Unit 2 java

11)a). What is inheritance in object-oriented programming? Discuss its significance.b). Define i. Class, ii. Superclass, iii. Subclass,

iv. Reusability and write the benefits of inheritance?

a) What is Inheritance in Object-Oriented Programming?

Inheritance is a fundamental concept in object-oriented programming (OOP) where a class (known as a **subclass** or **derived class**) inherits attributes and behaviors (methods) from another class (known as a **superclass** or **base class**). This allows the subclass to reuse the code defined in the superclass and extend its functionality.

Significance of Inheritance:

1. **Code Reusability:** Inheritance promotes code reuse by allowing subclasses to inherit methods and attributes from their superclasses. This reduces redundancy and makes the codebase more maintainable.
2. **Hierarchy and Organization:** Inheritance allows classes to be organized into hierarchical structures based on their relationships. This enhances code organization and makes it easier to understand and navigate.
3. **Polymorphism:** Inheritance facilitates polymorphism, which allows objects of different subclasses to be treated as objects of their superclass. This simplifies the handling of objects and promotes flexibility in programming.
4. **Modularity:** Inheritance promotes modular design by separating concerns into distinct classes and leveraging relationships between them.
5. **Easier Maintenance:** Changes made to the superclass automatically propagate to all subclasses. This reduces the effort required for maintenance and ensures consistency in behavior across related classes.

b) Definitions and Benefits of Inheritance

i. Class

A **class** in OOP is a blueprint or template for creating objects that define its properties (attributes) and behaviors (methods).

ii. Superclass (Base Class)

A **superclass** or **base class** is a class that is extended by another class (subclass). It defines common attributes and behaviors that subclasses inherit.

iii. Subclass (Derived Class)

A **subclass** or **derived class** is a class that inherits attributes and behaviors from its superclass. It can also add new attributes or methods and override existing methods of its superclass.

iv. Reusability

Reusability in OOP refers to the capability of using existing code (such as classes and their functionalities) to create new objects or classes. Inheritance enhances reusability by allowing subclasses to inherit and reuse code from their superclasses.

Benefits of Inheritance

1. **Code Reuse:** Inheritance promotes reuse of existing code, reducing redundancy and improving maintainability.
2. **Modularity:** Classes can be organized into hierarchical structures, making the codebase more modular and easier to manage.
3. **Extensibility:** Subclasses can extend the functionality of their superclasses by adding new methods or attributes.
4. **Polymorphism:** Subclasses can be treated as objects of their superclass, allowing for flexibility in designing and using classes.
5. **Easier Maintenance:** Changes made to the superclass automatically propagate to subclasses, ensuring consistency and reducing maintenance effort.
6. **Promotes Hierarchical Relationships:** Inheritance supports the creation of hierarchical relationships between classes, reflecting real-world entities and enhancing the conceptual integrity of the software design.

12)Define inheritance and explain the different types of inheritance (single, multilevel, hierarchical) with example programs?

Inheritance in Object-Oriented Programming

Inheritance is a fundamental concept in object-oriented programming (OOP) where a class (known as a **subclass** or **derived class**) inherits attributes and behaviors (methods) from another class (known as a **superclass** or **base class**). This allows the subclass to reuse the code defined in the superclass and extend its functionality.

Types of Inheritance

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**

1. Single Inheritance

Single inheritance involves one subclass inheriting from only one superclass. It forms a single chain of inheritance.

Example:

```
// Superclass
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
```

```
}

// Subclass inheriting from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

public class SingleInheritanceExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited method from Animal class
        dog.bark(); // Method from Dog class
    }
}
```

Explanation:

- `Animal` is the superclass with an `eat()` method.
- `Dog` is the subclass that inherits `eat()` from `Animal` and adds its own method `bark()`.
- In `main()`, we create an instance of `Dog` and demonstrate calling both inherited and subclass-specific methods.

2. Multilevel Inheritance

Multilevel inheritance involves a chain of inheritance where one subclass inherits from another subclass. It forms a hierarchical structure.

