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

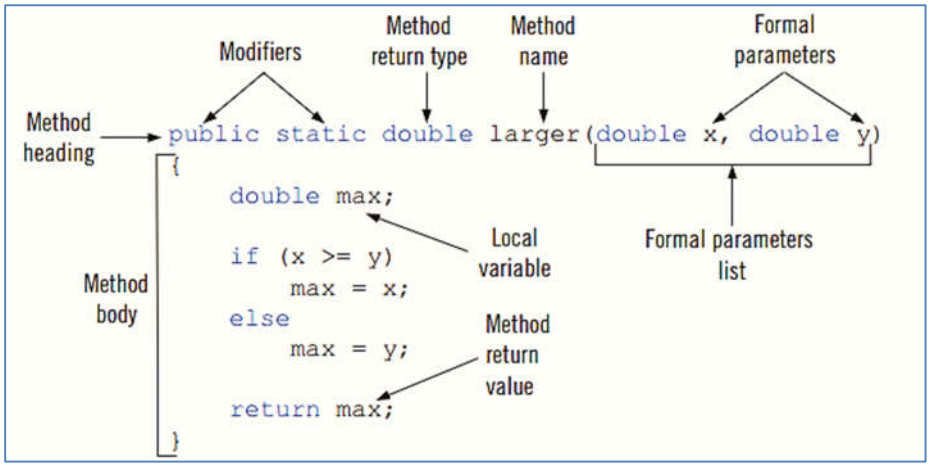
SYNTAX: VALUE-RETURNING METHOD

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
}
```

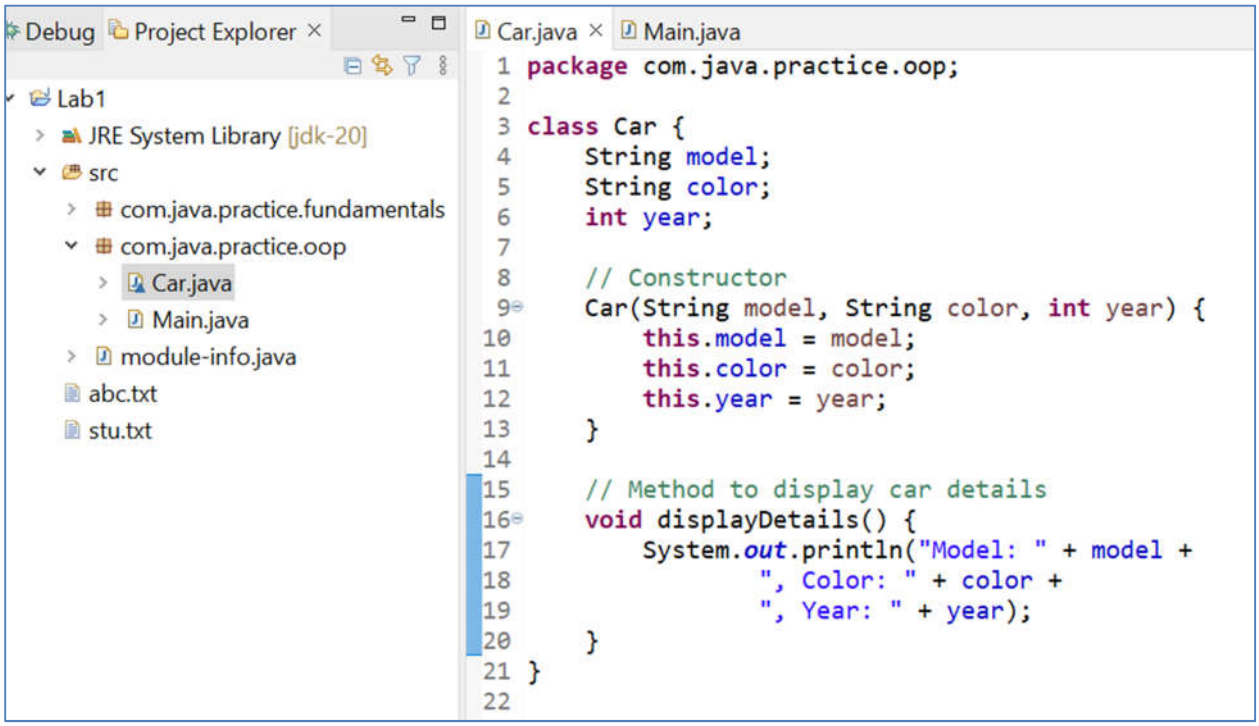
Example # 2

```
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
```

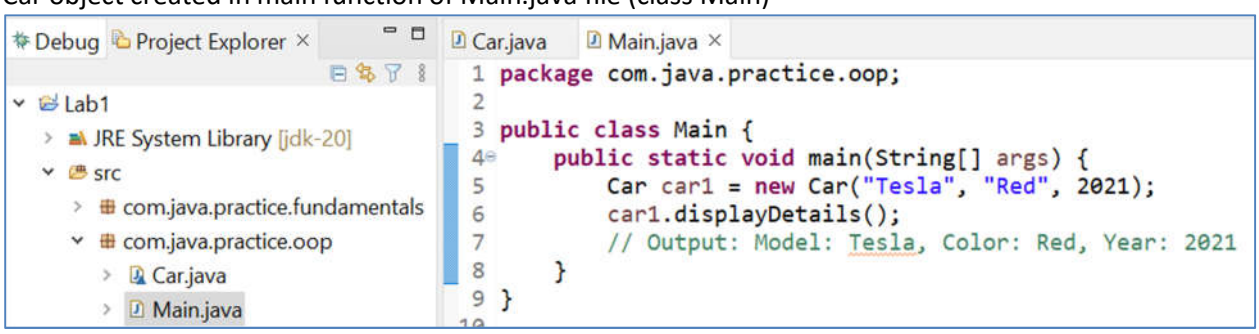
Core OOP Concepts

- 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



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



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

Debug | Project Explorer

- Lab1
  - JRE System Library [jdk-20]
  - src
    - com.java.practice.fundamentals
    - com.java.practice.oop
      - BankAccount.java
      - Car.java
      - Main.java
    - module-info.java
  - abc.txt
  - stu.txt

Main.java | BankAccount.java

```java
package com.java.practice.oop;

class BankAccount {
    private String accountNumber;
    private double balance;

    public BankAccount(String accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }

