

Lab 3: Object Oriented Programing - JAVA

CLO: 1

Objectives

- Introduction to Object-Oriented Programming (OOP)
- Core OOP Concepts
 - Classes and Objects
 - Encapsulation
 - Inheritance
 - Polymorphism
 - Abstraction
- Relationships among classes
 - Association
 - Aggregation
 - Composition

Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects. Objects are instances of classes, which encapsulate both data (attributes) and behavior (methods) relevant to that object.

Key Benefits of OOP:

- **Reusability:** Classes and objects can be reused in other programs.
- **Modularity:** Larger programs are broken into smaller, manageable components (objects).
- **Maintainability:** Code is easier to update, modify, and understand.
- **Scalability:** OOP makes it easier to manage large software projects.

Methods

Methods are used to divide complicated programs into manageable pieces.

1. **Predefined methods:** methods that are already written and provided by Java.
2. **User-defined methods:** methods created by you.
 - a. **Value-returning methods:** methods that have a return data type, these methods return a value of a specific data type using the **return** statement.
 - b. **Void methods:** methods that do not have a return data type, these methods do not use a **return** statement to return a value.

Value-returning methods

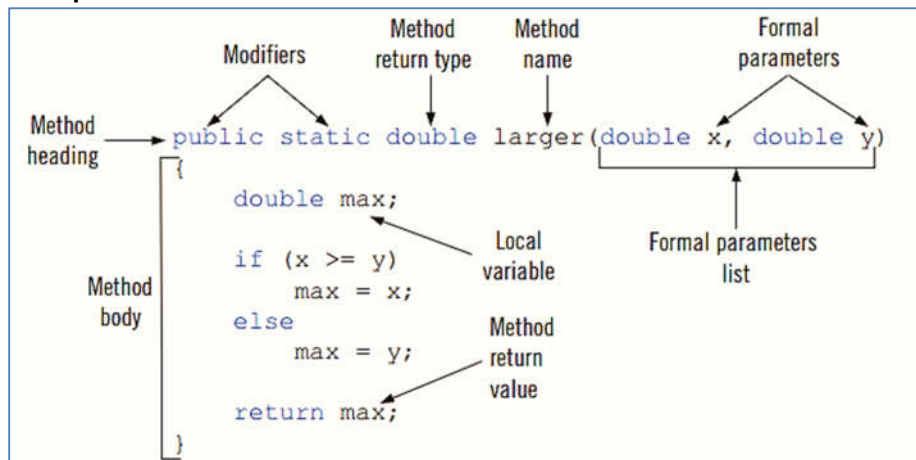
SYNTAX: VALUE-RETURNING METHOD

The syntax of a value-returning method is:

```
modifier(s) returnType methodName(formal parameter list)
{
    statements
}
```

Modifiers are: [public](#), [private](#), [protected](#), [static](#), [abstract](#), and [final](#).

Example # 1



Void methods

The definition of a void method with parameters has the following syntax:

```
modifier(s) void methodName(formal parameter list)
{
    statements
}
```

Example # 2

```
public static void printStars(int blanks, int starsInLine)
{
    int count = 1;

    //print the number of blanks before the stars in a line
    for (count <= blanks; count++)
        System.out.print(" ");

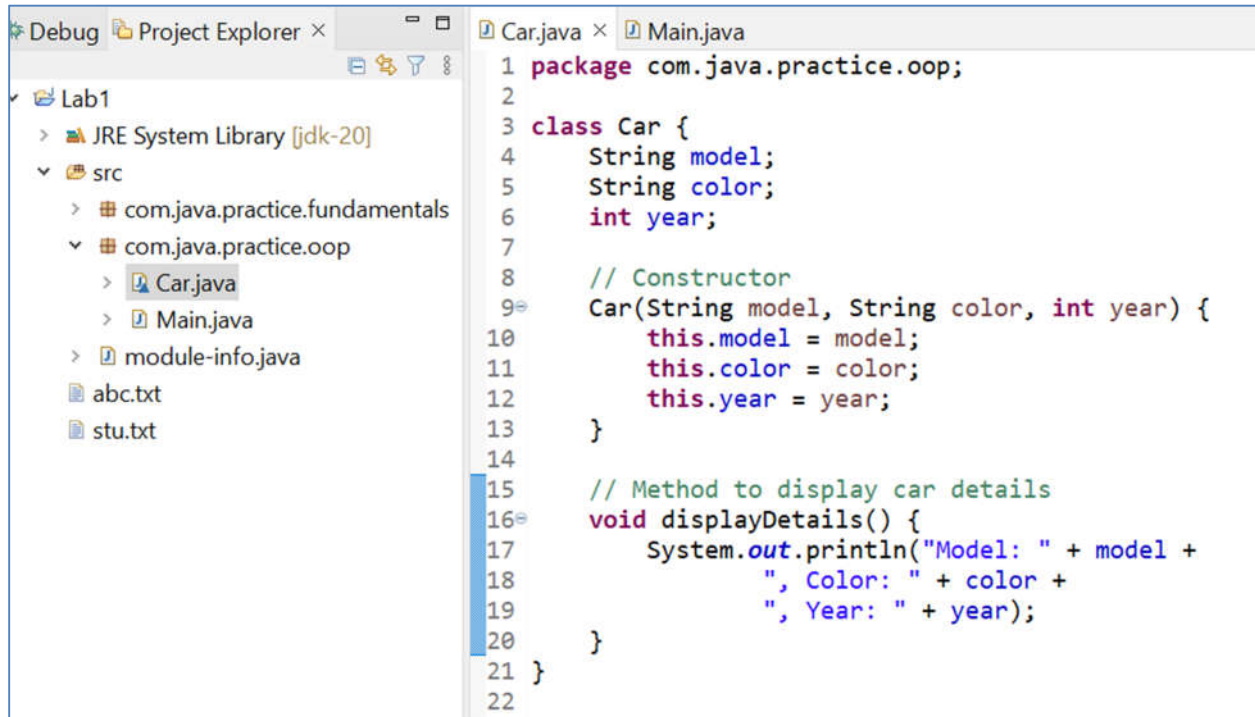
    //print the number of stars with a blank between stars
    for (count = 1; count <= starsInLine; count++)
        System.out.print(" *");

    System.out.println();
} //end printStars
```

Core OOP Concepts

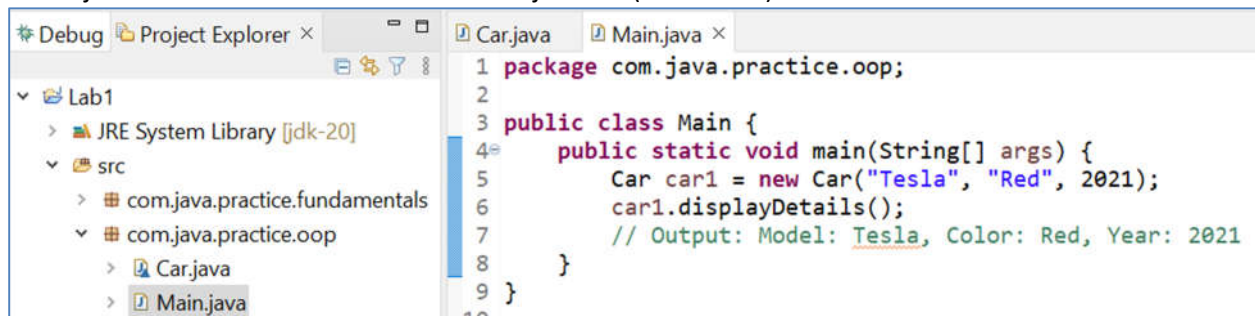
- A **class** is a blueprint for creating objects.
- An **object** is an instance of a class, containing attributes and methods that define its behavior and state.

