different parameter lists (constructor overloading). This lets you create objects in different ways based on the provided parameters.

Example:

```
public class BankAccount {
    private String accountNumber;
    private double balance;

    // Constructor with no parameters
    public BankAccount() {
        this("00000", 0.00); // Calling another constructor
    }

    // Constructor with parameters
    public BankAccount(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }
}
```

Why Use Constructors Instead of Setter Methods?

- Ensures that the object is created with a valid state from the start.
- Reduces the need for setter methods since values can be initialized at the time of object creation.

6. Small Program: Bank Application

This program demonstrates encapsulation, constructors, and setter/getter methods.

```
class BankAccount {
    private String accountNumber;
    private String customerName;
    private double balance;
    private String email;
    private String phoneNumber;

    // Constructor
    public BankAccount(String accountNumber, String customerName, double
balance, String email, String phoneNumber) {
        this.accountNumber = accountNumber;
        this.customerName = customerName;
        this.balance = balance;
        this.email = email;
        this.phoneNumber = phoneNumber;
    }

    // Deposit method
    public void deposit(double amount) {
        balance += amount;
        System.out.println("Deposited: " + amount + " New balance: " +
balance);
    }

    // Withdraw method
    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
            System.out.println("Withdrew: " + amount + " New balance: " +
balance);
        } else {
            System.out.println("Insufficient funds.");
        }
    }

    // Getter for balance
    public double getBalance() {
        return balance;
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("12345", "John Doe", 1000.0,
"john@example.com", "1234567890");
        account.deposit(200);
        account.withdraw(150);
    }
}
```

7. Constructor Overloading and VIP Customer Example

```
class VipCustomer {
    private String name;
    private double creditLimit;
    private String email;

    // Constructor with default values
    public VipCustomer() {
        this("Default Name", 5000.0, "default@example.com");
    }

    // Constructor with two parameters
    public VipCustomer(String name, double creditLimit) {
        this(name, creditLimit, "unknown@example.com");
    }

    // Constructor with three parameters
    public VipCustomer(String name, double creditLimit, String email) {
        this.name = name;
        this.creditLimit = creditLimit;
        this.email = email;
    }

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

Explanation:
In the above example, constructor overloading allows creating a `VipCustomer` object with varying levels of detail. You can either provide just the name and credit limit or use the default values if none are provided.