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

    // Setter method with business rule
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }
}
```

Debug | Project Explorer

- Lab1
  - JRE System Library [jdk-20]
  - src
    - com.java.practice.fundamentals
    - com.java.practice.oop
      - BankAccount.java
      - Car.java
      - Main.java
    - module-info.java
  - abc.txt
  - stu.txt

BankAccount.java | Main.java

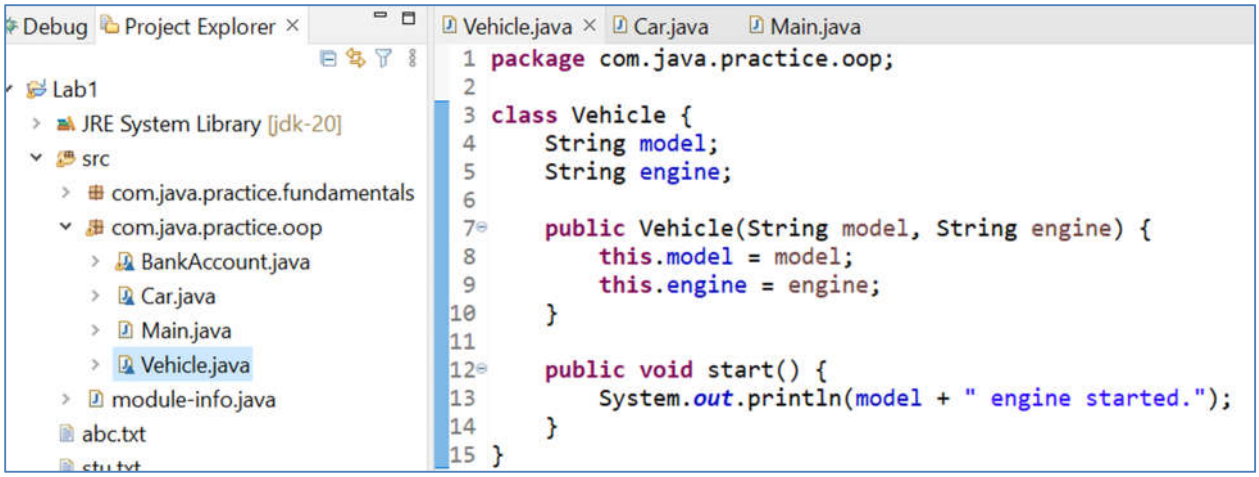
```java
package com.java.practice.oop;

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Tesla", "Red", 2021);
        car1.displayDetails();
        // Output: Model: Tesla, Color: Red, Year: 2021


        BankAccount account = new BankAccount("HBL0001110120",1000);
        account.deposit(500);
        System.out.println("Current balance: " + account.getBalance());
        // Output: 1500
    }
}
```

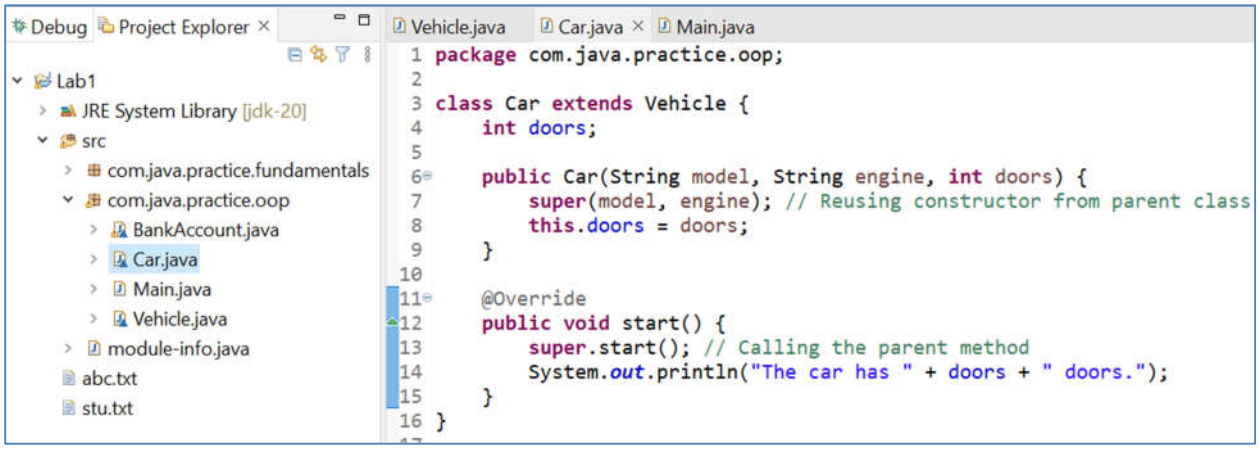
Inheritance

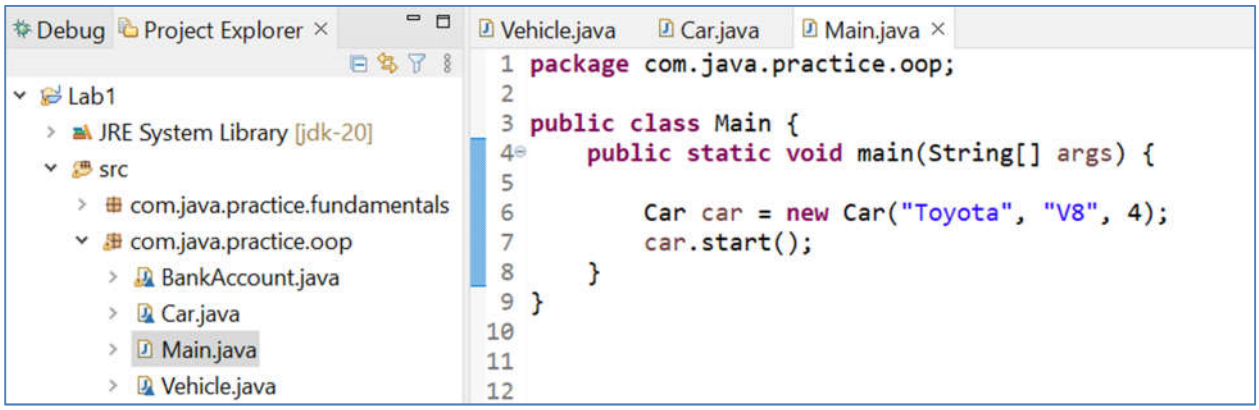
Inheritance allows one class (subclass) to inherit properties and methods from another class (superclass). This promotes code reusability and method overriding.

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.







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)

Debug | Project Explorer

- Lab1
  - JRE System Library [jdk-20]
  - src
    - com.java.practice.fundamentals
    - com.java.practice.oop
      - BankAccount.java
      - Calculator.java
      - Car.java
      - Department.java

Calculator.java | Main.java

```java
package com.java.practice.oop;

class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
}
```

Debug | Project Explorer

- Lab1
  - JRE System Library [jdk-20]
  - src
    - com.java.practice.fundamentals
    - com.java.practice.oop
      - BankAccount.java
      - Calculator.java
      - Car.java
      - Department.java
      - Employee.java
      - House.java
      - Main.java

Calculator.java | Main.java