Example # 3



```
1 package com.java.practice.oop;
2
3 class Car {
4     String model;
5     String color;
6     int year;
7
8     // Constructor
9     Car(String model, String color, int year) {
10         this.model = model;
11         this.color = color;
12         this.year = year;
13     }
14
15     // Method to display car details
16     void displayDetails() {
17         System.out.println("Model: " + model +
18                             ", Color: " + color +
19                             ", Year: " + year);
20     }
21 }
22
```

Car object created in main function of Main.java file (class Main)



```
1 package com.java.practice.oop;
2
3 public class Main {
4     public static void main(String[] args) {
5         Car car1 = new Car("Tesla", "Red", 2021);
6         car1.displayDetails();
7         // Output: Model: Tesla, Color: Red, Year: 2021
8     }
9 }
10
```

Encapsulation

Encapsulation is the concept of wrapping data (variables) and methods (functions) within a single unit (class) and restricting access to the inner workings of that class. This is achieved by using **access modifiers** like **private**, **protected**, and **public**.

Example # 4

The image consists of two screenshots of an IDE, likely Eclipse, showing Java code. The top screenshot displays the `BankAccount.java` file. The code defines a `BankAccount` class with two private attributes: `accountNumber` (String) and `balance` (double). It includes a constructor `BankAccount(String accountNumber, double initialBalance)` that initializes these attributes. There are also two methods: `getBalance()` which returns the current balance, and `deposit(double amount)` which adds the amount to the balance, provided it is greater than zero. The bottom screenshot displays the `Main.java` file. It shows a `Main` class with a `main` method. In the `main` method, a `Car` object is created and its details are displayed. Then, a `BankAccount` object is created with an initial balance of 1000, 500 is deposited, and the current balance is printed, resulting in 1500.

```
1 package com.java.practice.oop;
2
3 class BankAccount {
4     private String accountNumber;
5     private double balance;
6
7     public BankAccount(String accountNumber, double initialBalance) {
8         this.accountNumber = accountNumber;
9         this.balance = initialBalance;
10    }
11
12    // Getter method
13    public double getBalance() {
14        return balance;
15    }
16
17    // Setter method with business rule
18    public void deposit(double amount) {
19        if (amount > 0) {
20            balance += amount;
21        }
22    }
23
24    public void withdraw(double amount) {
25        if (amount > 0 && amount <= balance) {
26            balance -= amount;
27        }
28    }
29 }
```

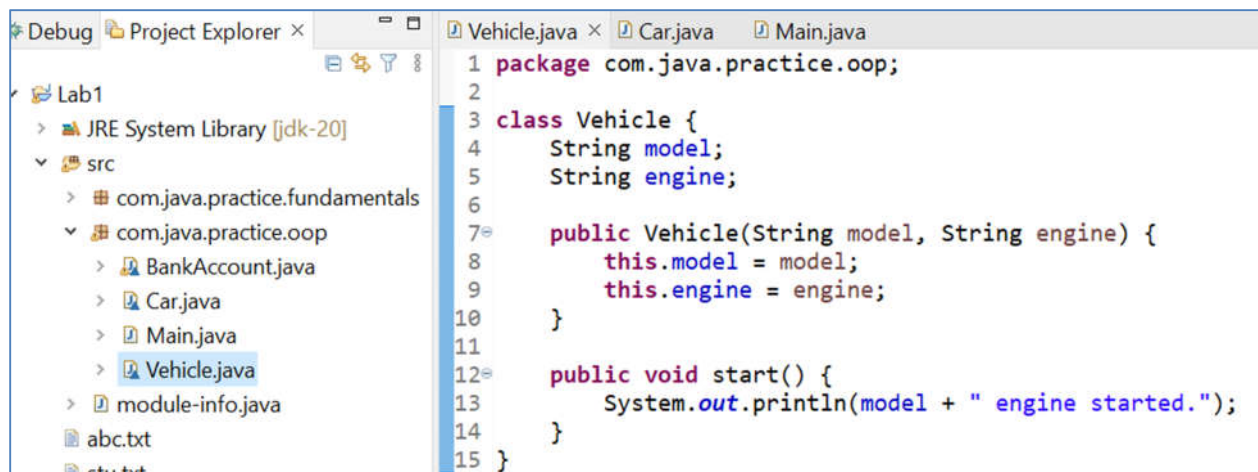
```
1 package com.java.practice.oop;
2
3 public class Main {
4     public static void main(String[] args) {
5         Car car1 = new Car("Tesla", "Red", 2021);
6         car1.displayDetails();
7         // Output: Model: Tesla, Color: Red, Year: 2021
8
9
10        BankAccount account = new BankAccount("HBL0001110120", 1000);
11        account.deposit(500);
12        System.out.println("Current balance: " + account.getBalance());
13        // Output: 1500
14    }
15 }
```

Inheritance

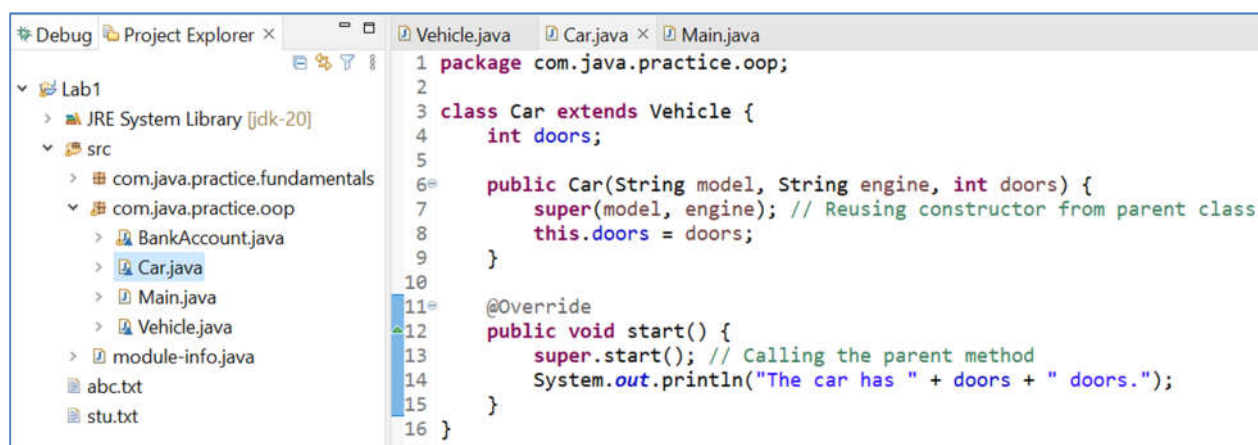
Inheritance allows one class (subclass) to inherit properties and methods from another class (superclass). This promotes code reusability and method overriding.