Example:

```
// Superclass
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

// Subclass inheriting from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

// Subclass inheriting from Dog
class Labrador extends Dog {
    void color() {
        System.out.println("Labrador is brown in color");
    }
}

public class MultilevelInheritanceExample {
    public static void main(String[] args) {
        Labrador labrador = new Labrador();
        labrador.eat();   // Inherited method from Animal class
        labrador.bark();  // Inherited method from Dog class
        labrador.color(); // Method from Labrador class
```

```
    }
}
```

Explanation:

- `Animal` is the superclass with an `eat()` method.
- `Dog` is a subclass inheriting from `Animal` and adds its own method `bark()`.
- `Labrador` is a subclass inheriting from `Dog` and adds its own method `color()`.
- In `main()`, we create an instance of `Labrador` and demonstrate calling methods from all levels of the inheritance chain.

3. Hierarchical Inheritance

Hierarchical inheritance involves multiple subclasses inheriting from the same superclass. It forms a tree-like structure.

Example:

```
// Superclass
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

// Subclass inheriting from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

// Another subclass inheriting from Animal
class Cat extends Animal {
    void meow() {
        System.out.println("Cat is meowing");
    }
}

public class HierarchicalInheritanceExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();

        dog.eat();  // Inherited method from Animal class
        dog.bark(); // Method from Dog class

        cat.eat();  // Inherited method from Animal class
        cat.meow(); // Method from Cat class
    }
}
```

Explanation:

- `Animal` is the superclass with an `eat()` method.
- `Dog` and `Cat` are subclasses inheriting from `Animal`.

- Both `Dog` and `Cat` have their own methods (`bark()` and `meow()` respectively).
- In `main()`, we create instances of `Dog` and `Cat` to demonstrate calling methods from their respective classes and inherited methods from `Animal`.

13)Write a java code to create a chain of new classes that inherits from their previous classes (Multilevel inheritance), adding specific functionalities.

```
// Superclass
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

// Subclass inheriting from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

// Subclass inheriting from Dog
class Labrador extends Dog {
    void color() {
        System.out.println("Labrador is brown in color");
    }
}

// Subclass inheriting from Labrador
class LabradorPuppy extends Labrador {
    void play() {
        System.out.println("Labrador puppy is playing");
    }
}

public class MultilevelInheritanceChain {
    public static void main(String[] args) {
        LabradorPuppy puppy = new LabradorPuppy();

        puppy.eat();   // Inherited method from Animal class
        puppy.bark();  // Inherited method from Dog class
        puppy.color(); // Inherited method from Labrador class
        puppy.play();  // Method from LabradorPuppy class
    }
}
```

Explanation:

1. **Animal Class:**
    - `Animal` is the superclass with an `eat()` method that prints "Animal is eating".
2. **Dog Class:**
    - `Dog` inherits from `Animal` and adds a `bark()` method that prints "Dog is barking".
3. **Labrador Class:**

    - `Labrador` inherits from `Dog` and adds a `color()` method that prints "Labrador is brown in color".
4. **LabradorPuppy Class:**
    - `LabradorPuppy` inherits from `Labrador` and adds a `play()` method that prints "Labrador puppy is playing".
5. **Main Method (MultilevelInheritanceChain):**
    - In `main()`, we create an instance of `LabradorPuppy` and demonstrate calling methods from all levels of the inheritance chain (`eat()`, `bark()`, `color()`, and `play()`).

Output:

```
Animal is eating
Dog is barking
Labrador is brown in color
Labrador puppy is playing
```

In this example:

- `LabradorPuppy` inherits `eat()` from `Animal`, `bark()` from `Dog`, and `color()` from `Labrador`.
- It adds its own method `play()` specific to `LabradorPuppy`.

14)Explain the usage of the super keyword in inheritance, elucidating its role in facilitating access to superclass members and constructors from subclass contexts.

Usage of the `super` Keyword in Inheritance

In Java, the `super` keyword is used in the context of inheritance to refer to the immediate superclass of a subclass. It can be used to access superclass methods, constructors, and instance variables from within the subclass. The `super` keyword helps in differentiating between superclass and subclass methods or variables that have the same name.

Role of `super` Keyword in Facilitating Access:

1. **Accessing Superclass Members:**
    - **Methods:** You can invoke superclass methods using `super.methodName()`. This is useful when the subclass overrides a method but still wants to call the superclass implementation.
    - **Variables:** You can access superclass instance variables using `super.variableName`. This allows subclasses to refer to superclass state directly.
2. **Calling Superclass Constructors:**
    - **Constructor Invocation:** In a subclass constructor, `super()` is used to explicitly call a superclass constructor. This is essential when the superclass constructor requires parameters or initialization that the subclass needs to pass or invoke.