```java
package com.java.practice.oop;

public class Main {
    public static void main(String[] args) {

        Calculator calc = new Calculator();
        System.out.println(calc.add(3, 4));      // Output: 7
        System.out.println(calc.add(2.5, 3.5)); // Output: 6.0
    }
}
```

Example # 7 (Overloading)

Debug | Project Explorer

- Lab1
  - JRE System Library [jdk-20]
  - src
    - com.java.practice.fundamentals
    - com.java.practice.oop
      - Animal.java
      - BankAccount.java

Animal.java | Dog.java | Main.java

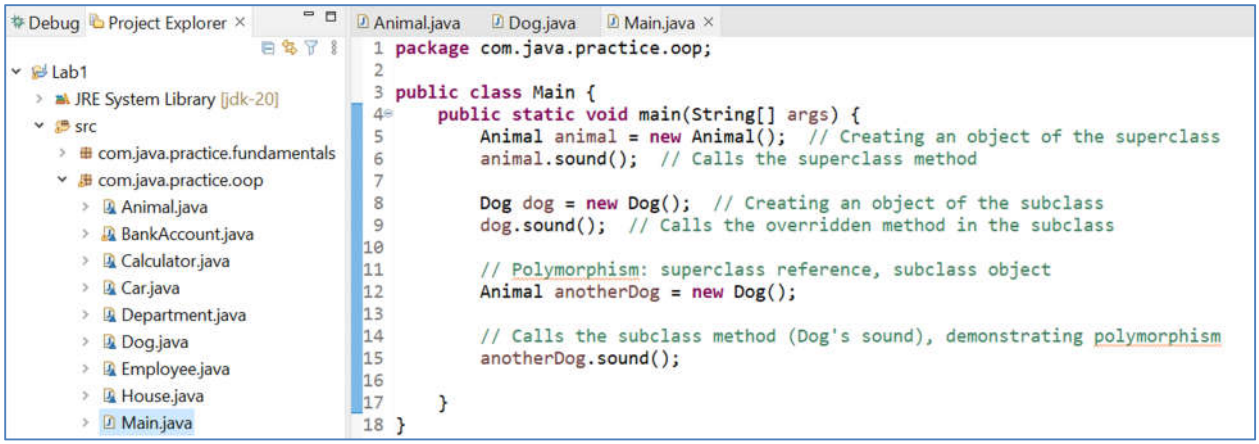
```java
package com.java.practice.oop;

//Superclass
class Animal {
    // Method to be overridden
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}
```

Debug | Project Explorer

- Lab1
  - JRE System Library [jdk-20]
  - src
    - com.java.practice.fundamentals
    - com.java.practice.oop
      - Animal.java
      - BankAccount.java
      - Calculator.java
      - Car.java
      - Department.java
      - Dog.java

Animal.java | Dog.java | Main.java

```java
package com.java.practice.oop;