Example # 5

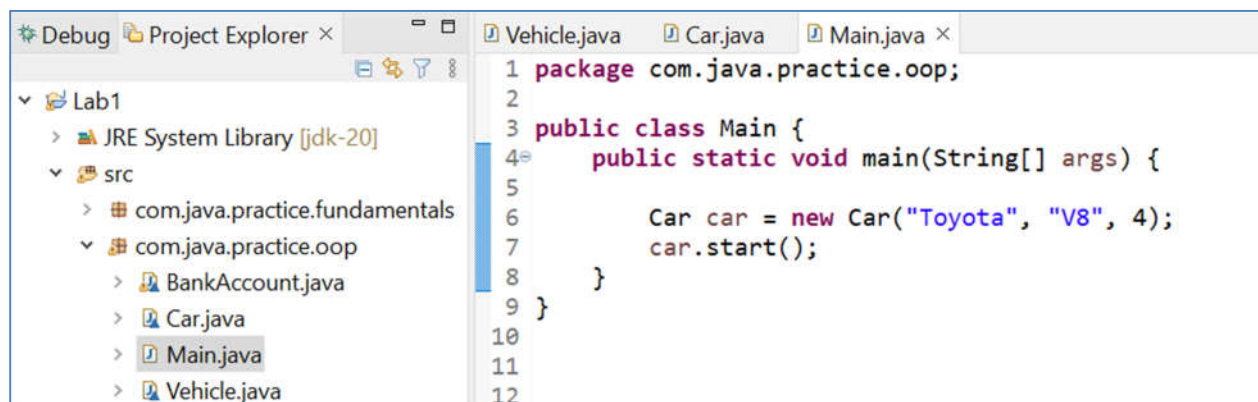
Inheritance is demonstrated here by the `Car` class inheriting from `Vehicle`. The `Car` class extends the functionality by adding doors and overriding the `start()` method.



```
1 package com.java.practice.oop;
2
3 class Vehicle {
4     String model;
5     String engine;
6
7     public Vehicle(String model, String engine) {
8         this.model = model;
9         this.engine = engine;
10    }
11
12    public void start() {
13        System.out.println(model + " engine started.");
14    }
15 }
```



```
1 package com.java.practice.oop;
2
3 class Car extends Vehicle {
4     int doors;
5
6     public Car(String model, String engine, int doors) {
7         super(model, engine); // Reusing constructor from parent class
8         this.doors = doors;
9     }
10
11    @Override
12    public void start() {
13        super.start(); // Calling the parent method
14        System.out.println("The car has " + doors + " doors.");
15    }
16 }
```



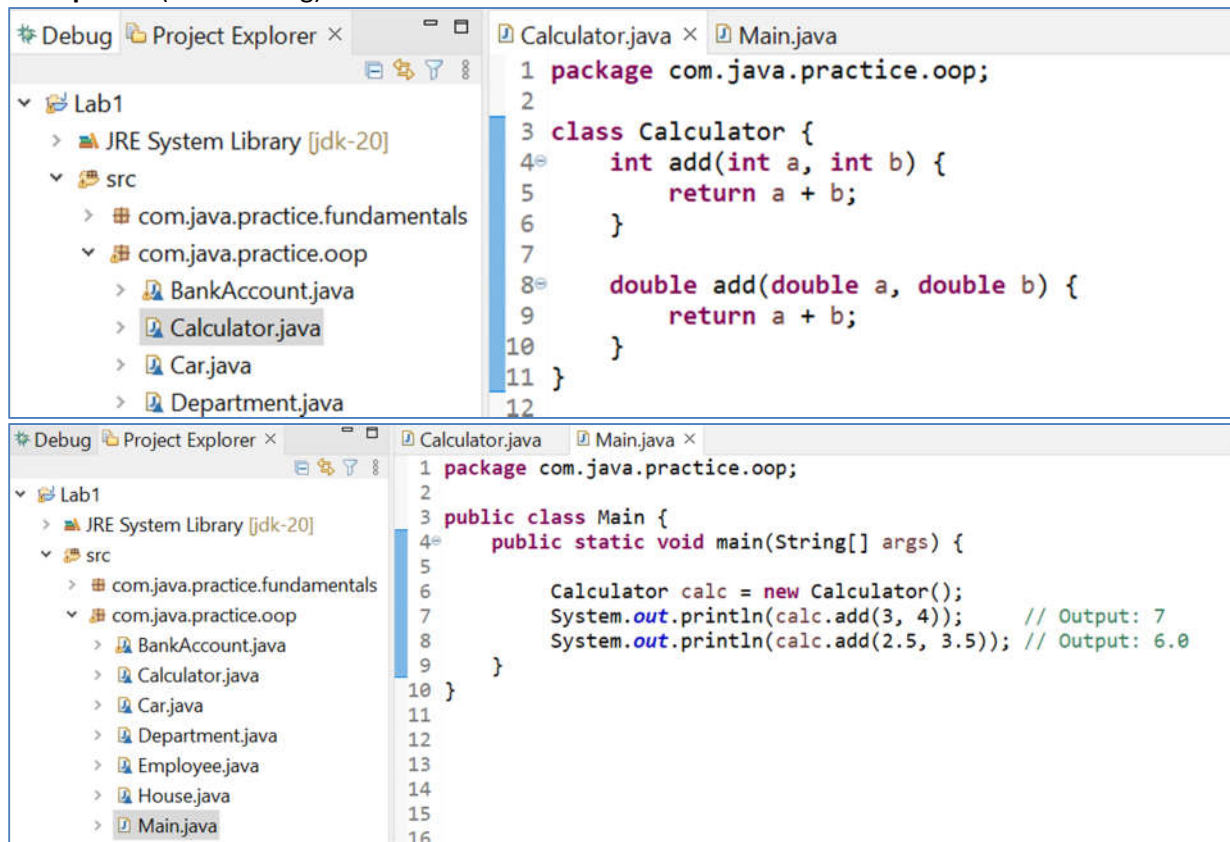
```
1 package com.java.practice.oop;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         Car car = new Car("Toyota", "V8", 4);
7         car.start();
8     }
9 }
10
11
12
```

Polymorphism

Polymorphism allows methods to take many forms. It can be achieved through method overloading and method overriding.

- **Method Overloading:** Same method name but different parameters.
- **Method Overriding:** Subclass redefines a method from its superclass.

