Answer the following

1. Define Class and Objects. State Fours Pillars of Object Oriented Programming System.

Class and Objects in Object-Oriented Programming (OOP)

- **Class**: A class is a blueprint or template for creating objects. It defines a set of properties (attributes) and behaviors (methods) that the objects created from the class can have. For example, a Car class may have properties like color, model, and year, and methods like start(), accelerate(), and brake().
- **Object**: An object is an instance of a class. It represents a specific entity created using the blueprint provided by the class. Each object has its own unique values for the properties defined by the class. For example, a car object might be a specific car with color "red", model "Honda", and year "2021".

Four Pillars of Object-Oriented Programming (OOP)

1. **Encapsulation**: Encapsulation is the concept of bundling the data (variables) and methods (functions) that operate on the data into a single unit, called a class. It also involves restricting direct access to some of an object's components (data hiding) by using access modifiers like private, public, and protected. This helps protect the internal state of the object from being modified directly.
2. **Abstraction**: Abstraction focuses on showing only the necessary details while hiding the complex implementation details. It simplifies the interaction between the user and the system by exposing only essential features. For instance, when you drive a car,

you don’t need to know how the engine works; you only need to know how to use the steering wheel and pedals.

3. **Inheritance**: Inheritance allows a new class (called a subclass or derived class) to inherit the properties and methods of an existing class (called a superclass or base class). This promotes code reusability and the creation of hierarchical relationships. For example, a Truck class might inherit from a Vehicle class, gaining all the properties of a vehicle but also having its own additional attributes.
4. **Polymorphism**: Polymorphism allows objects to be treated as instances of their parent class while behaving differently based on the context. It allows methods to be used in different ways. There are two types of polymorphism:
    - **Compile-time (Method Overloading)**: Methods can have the same name but different parameter lists within a class.
    - **Run-time (Method Overriding)**: A subclass can provide a specific implementation of a method that is already defined in its superclass.

These four pillars form the foundation of object-oriented programming, enabling the creation of modular, maintainable, and flexible software systems.

2. Explain Constructor and types of Constructors

Constructor in Object-Oriented Programming (OOP)

A **constructor** is a special type of method in a class that is automatically called when an object is created. Its primary purpose is to initialize the object, typically by assigning values to the object's attributes. A constructor has the same

name as the class and does not have a return type (not even void).

Types of Constructors

1. **Default Constructor**
    - A default constructor is a constructor that takes no arguments. If you don't define any constructor in your class, most programming languages provide a default constructor automatically.
    - Purpose: Initializes the object with default values.
    - Example:

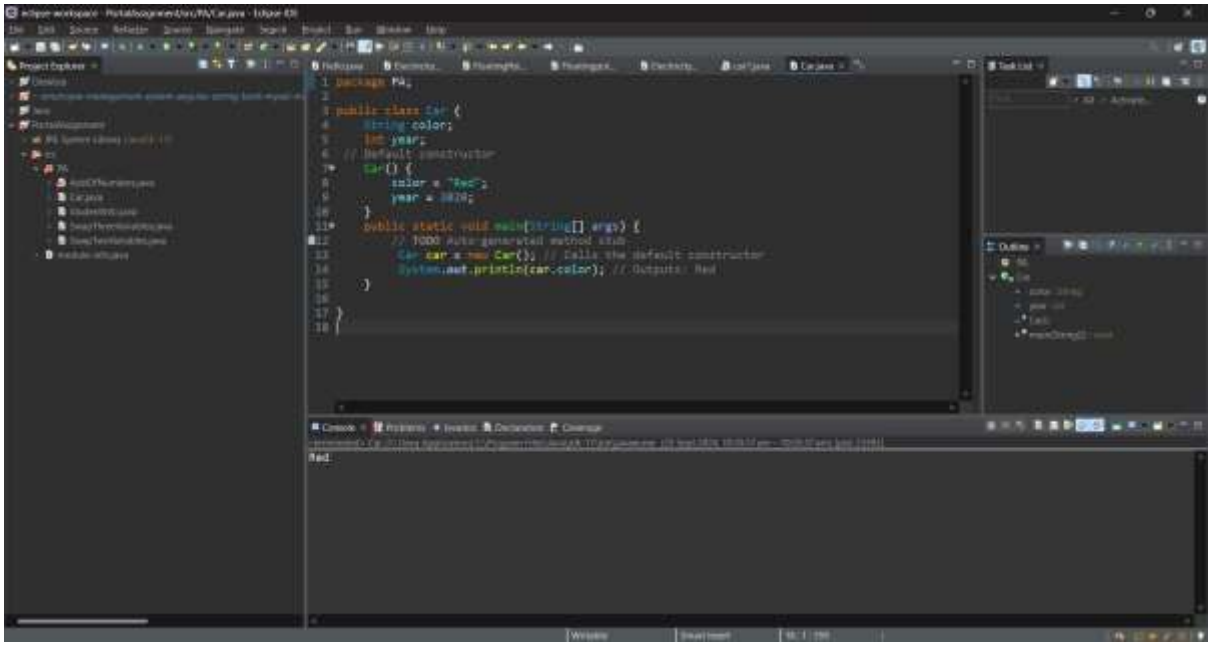
```java
package PA;

public class Car {
     String color;
    int year;
    // Default constructor
    Car() {
        color = "Red";
        year = 2020;
    }


     public static void main(String[] args) {
          // TODO Auto-generated method stub
           Car car = new Car(); // Calls the default constructor
          System.out.println(car.color); // Outputs: Red
     }

}
```

Output:-



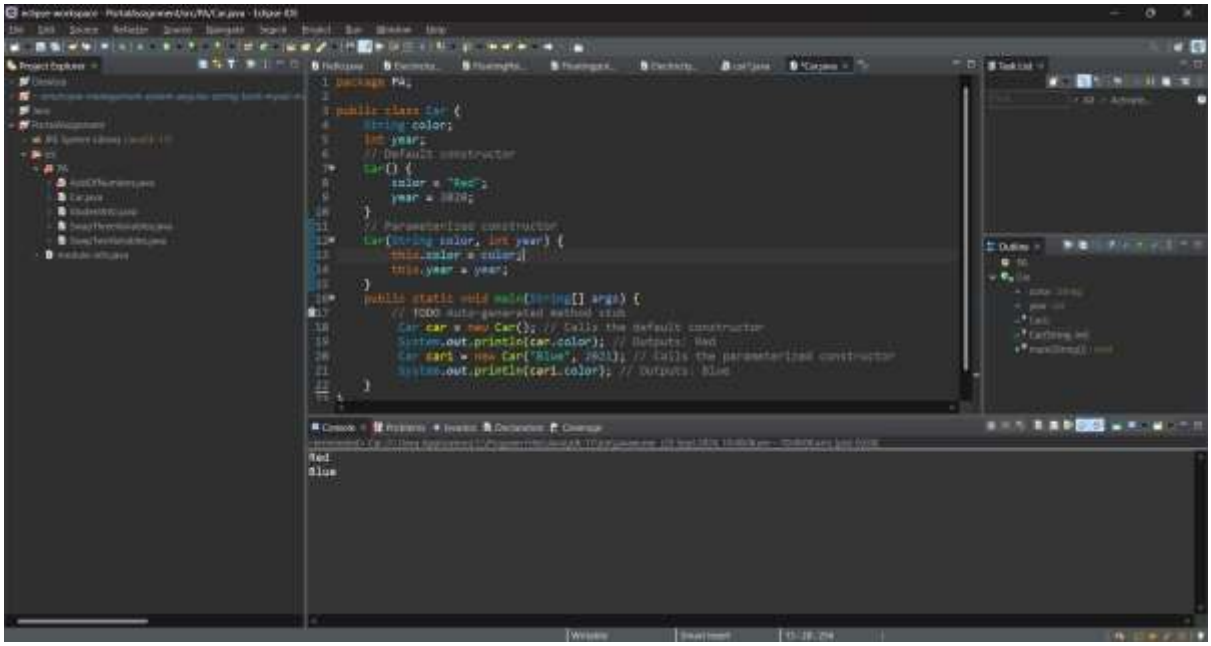
2. **Parameterized Constructor**
    - A parameterized constructor allows you to pass arguments when creating an object, enabling you to initialize the object with custom values.
    - Purpose: To provide specific initial values for the object's attributes during object creation.
    - Example:

```
package PA;

public class Car {
     String color;
    int year;
    // Default constructor
    Car() {
        color = "Red";
        year = 2020;
    }
    // Parameterized constructor
    Car(String color, int year) {
        this.color = color;
        this.year = year;
    }
     public static void main(String[] args) {
           // TODO Auto-generated method stub
            Car car = new Car(); // Calls the default constructor
          System.out.println(car.color); // Outputs: Red
```

```
        Car car1 = new Car("Blue", 2021); // Calls the
parameterized constructor
        System.out.println(car1.color); // Outputs: Blue
    }
}
```