//Subclass
class Dog extends Animal {
    // Overriding the sound method in the subclass
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}
```



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**.

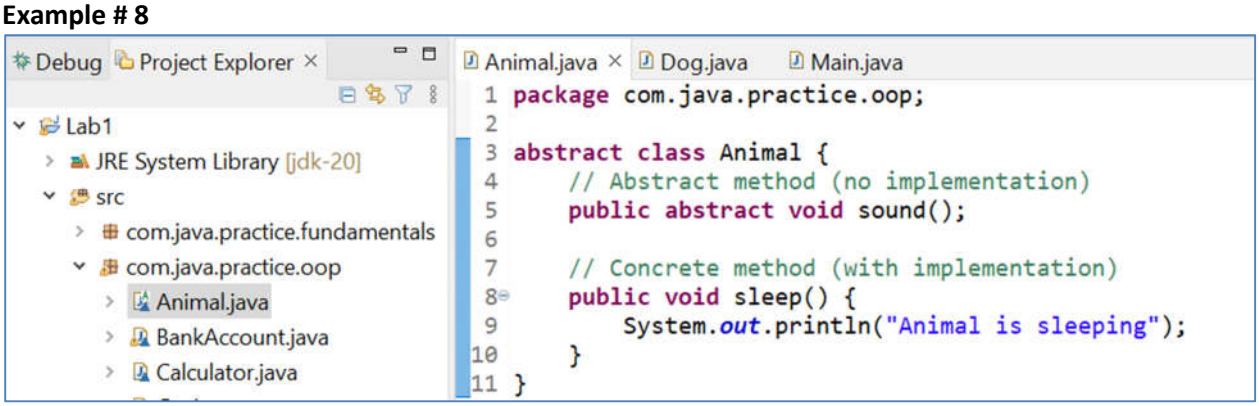
Abstract Class

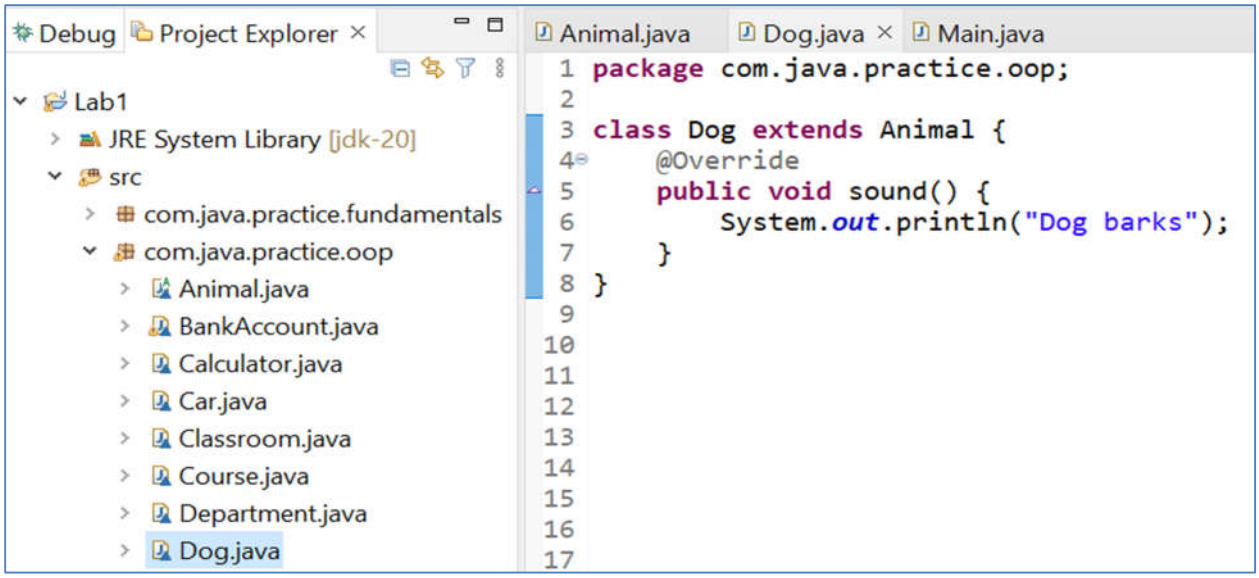
An **abstract class** is a class that cannot be instantiated and may contain abstract methods (methods without a body) as well as concrete methods (methods with a body). A subclass must implement the abstract methods of an abstract class.

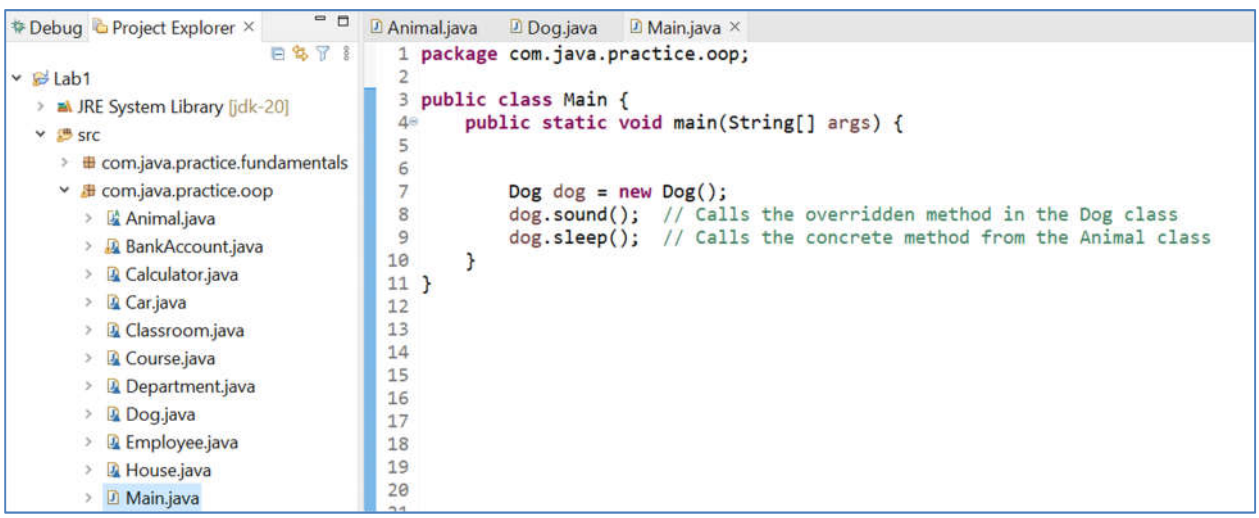
Key Features of Abstract Classes:

- 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







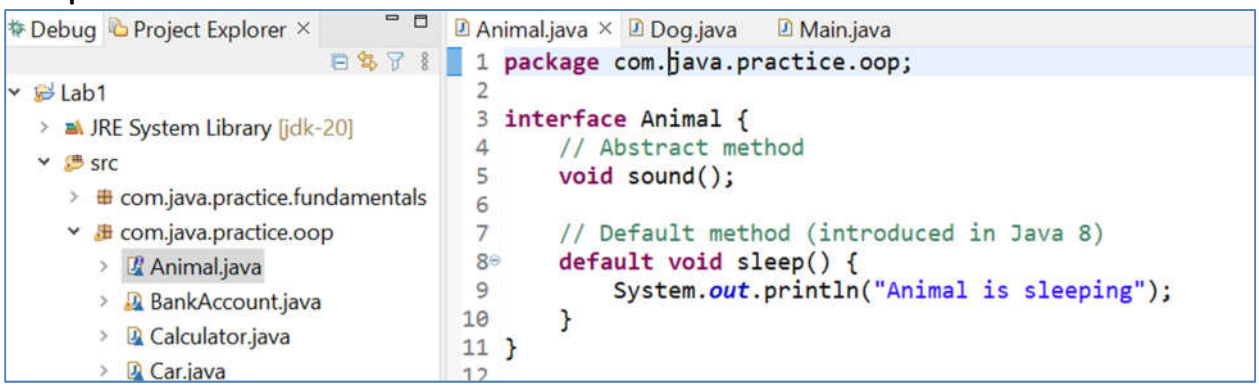
Interface

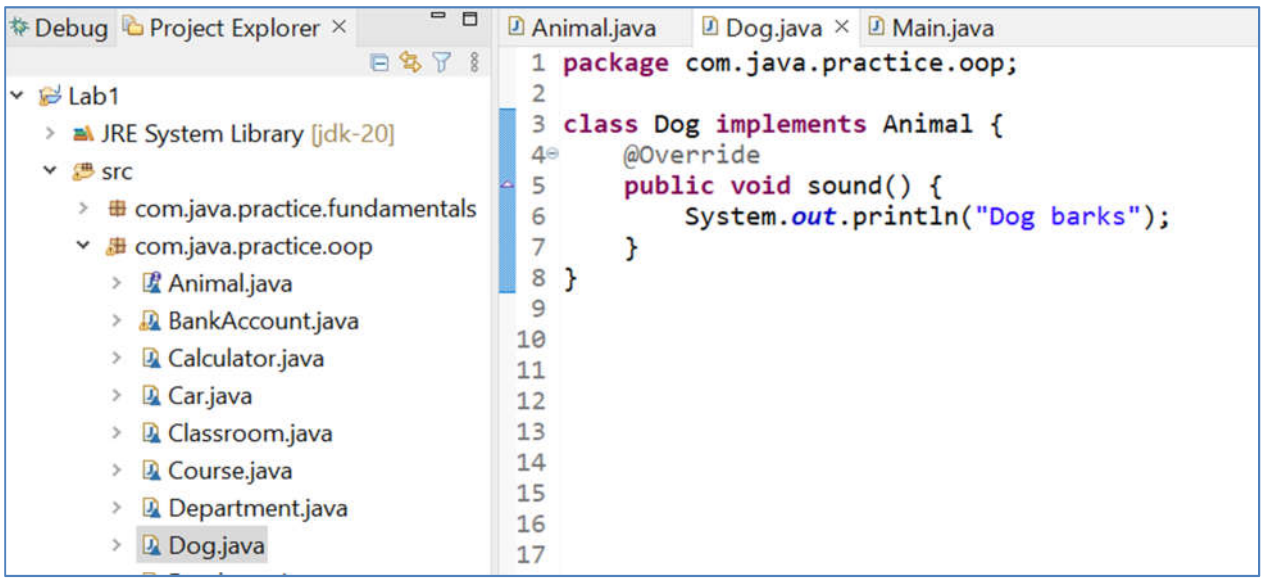
An **interface** is a completely abstract class that defines a list of methods that implementing classes must provide. From **Java 8**, interfaces can also include **default methods** and **static methods**, with some behavior provided. Interfaces are meant to define a contract that implementing classes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

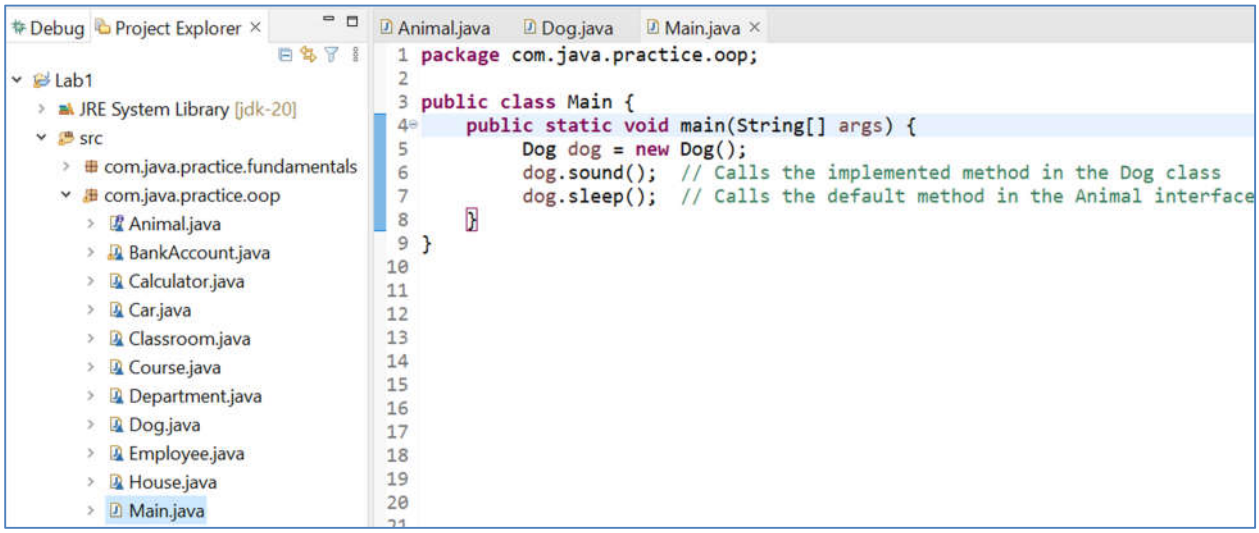
Key Features of Interfaces:

- **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

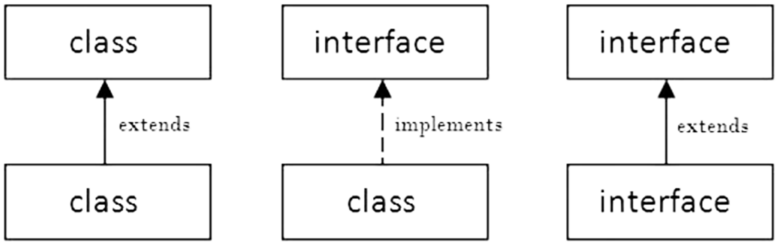

Debug
Project Explorer
Lab1
JRE System Library [jdk-20]
src
com.java.practice.fundamentals
com.java.practice.oop
Animal.java
BankAccount.java
Calculator.java
Car.java
Animal.java
Dog.java
Main.java
package com.java.practice.oop;
interface Animal {