Examples of `super` Keyword Usage:

Accessing Superclass Members:

```
// Superclass
class Animal {
    String name = "Animal";

    void eat() {
        System.out.println("Animal is eating");
    }
}

// Subclass inheriting from Animal
class Dog extends Animal {
    String name = "Dog";

    void display() {
        // Access superclass method
        super.eat();

        // Access superclass variable
        System.out.println("Name from superclass: " + super.name);

        // Access subclass variable
        System.out.println("Name from subclass: " + this.name);
    }
}

public class SuperKeywordExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.display();
    }
}
```

Output:

```
Animal is eating
Name from superclass: Animal
Name from subclass: Dog
```

Calling Superclass Constructor:

```
// Superclass
class Animal {
    String type;

    // Constructor with parameter
    Animal(String type) {
        this.type = type;
    }
}

// Subclass inheriting from Animal
class Dog extends Animal {
    String breed;

    // Subclass constructor calling superclass constructor
```

```
    Dog(String type, String breed) {
        super(type);  // Call to superclass constructor
        this.breed = breed;
    }

    void display() {
        System.out.println("Type: " + type);
        System.out.println("Breed: " + breed);
    }
}

public class SuperConstructorExample {
    public static void main(String[] args) {
        Dog dog = new Dog("Mammal", "Labrador");
        dog.display();
    }
}
```

Output:

```
Type: Mammal
Breed: Labrador
```

Key Points:

- The `super` keyword allows subclasses to access and invoke superclass methods, variables, and constructors.
- It is particularly useful in scenarios where the subclass overrides superclass methods but still needs to invoke the superclass implementation.
- When invoking constructors, `super()` must be the first statement in the subclass constructor to ensure superclass initialization occurs before subclass initialization.

15)Discuss the concept of preventing inheritance in Java programming, focusing on its significance, mechanisms, and potential use cases.

Preventing Inheritance in Java

In Java, preventing inheritance is a concept that allows a class to declare that it cannot be subclassed. This means that other classes cannot extend it to create subclasses. Java provides mechanisms to achieve this, primarily through the use of the `final` keyword and private constructors.

Significance of Preventing Inheritance

1. **Security and Stability:** By preventing inheritance, you can control and stabilize the behavior of your classes. This ensures that subclasses cannot modify or extend the functionality in unintended ways, which can lead to security vulnerabilities or instability in the application.
2. **Design Intent:** It communicates design intent clearly. When a class is marked as non-inheritable, it indicates to other developers that the class's design does not support extension, encouraging them to use the class as-is without attempting to subclass it.

3. **Performance Optimization:** Classes marked as `final` can sometimes be optimized by the compiler or runtime environment, knowing that no subclasses will override certain methods or behaviors.

Mechanisms to Prevent Inheritance

1. **Using `final` Keyword:**
   - The `final` keyword can be applied to classes, methods, and variables.
   - When applied to a class (`final class ClassName`), it indicates that the class cannot be subclassed.
   - Example:

```
final class FinalClass {
    // Class definition
}
```

2. **Private Constructors:**
   - By defining all constructors of a class as private, you prevent other classes from extending it because subclasses cannot invoke those constructors.
   - Example:

```
class NonInheritableClass {
    private NonInheritableClass() {
        // Private constructor
    }
}
```

Potential Use Cases

1. **Immutable Classes:** Classes designed to be immutable (unchangeable) are often marked as `final` to prevent unintended modification by subclasses.
2. **Utility Classes:** Utility classes that provide static methods and do not require instantiation can be made `final` to prevent subclassing, ensuring they are used as intended.
3. **Singleton Classes:** Singleton classes often use private constructors and may also be marked as `final` to ensure there is only one instance and prevent inheritance that could lead to multiple instances.
4. **Security-Sensitive Classes:** Classes dealing with sensitive data or security-critical operations may benefit from being `final` to prevent subclasses from overriding methods and compromising security protocols.

Example:

```
final class Singleton {
    private static final Singleton instance = new Singleton();

    private Singleton() {
        // Private constructor
    }

    public static Singleton getInstance() {
        return instance;
    }
```

```
    public void performOperation() {
        // Method implementation
    }
}

// Attempting to extend Singleton class (compile-time error)
// class Subclass extends Singleton {
//     // Subclass definition
// }
```

In the example above, the `Singleton` class is marked as `final`, ensuring that it cannot be extended. This guarantees that only one instance of `Singleton` exists throughout the application, adhering to the singleton design pattern.