Output:-



3. **Copy Constructor**
    - A copy constructor is used to create a new object by copying the attributes of an existing object.
    - Purpose: To create a copy of an object with the same values as another object.
    - Example:

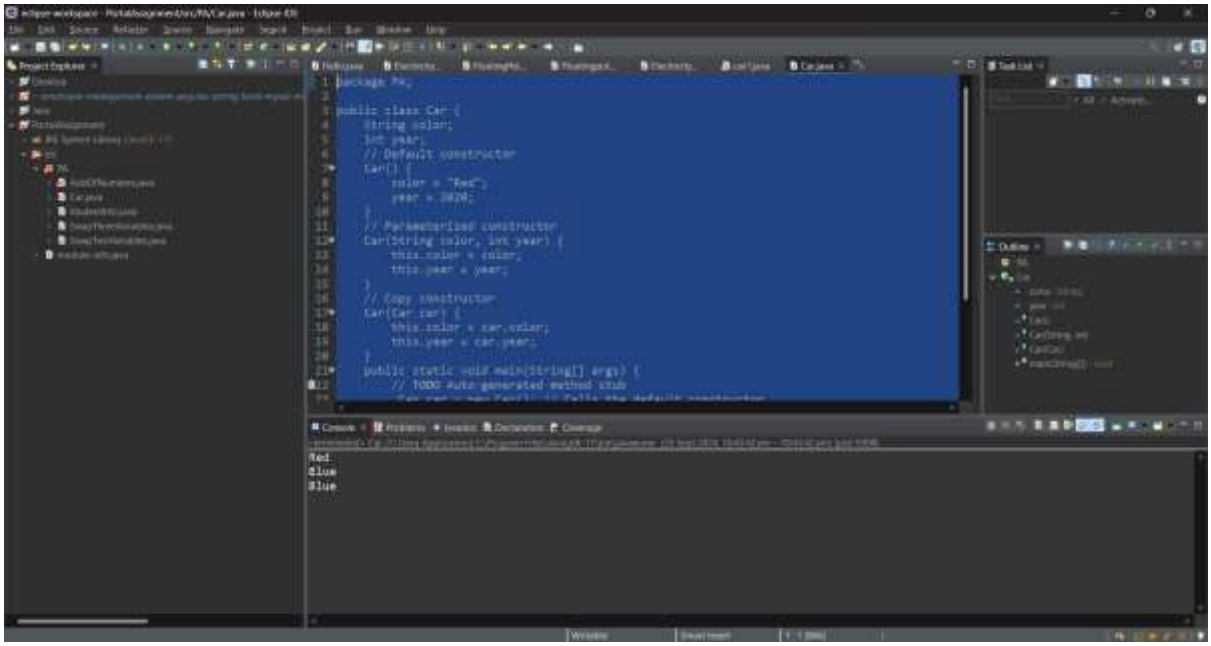
```
package PA;

public class Car {
     String color;
    int year;
    // Default constructor
    Car() {
        color = "Red";
        year = 2020;
    }
    // Parameterized constructor
    Car(String color, int year) {
        this.color = color;
```

```
        this.year = year;
    }
    // Copy constructor
    Car(Car car) {
        this.color = car.color;
        this.year = car.year;
    }
     public static void main(String[] args) {
          // TODO Auto-generated method stub
           Car car = new Car(); // Calls the default constructor
          System.out.println(car.color); // Outputs: Red
          Car car1 = new Car("Blue", 2021); // Calls the
parameterized constructor
          System.out.println(car1.color); // Outputs: Blue
          Car car2 = new Car(car1); // Calls the copy constructor
          System.out.println(car2.color); // Outputs: Green

     }
}
```