// Abstract method
void sound();
// Default method (introduced in Java 8)
default void sleep() {
System.out.println("Animal is sleeping");
}
}


Debug
Project Explorer
Lab1
JRE System Library [jdk-20]
src
com.java.practice.fundamentals
com.java.practice.oop
Animal.java
BankAccount.java
Calculator.java
Car.java
Classroom.java
Course.java
Department.java
Dog.java
Animal.java
Dog.java
Main.java
package com.java.practice.oop;
class Dog implements Animal {
@Override
public void sound() {
System.out.println("Dog barks");
}
}


Debug
Project Explorer
Lab1
JRE System Library [jdk-20]
src
com.java.practice.fundamentals
com.java.practice.oop
Animal.java
BankAccount.java
Calculator.java
Car.java
Classroom.java
Course.java
Department.java
Dog.java
Employee.java
House.java
Main.java
Animal.java
Dog.java
Main.java
package com.java.practice.oop;
public class Main {
public static void main(String[] args) {
Dog dog = new Dog();
dog.sound(); // Calls the implemented method in the Dog class
dog.sleep(); // Calls the default method in the Animal interface
}
}

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.

Debug | Project Explorer

- Lab1
  - JRE System Library [jdk-20]
  - src
    - com.java.practice.fundamentals
    - com.java.practice.oop
      - Animal.java
      - BankAccount.java
      - Calculator.java
      - Car.java
      - Classroom.java
      - Department.java
      - Dog.java
      - Employee.java
      - House.java
      - Main.java
      - Room.java
      - Teacher.java

Teacher.java | Classroom.java | Main.java