Example # 6 (Overloading)

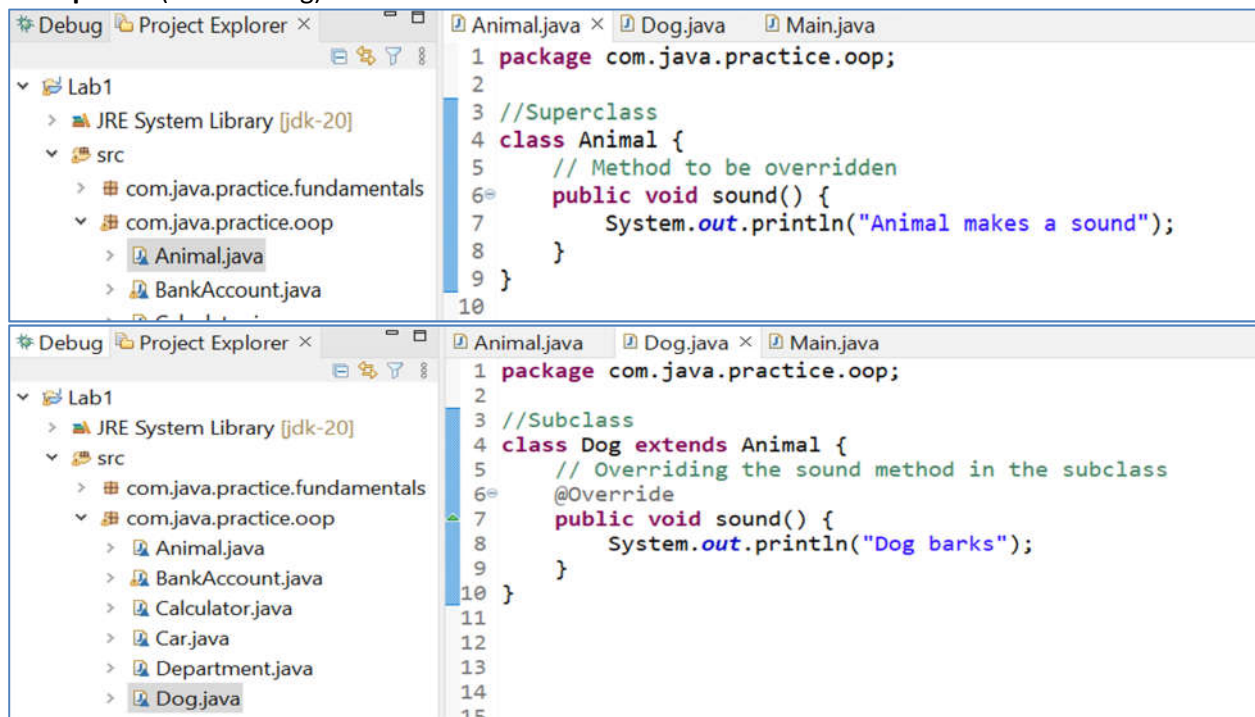


The screenshot shows an IDE with two tabs: Calculator.java and Main.java. The Project Explorer on the left shows a package structure: Lab1 > JRE System Library [jdk-20] > src > com.java.practice.fundamentals > com.java.practice.oop > Calculator.java. The Calculator.java file contains two overloaded methods: an integer add method and a double add method. The Main.java file contains a main method that calls both add methods with integer and double arguments, with comments indicating the expected output.

```
1 package com.java.practice.oop;
2
3 class Calculator {
4     int add(int a, int b) {
5         return a + b;
6     }
7
8     double add(double a, double b) {
9         return a + b;
10    }
11 }
12
```

```
1 package com.java.practice.oop;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         Calculator calc = new Calculator();
7         System.out.println(calc.add(3, 4)); // Output: 7
8         System.out.println(calc.add(2.5, 3.5)); // Output: 6.0
9     }
10 }
11
12
13
14
15
16
```

Example # 7 (Overloading)



The screenshot shows an IDE with two tabs: Animal.java and Dog.java. The Project Explorer on the left shows a package structure: Lab1 > JRE System Library [jdk-20] > src > com.java.practice.fundamentals > com.java.practice.oop > Animal.java. The Animal.java file contains a superclass with a sound method. The Dog.java file contains a subclass that overrides the sound method. The Main.java file is also visible in the tabs.

```
1 package com.java.practice.oop;
2
3 //Superclass
4 class Animal {
5     // Method to be overridden
6     public void sound() {
7         System.out.println("Animal makes a sound");
8     }
9 }
10
```

```
1 package com.java.practice.oop;
2
3 //Subclass
4 class Dog extends Animal {
5     // Overriding the sound method in the subclass
6     @Override
7     public void sound() {
8         System.out.println("Dog barks");
9     }
10 }
11
12
13
14
15
```

Abstraction

Abstraction is the process of hiding the implementation details and showing only the essential features of an object. It can be achieved using **abstract classes** or **interfaces**.

Abstract Class

An **abstract class** is a class that cannot be instantiated and may contain abstract methods (methods without a body) as well as concrete methods (methods with a body). A subclass must implement the abstract methods of an abstract class.

Key Features of Abstract Classes:

- Can have both abstract and concrete methods.
- Can have member variables (fields) and constructors.
- Supports single inheritance. A class can extend only one abstract class.
- Methods can have any access specifier (public, private, protected).
- Can have static methods.

Example # 8

```
1 package com.java.practice.oop;
2
3 class Dog extends Animal {
4     @Override
5     public void sound() {
6         System.out.println("Dog barks");
7     }
8 }
9
10
11
12
13
14
15
16
17
```

```
1 package com.java.practice.oop;
2
3 public class Main {
4     public static void main(String[] args) {
5
6
7         Dog dog = new Dog();
8         dog.sound(); // Calls the overridden method in the Dog class
9         dog.sleep(); // Calls the concrete method from the Animal class
10     }
11 }
12
13
14
15
16
17
18
19
20
21
```

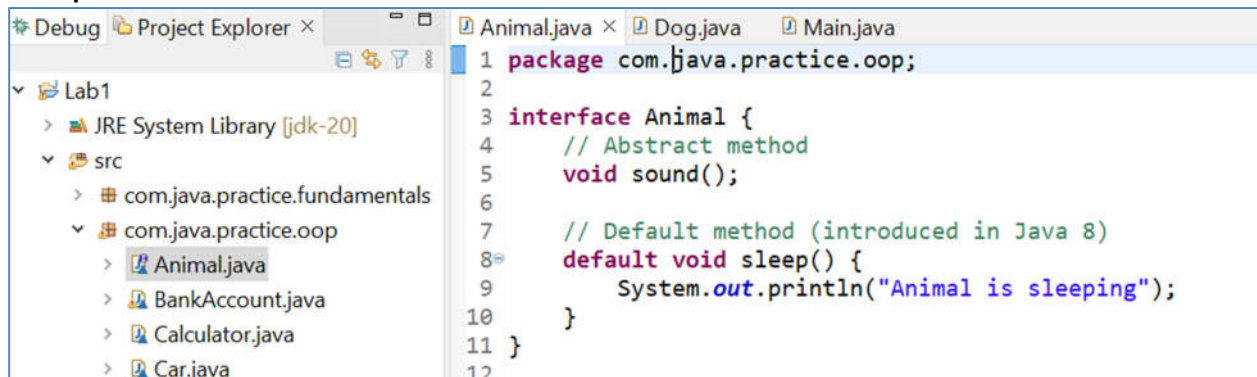
Interface

An **interface** is a completely abstract class that defines a list of methods that implementing classes must provide. From **Java 8**, interfaces can also include **default methods** and **static methods**, with some behavior provided. Interfaces are meant to define a contract that implementing classes must follow.