Output:-



Key Points about Constructors:

- A constructor is automatically invoked when an object is created.
- Constructors do not have a return type.

- If no constructor is defined, a default constructor is provided by the compiler.
- A class can have multiple constructors (constructor overloading) to support different initialization scenarios.

3. What is Method overloading and state different ways of overloading a methods in Java.

Method Overloading in Java

Method Overloading is a feature in Java that allows a class to have more than one method with the same name but different parameters. This is known as **compile-time polymorphism** because the method that will be called is determined at compile time based on the method signature.

The primary purpose of method overloading is to increase the readability and flexibility of the program by allowing methods to perform similar operations but with different types or numbers of inputs.

Different Ways to Overload Methods in Java

There are two main ways to overload methods in Java:

1. **By Changing the Number of Parameters**
    - You can overload methods by defining multiple methods with the same name but different numbers of parameters. This allows the method to accept varying amounts of input.
    - Example:

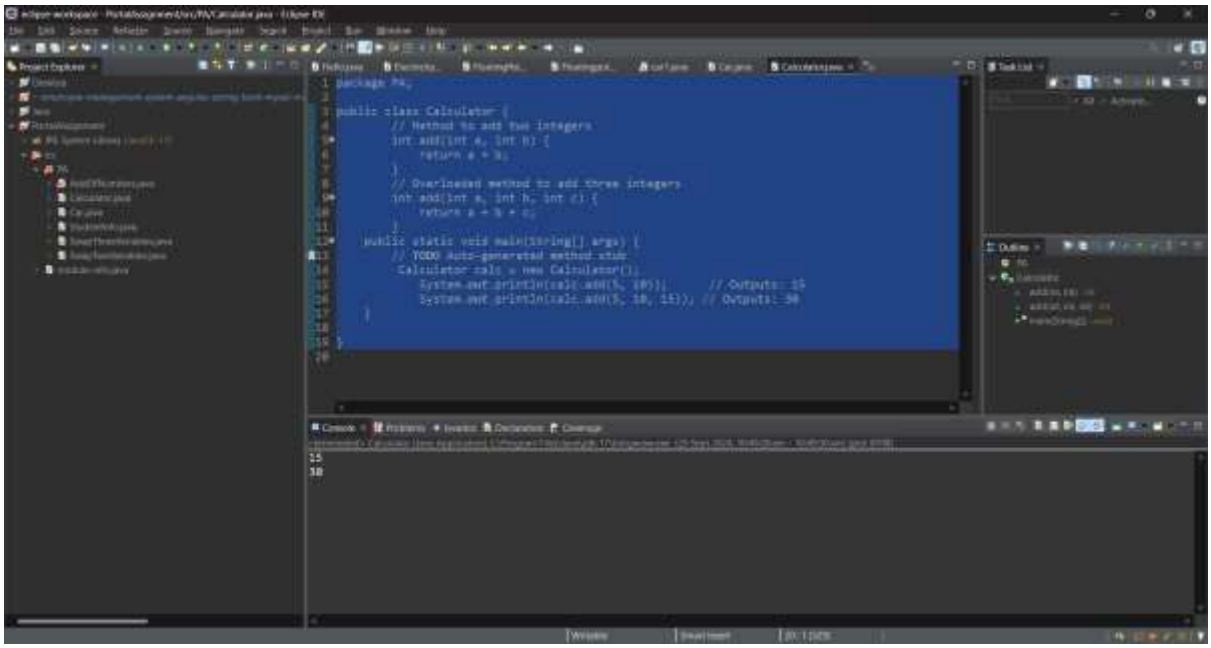
```
package PA;

public class Calculator {
        // Method to add two integers
        int add(int a, int b) {
            return a + b;
```

```
        }
        // Overloaded method to add three integers
        int add(int a, int b, int c) {
            return a + b + c;
        }
    public static void main(String[] args) {
         // TODO Auto-generated method stub
         Calculator calc = new Calculator();
          System.out.println(calc.add(5, 10));       // Outputs: 15
          System.out.println(calc.add(5, 10, 15)); // Outputs: 30
    }

}
```