```java
package com.java.practice.oop;

class Teacher {
    String name;

    Teacher(String name) {
        this.name = name;
    }

    void teach() {
        System.out.println(name + " is teaching.");
    }
}
```

Debug | Project Explorer

- Lab1
  - JRE System Library [jdk-20]
  - src
    - com.java.practice.fundamentals
    - com.java.practice.oop
      - Animal.java
      - BankAccount.java
      - Calculator.java
      - Car.java
      - Classroom.java
      - Department.java
      - Dog.java

Teacher.java | Classroom.java | Main.java

```java
package com.java.practice.oop;

class Classroom {
    String roomNumber;
    Teacher teacher;  // One-way association

    Classroom(String roomNumber, Teacher teacher) {
        this.roomNumber = roomNumber;
        this.teacher = teacher;
    }

    void classDetails() {
        System.out.println("Classroom: " + roomNumber);
        teacher.teach();  // Association with the Teacher class
    }
}
```

Debug | Project Explorer

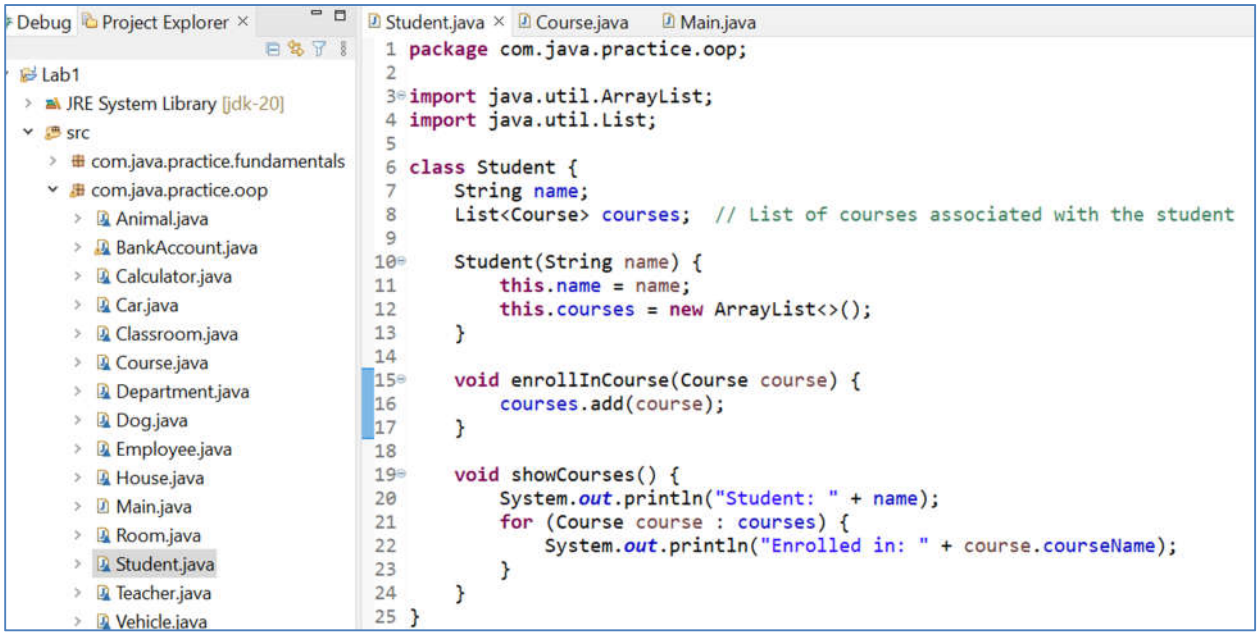
- Lab1
  - JRE System Library [jdk-20]
  - src
    - com.java.practice.fundamentals
    - com.java.practice.oop
      - Animal.java
      - BankAccount.java
      - Calculator.java
      - Car.java
      - Classroom.java
      - Department.java
      - Dog.java
      - Employee.java
      - House.java
      - Main.java

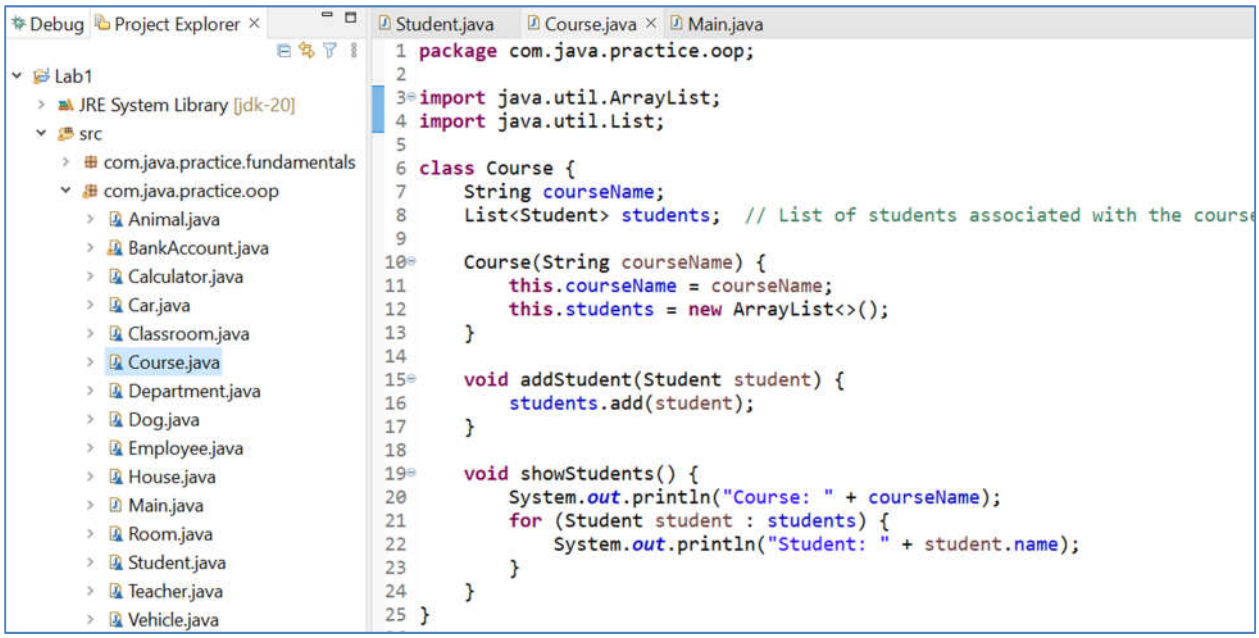
Teacher.java | Classroom.java | Main.java

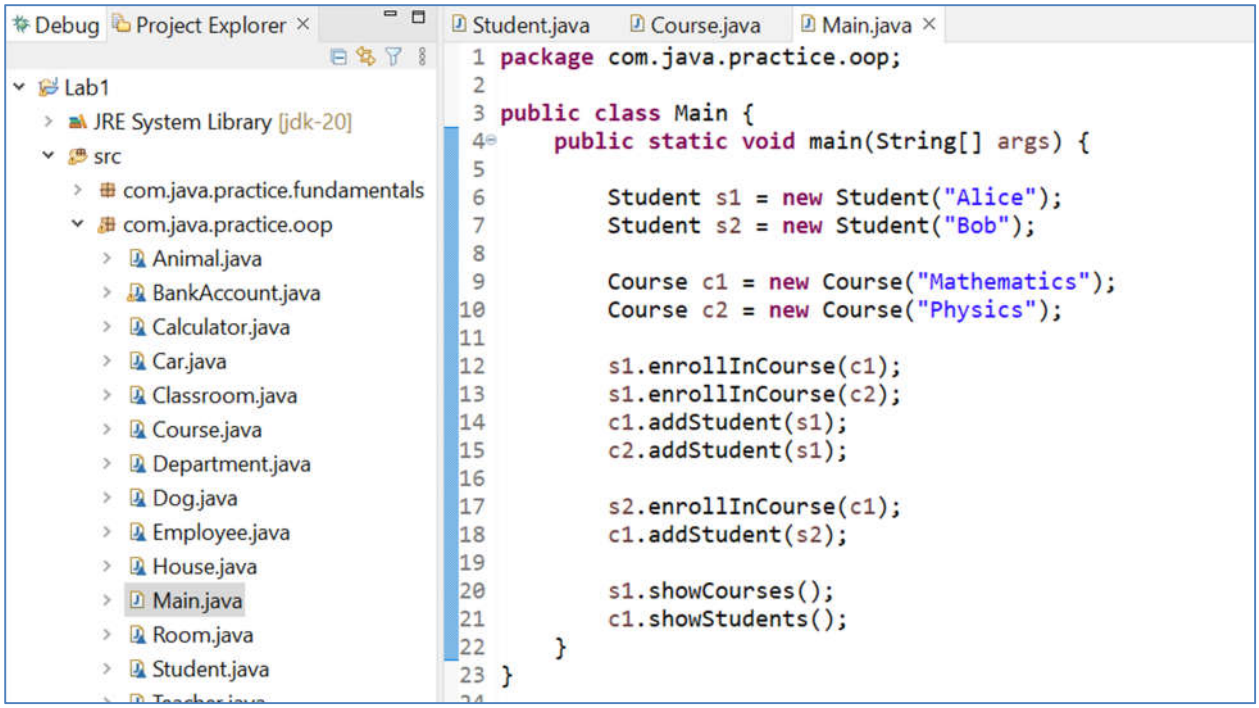
```java
package com.java.practice.oop;

public class Main {
    public static void main(String[] args) {

        Teacher teacher = new Teacher("Mrs. Johnson");
        Classroom classroom = new Classroom("Room 101", teacher);
        classroom.classDetails();