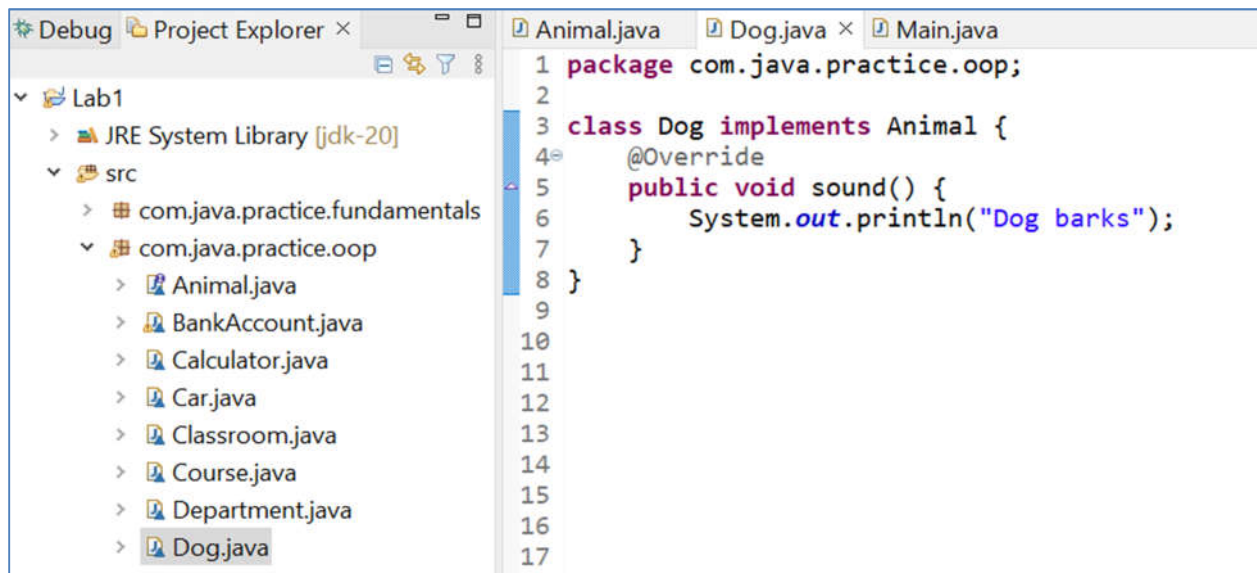
Key Features of Interfaces:

- **All methods are abstract by default** (before Java 8).
- Can have **default methods** (from Java 8) that provide a default implementation.
- **Cannot have instance variables**, but can have static and final variables (constants).
- **Supports multiple inheritance**. A class can implement multiple interfaces.
- All methods are implicitly `public` and abstract, and all variables are `public static final` by default.
- **No constructors**.

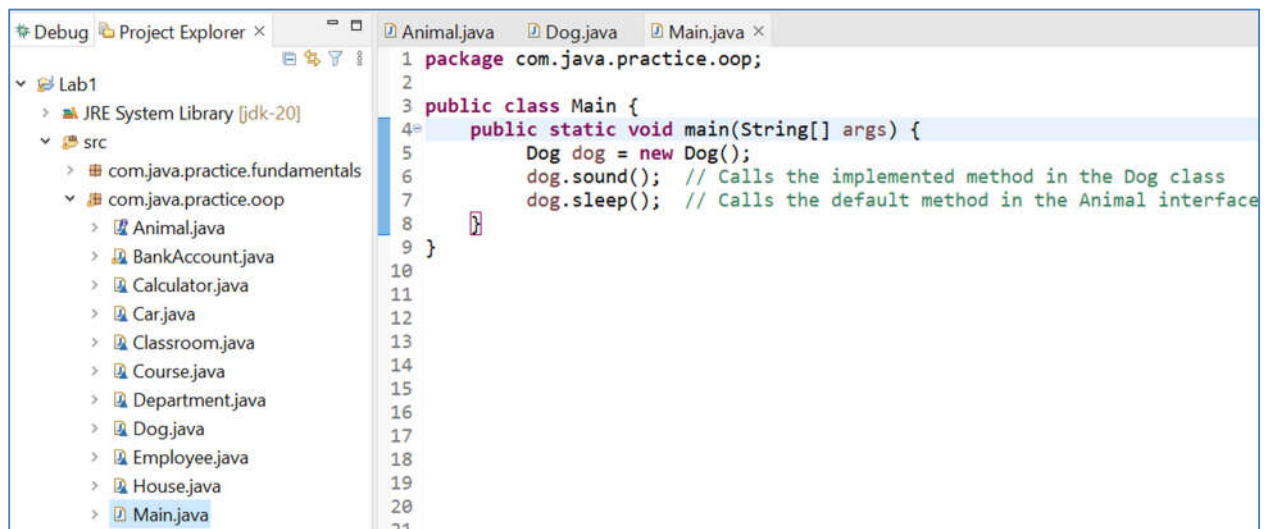
Example # 9



```
1 package com.java.practice.oop;
2
3 interface Animal {
4     // Abstract method
5     void sound();
6
7     // Default method (introduced in Java 8)
8     default void sleep() {
9         System.out.println("Animal is sleeping");
10    }
11 }
12
```



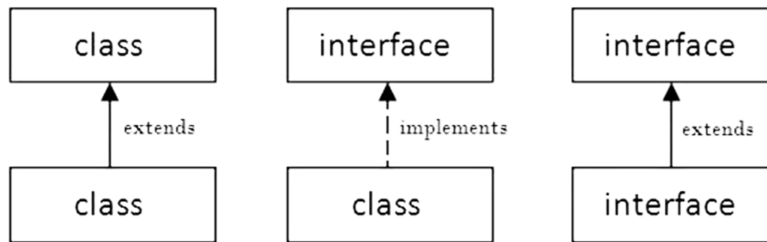
```
1 package com.java.practice.oop;
2
3 class Dog implements Animal {
4     @Override
5     public void sound() {
6         System.out.println("Dog barks");
7     }
8 }
9
10
11
12
13
14
15
16
17
```



```
1 package com.java.practice.oop;
2
3 public class Main {
4     public static void main(String[] args) {
5         Dog dog = new Dog();
6         dog.sound(); // Calls the implemented method in the Dog class
7         dog.sleep(); // Calls the default method in the Animal interface
8     }
9 }
10
11
12
13
14
15
16
17
18
19
20
21
```

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Relationships among classes

Each of these relationships, **association**, **aggregation**, and **composition**, can be implemented in various contexts to reflect real-world scenarios.