Output:-



2. **By Changing the Data Type of Parameters**
    - You can also overload methods by changing the data types of the parameters. This enables methods with the same name to handle different types of data.
    - Example:

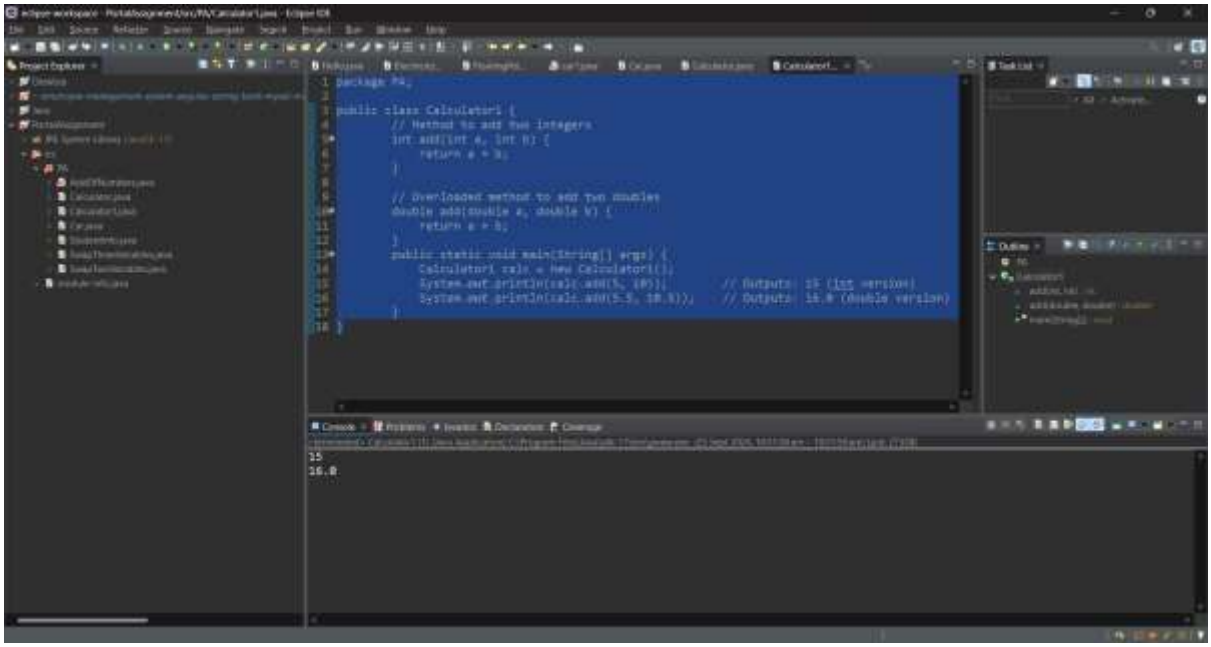
```
package PA;

public class Calculator1 {
        // Method to add two integers
        int add(int a, int b) {
```

```
            return a + b;
        }

        // Overloaded method to add two doubles
        double add(double a, double b) {
            return a + b;
        }
        public static void main(String[] args) {
            Calculator1 calc = new Calculator1();
            System.out.println(calc.add(5, 10));         //
Outputs: 15 (int version)
            System.out.println(calc.add(5.5, 10.5));     //
Outputs: 16.0 (double version)
        }
}
```