16)With a suitable example programs, discuss the importance of final variable, final method and final class to prevent inheritance in Java?

In Java, the `final` keyword is used to restrict the modification or extension of classes, methods, and variables. Let's discuss the importance of `final` variables, `final` methods, and `final` classes in preventing inheritance, along with suitable examples for each.

1. Final Variable

A `final` variable in Java can only be assigned once and cannot be changed thereafter. If applied to an instance variable, it must be initialized either at the time of declaration or in the constructor of the class.

Example:

```
java
Copy code
class Circle {
    final double PI = 3.14; // final variable

    double calculateArea(double radius) {
        return PI * radius * radius;
    }
}
```

Importance:

- **Immutable State:** Final variables ensure that their values cannot be altered once initialized. This guarantees consistency and prevents inadvertent changes.
- **Thread Safety:** Immutable variables are inherently thread-safe because their values cannot be modified concurrently by multiple threads.
- **Design Clarity:** Marking constants or variables that should not change as `final` makes the intent of the code clear to other developers.

2. Final Method

A `final` method in Java cannot be overridden by subclasses. This is useful when you want to ensure that a method's implementation remains unchanged in all subclasses.

Example:

```
class Animal {
    final void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    // Compiler error: Cannot override the final method from Animal
    // void makeSound() {
    //     System.out.println("Dog barks");
    // }
}
```

Importance:

- **Method Immunity:** Final methods cannot be overridden, ensuring that their behavior remains consistent across all subclasses.
- **Security and Stability:** Prevents subclasses from modifying critical behaviors defined in the superclass.
- **Performance Optimization:** The compiler or runtime environment may optimize calls to `final` methods, knowing that they cannot change dynamically.

3. Final Class

A `final` class in Java cannot be subclassed. This is useful when you want to prevent other classes from extending a class and potentially modifying its behavior.

Example:

```
final class Singleton {
    private static final Singleton instance = new Singleton();

    private Singleton() {
        // Private constructor
    }

    public static Singleton getInstance() {
        return instance;
    }

    public void performOperation() {
        System.out.println("Singleton instance performing operation");
    }
}

// Compiler error: Cannot inherit from final Singleton
// class Subclass extends Singleton {
//     // Subclass definition
// }
```

Importance:

- **Immutable Class:** Marking a class as `final` ensures that its implementation cannot be changed by subclassing.
- **Singleton Pattern:** Final classes are commonly used for singleton implementations, ensuring only one instance exists throughout the application.
- **Security and Design:** Final classes communicate that the class design is complete and should not be extended, enhancing code security and stability.

17)Explain the concept of compile time polymorphism (method overloading). Elaborate its types with illustrative examples and discuss its advantages.

Compile-Time Polymorphism (Method Overloading)

Compile-time polymorphism, also known as **method overloading**, is a feature in Java where multiple methods can have the same name but different parameters. The decision on which method to call is made at compile time based on the number of parameters, types of parameters, and their order.

Types of Method Overloading

1. **Overloading by Number of Parameters**
2. **Overloading by Data Type of Parameters**
3. **Overloading by Sequence of Data Type of Parameters**

1. Overloading by Number of Parameters

In this type, methods have the same name but a different number of parameters.

Example:

```
class Calculator {
    // Method with two int parameters
    void add(int a, int b) {
        System.out.println("Sum: " + (a + b));
    }

    // Method with three int parameters
    void add(int a, int b, int c) {
        System.out.println("Sum: " + (a + b + c));
    }
}

public class MethodOverloadingExample1 {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        calc.add(10, 20);        // Calls add(int, int)
        calc.add(10, 20, 30);    // Calls add(int, int, int)
    }
}
```

Advantages:

- **Code Readability:** Methods performing similar operations can be grouped under the same name, making the code more readable.
- **Reduced Complexity:** Avoids the need for multiple method names for performing similar tasks with different parameters.
- **Flexibility:** Provides flexibility to call methods with different parameter configurations based on requirements.

2. Overloading by Data Type of Parameters

In this type, methods have the same name but parameters differ in type.