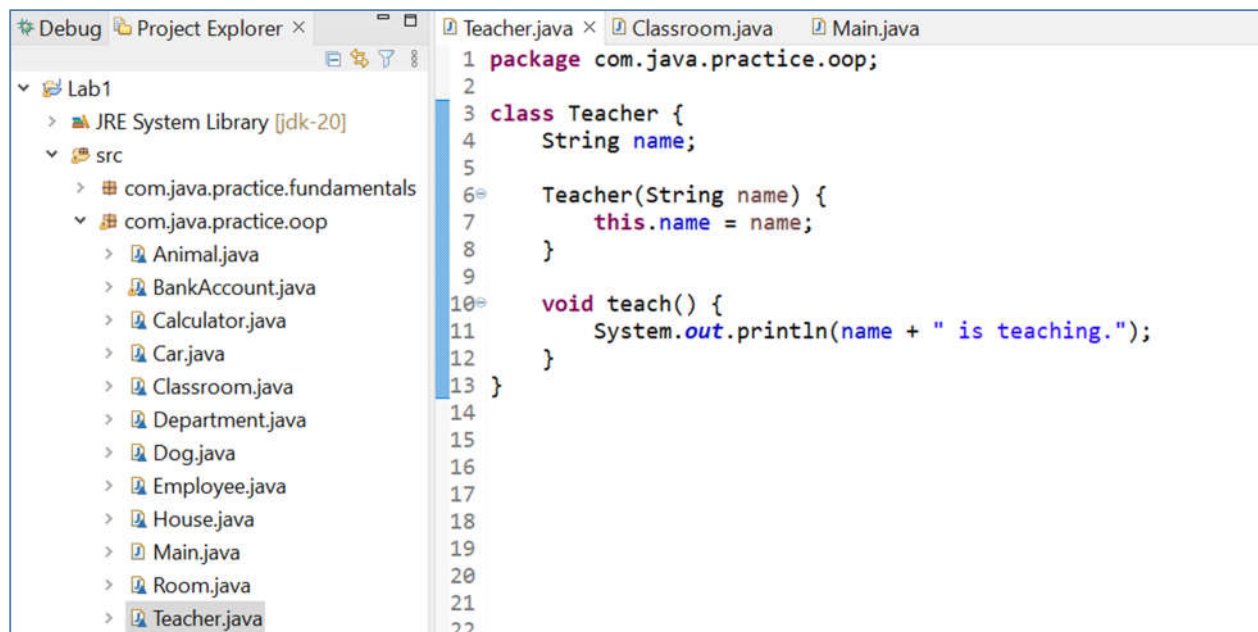
Concept	Definition	Strength
Association	A general relationship between two classes that can exist independently.	Weak
Aggregation	A relationship where one class contains another, but the contained object can exist independently.	Medium
Composition	A relationship where one class contains another and the contained object cannot exist independently.	Strong

Association

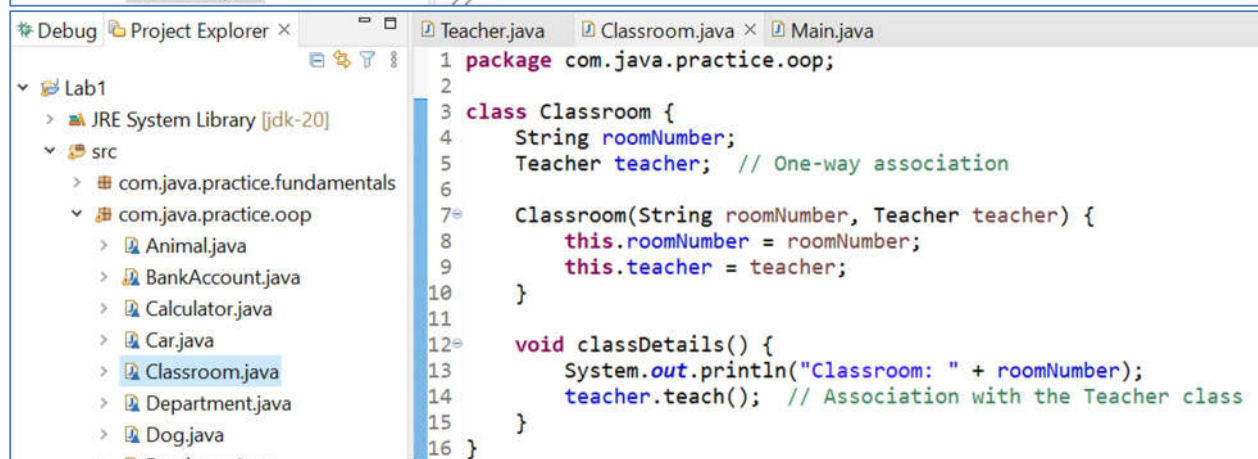
Association is a general relationship between two independent classes. It can be either one-way (unidirectional) or two-way (bidirectional).

Example 10: One-Way (Unidirectional) Association

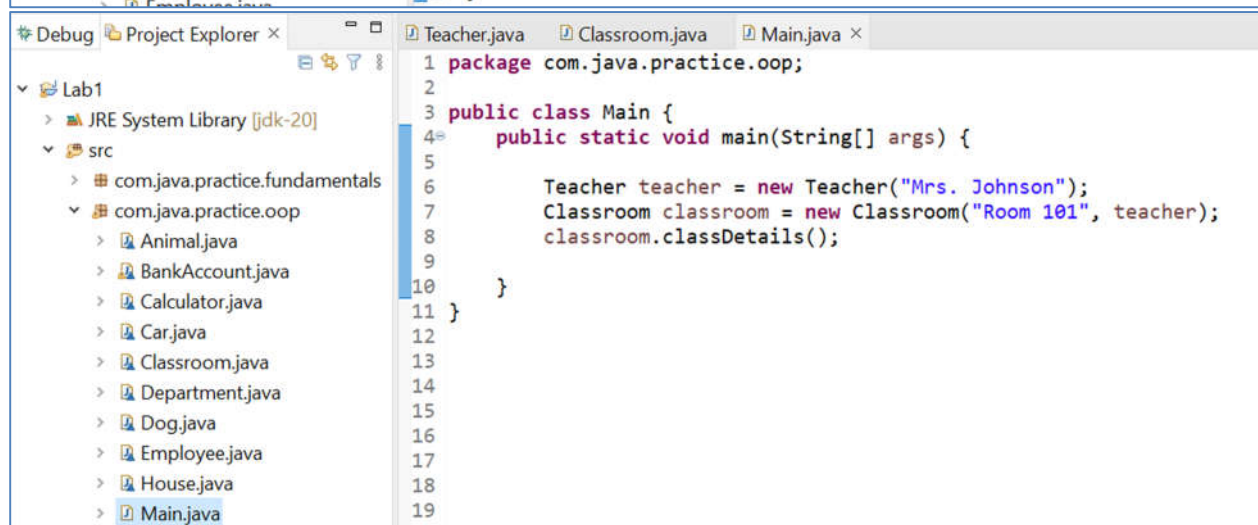
In this example, we have two independent classes, `Teacher` and `Classroom`. A `Teacher` can teach in a `Classroom`, but a classroom does not know anything about the teacher.



```
1 package com.java.practice.oop;
2
3 class Teacher {
4     String name;
5
6     Teacher(String name) {
7         this.name = name;
8     }
9
10    void teach() {
11        System.out.println(name + " is teaching.");
12    }
13 }
```



```
1 package com.java.practice.oop;
2
3 class Classroom {
4     String roomNumber;
5     Teacher teacher; // One-way association
6
7     Classroom(String roomNumber, Teacher teacher) {
8         this.roomNumber = roomNumber;
9         this.teacher = teacher;
10    }
11
12    void classDetails() {
13        System.out.println("Classroom: " + roomNumber);
14        teacher.teach(); // Association with the Teacher class
15    }
16 }
```



```
1 package com.java.practice.oop;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         Teacher teacher = new Teacher("Mrs. Johnson");
7         Classroom classroom = new Classroom("Room 101", teacher);
8         classroom.classDetails();
9
10    }
11 }
```