Output:-



Other Considerations for Overloading:

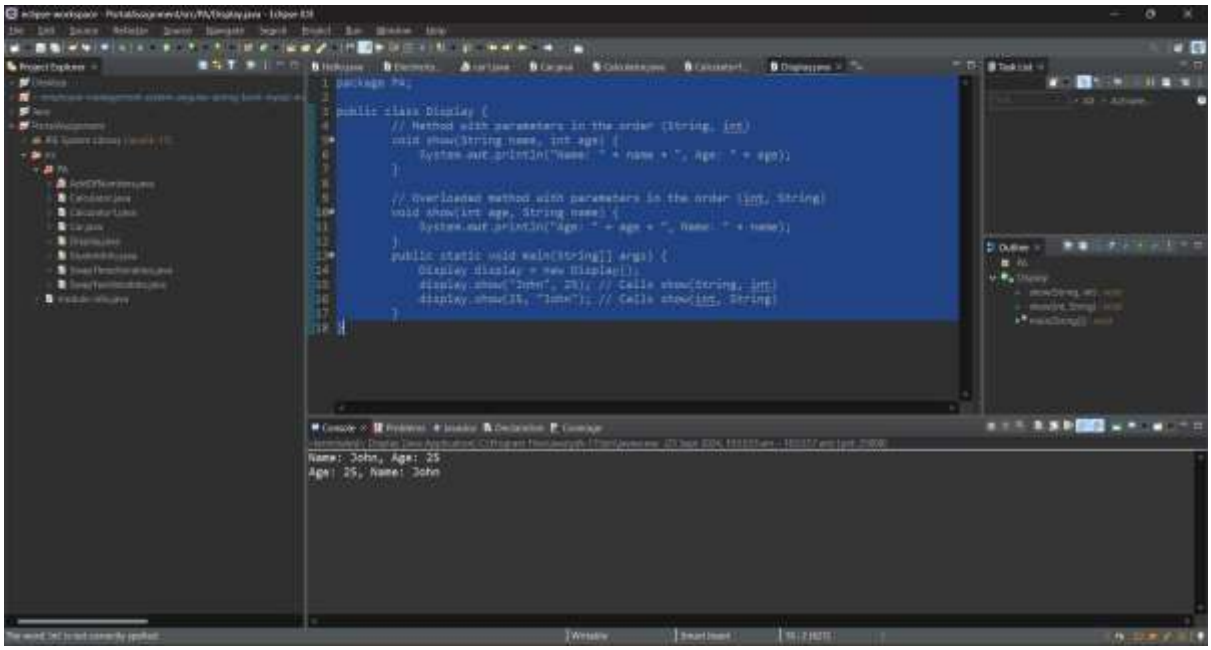
3. **By Changing the Order of Parameters**
    - Overloading can also occur by changing the sequence of parameters if they are of different types.
    - Example:

```
package PA;
```

```java
public class Display {
        // Method with parameters in the order (String, int)
        void show(String name, int age) {
            System.out.println("Name: " + name + ", Age: " + age);
        }

        // Overloaded method with parameters in the order (int,
String)
        void show(int age, String name) {
            System.out.println("Age: " + age + ", Name: " + name);
        }
        public static void main(String[] args) {
            Display display = new Display();
            display.show("John", 25); // Calls show(String, int)
            display.show(25, "John"); // Calls show(int, String)
        }
}
```

Output:-



Important Points to Remember:

- **Return type doesn't matter**: You cannot overload methods solely based on return type. The parameter list must differ in some way.
    - This would cause a compile-time error:

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

double add(int a, int b) { return a + b; } // Error: Duplicate method

- **Access modifiers do not affect overloading**: Changing the access modifier (like public, private, protected) does not count as overloading.
- **Static and instance methods**: Both static and non-static methods can be overloaded, but overloading occurs based on the method's signature (i.e., the parameters), not on whether the method is static or not.

By providing these different ways of overloading methods, Java allows for more intuitive and readable code, where the method name stays consistent, but the parameters can vary based on what is needed.