    }
}
```

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`.





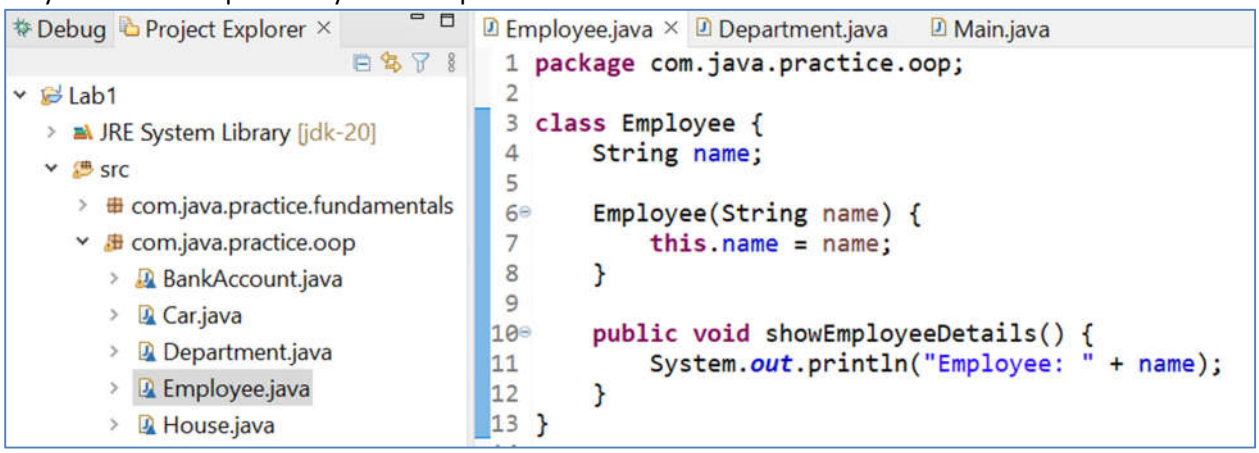


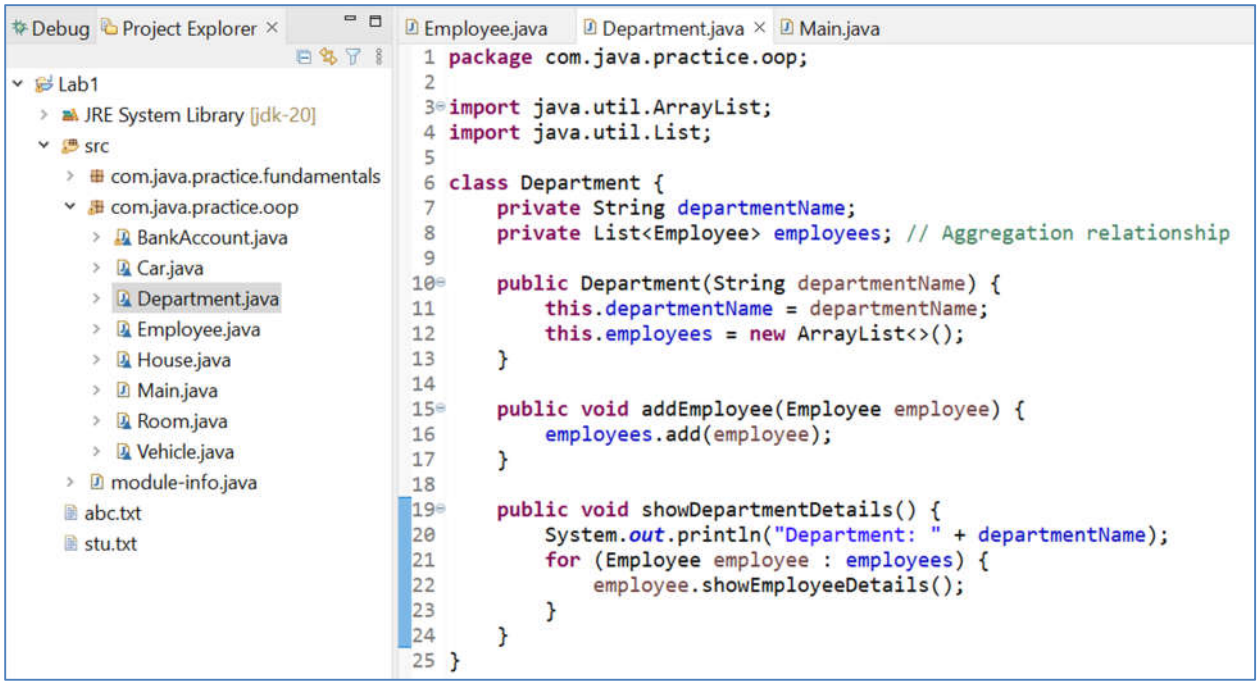
Aggregation

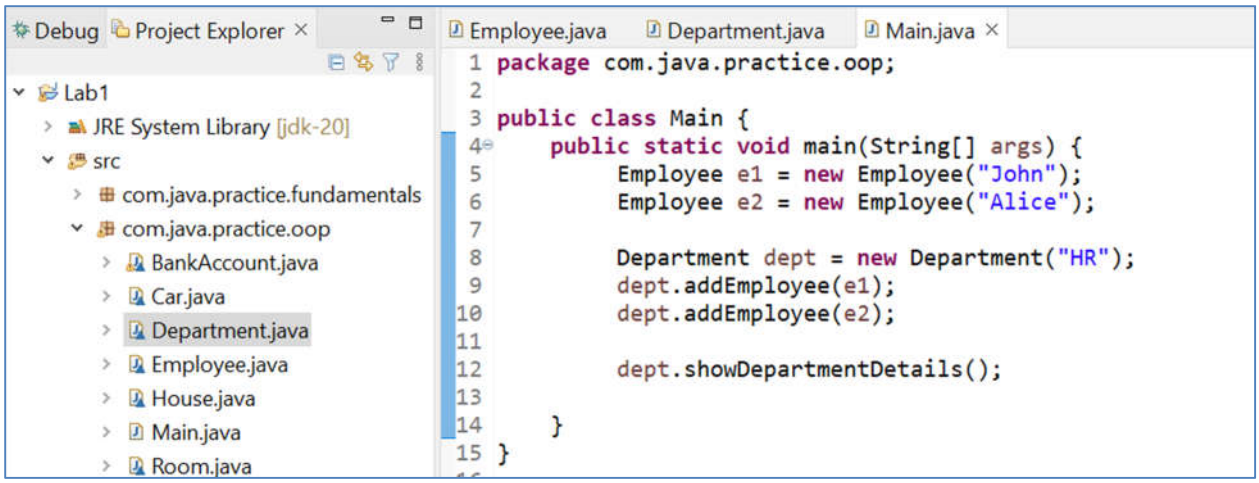
Aggregation represents a relationship where one class contains another class, but the contained class can exist independently.

Example # 12

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





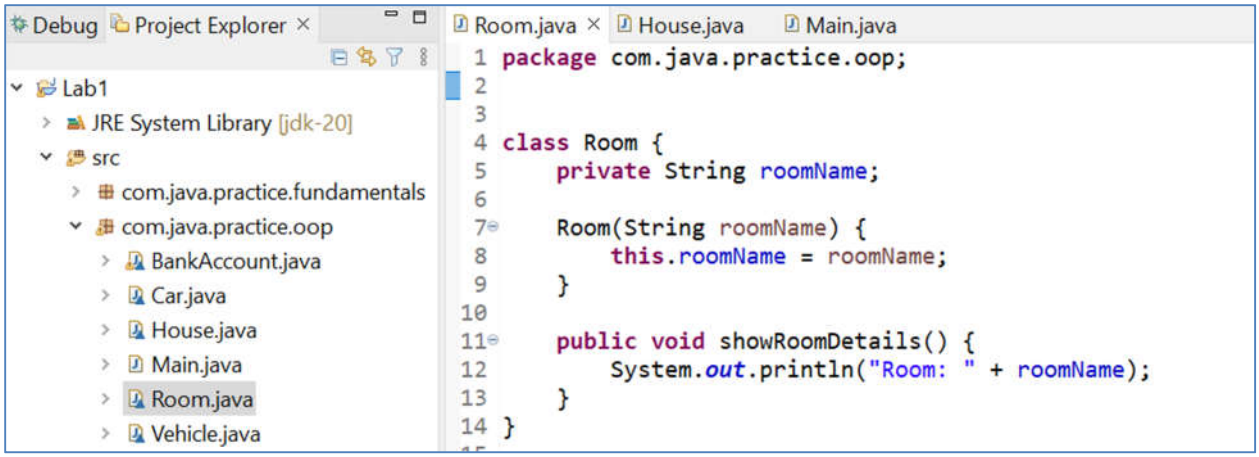