Example 11: two-way (bidirectional) Association

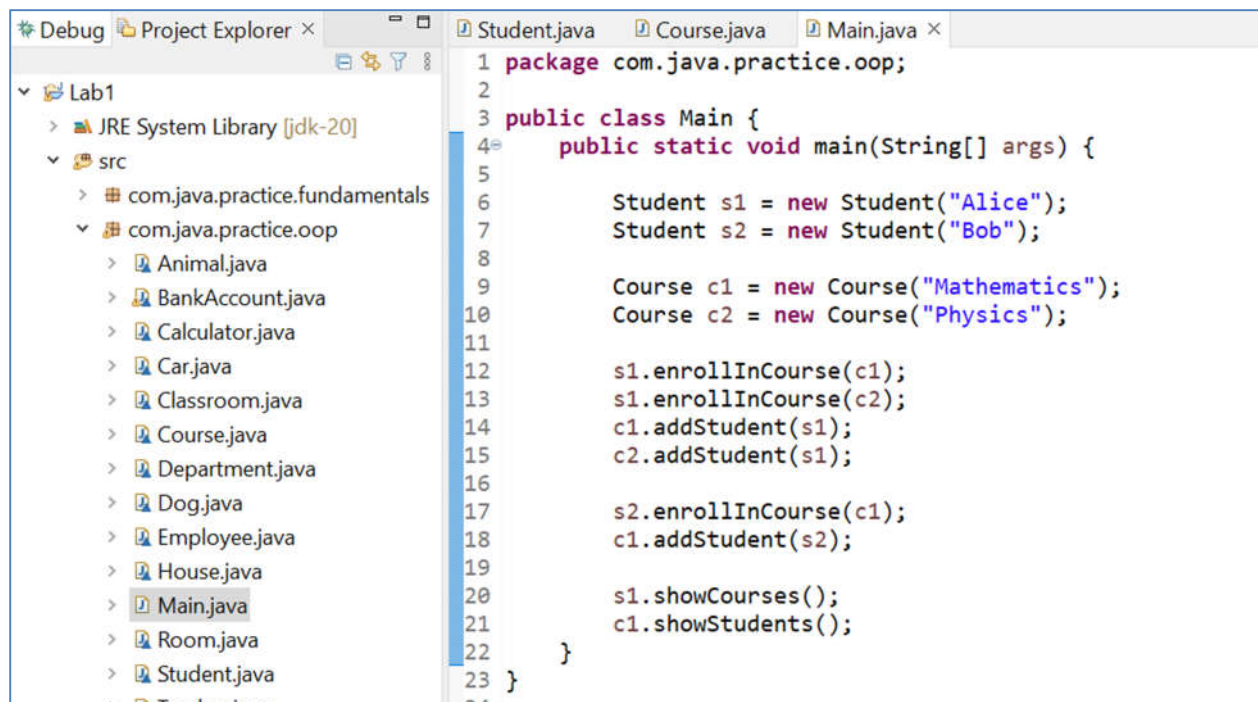
In a bidirectional association, both objects are aware of each other. For example, a `Student` is enrolled in a `Course`, and the `Course` knows about the `Student`.

The screenshot displays an IDE with two tabs open: `Student.java` and `Course.java`. The `Project Explorer` on the left shows the project structure, including the `src` directory and the `com.java.practice.oop` package. The `Student.java` file contains the following code:

```
1 package com.java.practice.oop;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 class Student {
7     String name;
8     List<Course> courses; // List of courses associated with the student
9
10    Student(String name) {
11        this.name = name;
12        this.courses = new ArrayList<>();
13    }
14
15    void enrollInCourse(Course course) {
16        courses.add(course);
17    }
18
19    void showCourses() {
20        System.out.println("Student: " + name);
21        for (Course course : courses) {
22            System.out.println("Enrolled in: " + course.courseName);
23        }
24    }
25 }
```

The `Course.java` file contains the following code:

```
1 package com.java.practice.oop;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 class Course {
7     String courseName;
8     List<Student> students; // List of students associated with the course
9
10    Course(String courseName) {
11        this.courseName = courseName;
12        this.students = new ArrayList<>();
13    }
14
15    void addStudent(Student student) {
16        students.add(student);
17    }
18
19    void showStudents() {
20        System.out.println("Course: " + courseName);
21        for (Student student : students) {
22            System.out.println("Student: " + student.name);
23        }
24    }
25 }
```



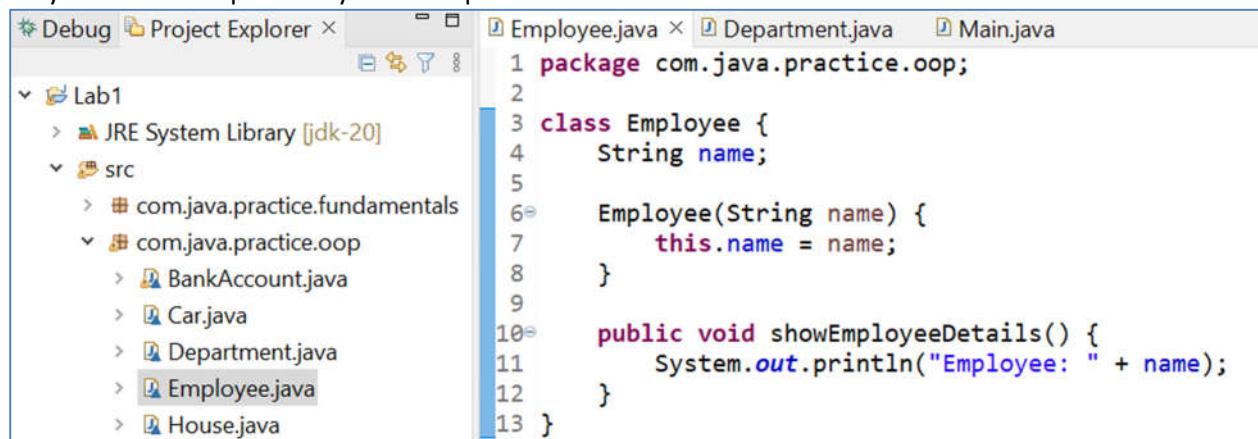
```
1 package com.java.practice.oop;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         Student s1 = new Student("Alice");
7         Student s2 = new Student("Bob");
8
9         Course c1 = new Course("Mathematics");
10        Course c2 = new Course("Physics");
11
12        s1.enrollInCourse(c1);
13        s1.enrollInCourse(c2);
14        c1.addStudent(s1);
15        c2.addStudent(s1);
16
17        s2.enrollInCourse(c1);
18        c1.addStudent(s2);
19
20        s1.showCourses();
21        c1.showStudents();
22    }
23 }
```

