Zoho Schools for Graduate Studies

Notes

DAY - 6(24-07-2025)

Session -1

Encapsulation

Definition:

Encapsulation refers to the **wrapping of data (variables) and methods (functions) into a single unit**, typically a **class**, and restricting direct access to some of the object's components. By making class variables private, we restrict direct access from outside the class. Instead, we provide public **getter and setter methods**—also known as **accessors and mutators**—to read and update these variables. This enables the developer to add validation logic, logging, or restrictions within those methods before the data is accessed or modified.

What we can do:

1. Protect Data (Data Hiding).
2. Control Access with Getters/Setters.
3. Improve Maintainability and Flexibility.
4. Achieve Better Security and Integrity.

Example Code:

```
public class Student {
    private int rollNo;
    private String name;

    public int getRollNo() {
        return rollNo;
    }
    public void setRollNo(int rollNo) {
        if (rollNo > 0) {
            this.rollNo = rollNo;
        }
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        if (!name.isEmpty()) {
            this.name = name;
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.setRollNo(102);
        s.setName("Hariram");
        System.out.println("Student Details:");
        System.out.println("Roll No: " + s.getRollNo());
        System.out.println("Name: " + s.getName());
    }
}
```

Accessor Methods (Getters):

Definition:

- Used to read the value of a private variable.
- Do not modify the data.
- Typically start with get.

.

Example:

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

Mutator Methods (Setters):

Definition:

- Used to **modify** (change) the value of a private variable.
- Typically start with set.

.

Example:

```
`       public void setName(String name) {
           this.name = name;
```

Example Code(Accessor Methods & Mutator Methods):

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

    // Accessor (getter)
    public String getName() {
        return name;
    }

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

    // Accessor
    public int getAge() {
        return age;
    }

    // Mutator
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }
```

Session -2

Packages:

Definition:

A package in Java is a group of related classes, interfaces, and sub-packages. It serves as a namespace that helps avoid class name conflicts and allows better control over code organization and access.

Use Cases:

- **Avoid Name Conflicts**:
  Two classes with the same name can exist in different packages.
- **Access Protection**:
  Classes, methods, and fields can be given package-level access.
- **Modular Development**:
  Code is organized logically, making it easier to manage and maintain.
- **Reusability**:
  Packages promote code reusability. You can import and reuse classes in other programs.

Types of Packages:

1. Built-in Packages (Predefined):

These come with the Java API.

Examples:

- **java.lang** → Core classes like String, Math, System
- **java.util** → Utility classes like ArrayList, Date, Scanner
- **java.io** → Input-output classes like File, BufferedReader
- **java.sql** → Database classes like Connection, Statement

Example Code:

1) java.util package:

```
import java.util.Scanner;
public class InbuiltExample {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = sc.nextLine();
        System.out.println("Hello, " + name + "!");
    }
}
```

2) java.lang package

```
public class Example2 {
    public static void main(String[] args) {
        double result = Math.sqrt(25);
        System.out.println("Square root of 25 is: " + result);
    }
}
```

3) java.io package

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Example3 {
    public static void main(String[] args) throws
FileNotFoundException {
        File file = new File("test.txt");
        Scanner sc = new Scanner(file);
        while (sc.hasNextLine()) {
            System.out.println(sc.nextLine());
        }
    }
}
```

4) java.sql package

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;


public class Example5 {

    public static void main(String[] args) {

        try {

            Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/test",
"user", "pass");

            System.out.println("Connected to database");

        } catch (SQLException e) {

            System.out.println("Connection failed");

        }

    }

}
```

2. User-defined Packages

These are packages you create to group your own classes.

Example Code:

Student.java:

```
package mycodes;
public class Student {
    public void display() {
        System.out.println("This is a user-defined package
example.");
    }
}
```

Main.java:

```
import mycodes.Student;

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.display();
    }
}
```

Use cases of packages:

1. Avoiding Name Conflicts:

Packages help prevent class name collisions when different developers or libraries use the same class names.

2. Access Control:

Packages help restrict access to classes and members using access modifiers like default and protected.

3. Modular Code Development:

Packages support modular programming by separating different features or components into different folders/packages.

4. Code Reusability:

Once a package is created, its classes can be imported and reused in multiple applications.

5. Simplifies Maintenance:

When code is organized into packages, it's easier to maintain or update a specific feature without affecting others.