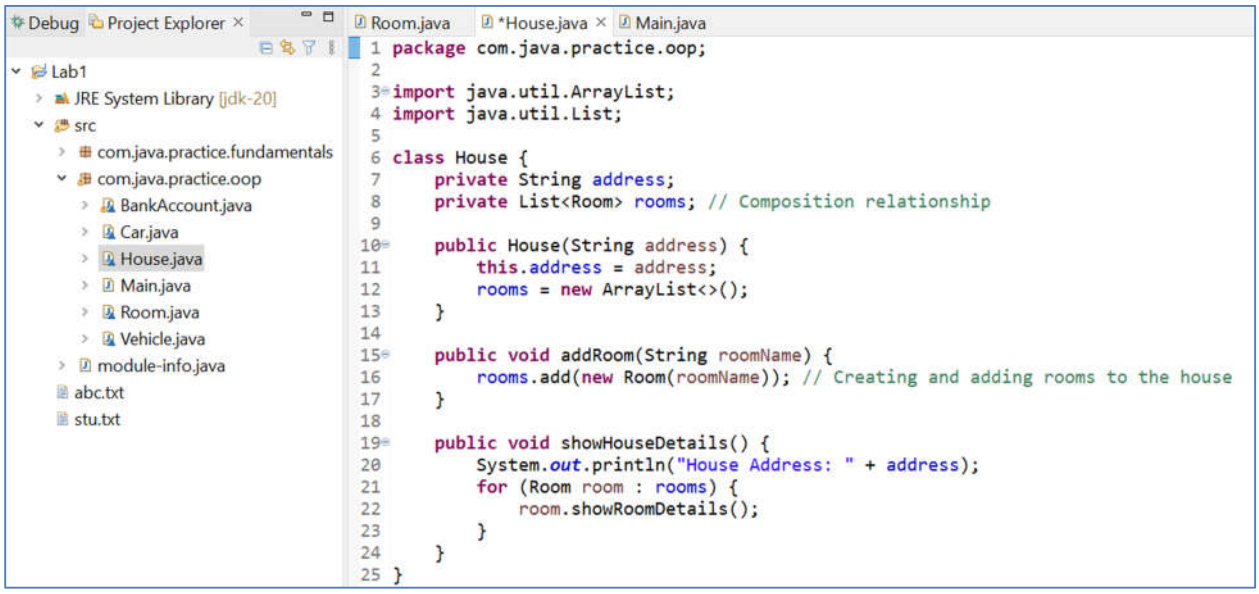
Composition

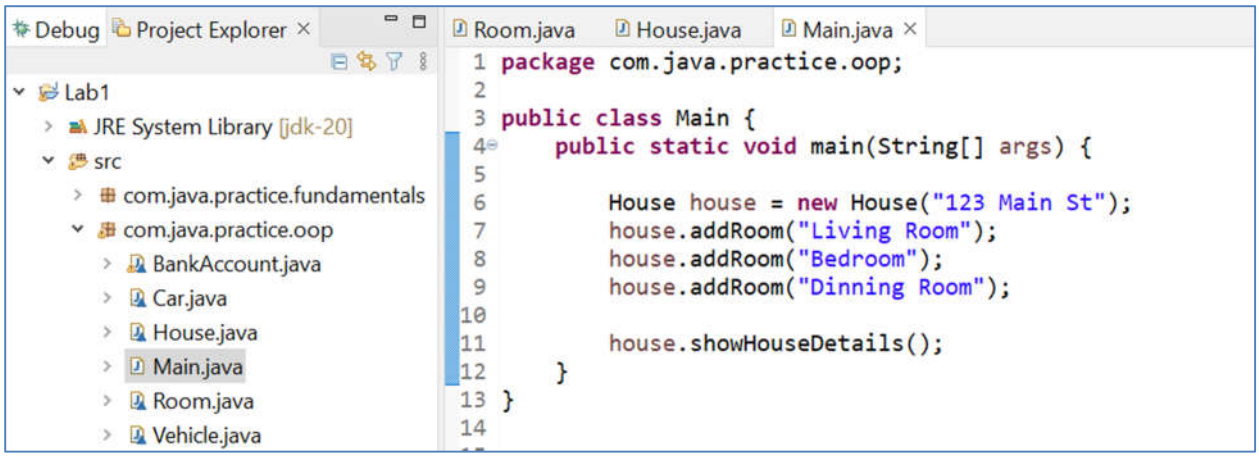
Composition is a strong association where the contained object cannot exist without the parent object. If the parent is destroyed, the child is destroyed as well.

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.







Task

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

Implement the following class diagram with the concepts of OOP