Aggregation

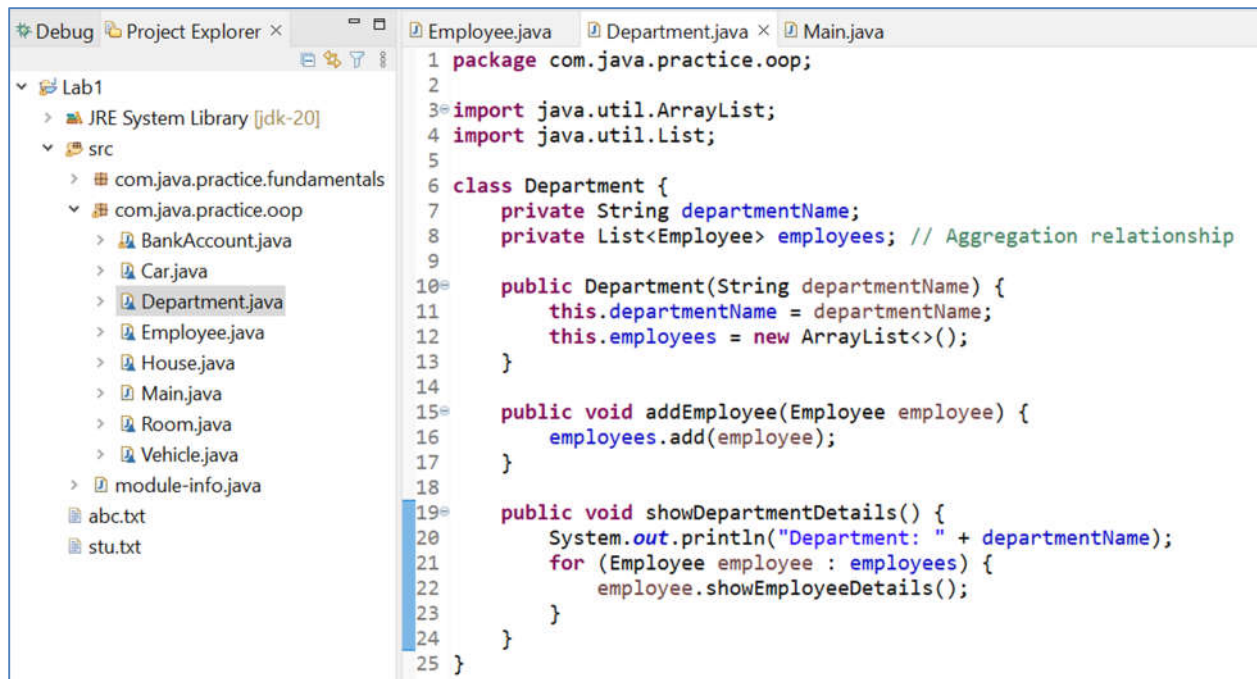
Aggregation represents a relationship where one class contains another class, but the contained class can exist independently.

Example # 12

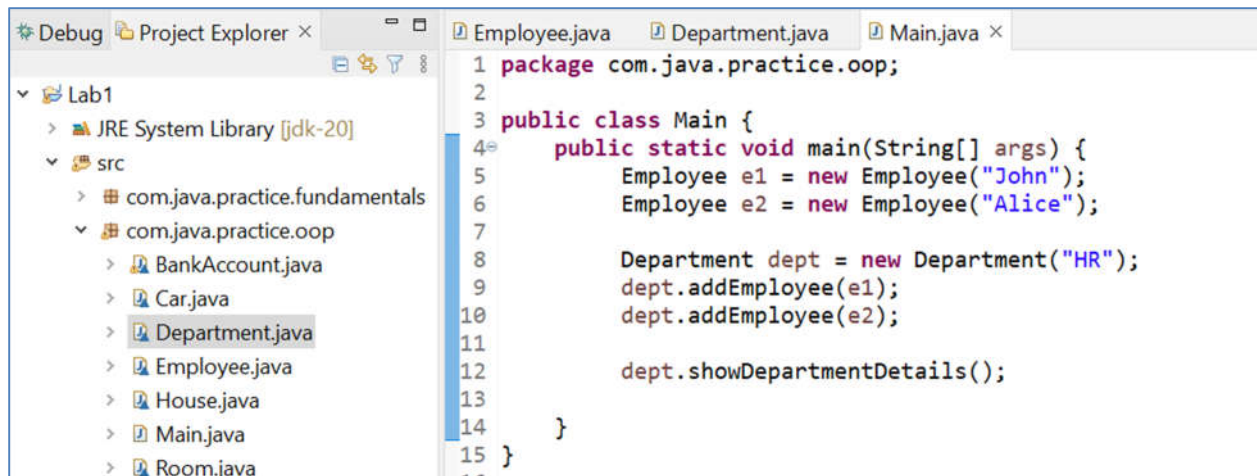
Aggregation is used between `Department` and `Employee`. Employees are part of the department, but they can exist independently of the department.



```
1 package com.java.practice.oop;
2
3 class Employee {
4     String name;
5
6     Employee(String name) {
7         this.name = name;
8     }
9
10    public void showEmployeeDetails() {
11        System.out.println("Employee: " + name);
12    }
13 }
```

```
1 package com.java.practice.oop;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 class Department {
7     private String departmentName;
8     private List<Employee> employees; // Aggregation relationship
9
10    public Department(String departmentName) {
11        this.departmentName = departmentName;
12        this.employees = new ArrayList<>();
13    }
14
15    public void addEmployee(Employee employee) {
16        employees.add(employee);
17    }
18
19    public void showDepartmentDetails() {
20        System.out.println("Department: " + departmentName);
21        for (Employee employee : employees) {
22            employee.showEmployeeDetails();
23        }
24    }
25 }
```



```
1 package com.java.practice.oop;
2
3 public class Main {
4     public static void main(String[] args) {
5         Employee e1 = new Employee("John");
6         Employee e2 = new Employee("Alice");
7
8         Department dept = new Department("HR");
9         dept.addEmployee(e1);
10        dept.addEmployee(e2);
11
12        dept.showDepartmentDetails();
13    }
14 }
15 }
```

Composition

Composition is a strong association where the contained object cannot exist without the parent object. If the parent is destroyed, the child is destroyed as well.

Example # 13

Composition is shown between the `House` and `Room` classes. Rooms cannot exist without the house, and if the house is destroyed, all the rooms are also destroyed.

```

1 package com.java.practice.oop;
2
3
4 class Room {
5     private String roomName;
6
7     Room(String roomName) {
8         this.roomName = roomName;
9     }
10
11     public void showRoomDetails() {
12         System.out.println("Room: " + roomName);
13     }
14 }

```

```

1 package com.java.practice.oop;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 class House {
7     private String address;
8     private List<Room> rooms; // Composition relationship
9
10    public House(String address) {
11        this.address = address;
12        rooms = new ArrayList<>();
13    }
14
15    public void addRoom(String roomName) {
16        rooms.add(new Room(roomName)); // Creating and adding rooms to the house
17    }
18
19    public void showHouseDetails() {
20        System.out.println("House Address: " + address);
21        for (Room room : rooms) {
22            room.showRoomDetails();
23        }
24    }
25 }

```

```

1 package com.java.practice.oop;
2
3 public class Main {
4     public static void main(String[] args) {
5
6         House house = new House("123 Main St");
7         house.addRoom("Living Room");
8         house.addRoom("Bedroom");
9         house.addRoom("Dinning Room");
10
11         house.showHouseDetails();
12     }
13 }

```

Task

Marks: 13 (Examples) + 7 (Task – OOP Concept, Relationships)

Implement the following class diagram with the concepts of OOP