Example:

```
class Display {
    // Method with int parameter
    void show(int num) {
        System.out.println("Integer: " + num);
    }

    // Method with double parameter
    void show(double num) {
        System.out.println("Double: " + num);
    }
}

public class MethodOverloadingExample2 {
    public static void main(String[] args) {
        Display display = new Display();

        display.show(10);        // Calls show(int)
        display.show(10.5);      // Calls show(double)
    }
}
```

Advantages:

- **Code Reusability:** Methods with the same functionality but different parameter types can be reused.
- **Promotes Method Naming Consistency:** Allows methods performing similar tasks to have consistent names, improving code maintainability.

3. Overloading by Sequence of Data Type of Parameters

In this type, methods have the same name and type, but parameters differ in sequence.

Example:

```
class Sequence {
    // Method with two int parameters
    void print(int a, int b) {
        System.out.println("Integers: " + a + ", " + b);
    }
```

```
    // Method with int followed by double parameters
    void print(int a, double b) {
        System.out.println("Integer and Double: " + a + ", " + b);
    }
}

public class MethodOverloadingExample3 {
    public static void main(String[] args) {
        Sequence sequence = new Sequence();

        sequence.print(10, 20);         // Calls print(int, int)
        sequence.print(10, 20.5);       // Calls print(int, double)
    }
}
```

Advantages:

- **Improved Code Flexibility:** Allows methods to handle different parameter sequences effectively.
- **Simplifies Method Naming:** Enables methods performing similar operations with variations in parameter order to have the same name.

18)a). Define the concept of run time polymorphism (method overriding).

b). Write a java program to get interests from different banks using the Hierarchical inheritance and run time polymorphism.

Run-Time Polymorphism (Method Overriding)

Run-time polymorphism, also known as **method overriding**, is a feature in Java that allows a subclass to provide a specific implementation of a method that is already provided by its superclass. Method overriding is achieved when a subclass provides a specific implementation of a method that is already defined in its superclass. This allows a subclass to provide a specific implementation of a method that is already defined in its superclass.

Key Points of Method Overriding:

1. **Method Signature:** The method in the subclass must have the same signature (method name, parameters, and return type) as the method in the superclass.
2. **Dynamic Binding:** The decision on which method to call is made at runtime based on the object type, not the reference type.
3. **@Override Annotation:** It is good practice to use the `@Override` annotation when overriding a method to ensure that you are actually overriding a method from the superclass.

Example Program Using Hierarchical Inheritance and Method Overriding

Here's a Java program demonstrating hierarchical inheritance and run-time polymorphism through method overriding:

```
// Superclass Bank
```

```
class Bank {
    double getInterestRate() {
        return 0.0;
    }
}

// Subclass HDFC extending Bank
class HDFC extends Bank {
    @Override
    double getInterestRate() {
        return 6.5; // HDFC bank interest rate
    }
}

// Subclass ICICI extending Bank
class ICICI extends Bank {
    @Override
    double getInterestRate() {
        return 7.0; // ICICI bank interest rate
    }
}

// Subclass SBI extending Bank
class SBI extends Bank {
    @Override
    double getInterestRate() {
        return 6.0; // SBI bank interest rate
    }
}

public class BankInterest {
    public static void main(String[] args) {
        HDFC hdfc = new HDFC();
        ICICI icici = new ICICI();
        SBI sbi = new SBI();

        // Polymorphic behavior
        Bank[] banks = { hdfc, icici, sbi };

        // Display interest rates using polymorphism
        for (Bank bank : banks) {
            System.out.println("Interest rate of " +
bank.getClass().getSimpleName() + ": " + bank.getInterestRate() + "%");
        }
    }
}
```

Explanation:

1. **Bank Class:** Defines a superclass `Bank` with a method `getInterestRate()` returning 0.0 (to be overridden).
2. **HDFC, ICICI, SBI Classes:** Subclasses extending `Bank`, each overriding `getInterestRate()` method with bank-specific interest rates.
3. **BankInterest Class:** Main class demonstrating run-time polymorphism:
    - Creates instances of `HDFC`, `ICICI`, and `SBI`.
    - Stores them in an array of type `Bank`.
    - Iterates through the array and invokes `getInterestRate()` method on each object, demonstrating polymorphic behavior.

Output:

```
Interest rate of HDFC: 6.5%
Interest rate of ICICI: 7.0%
Interest rate of SBI: 6.0%
```

Advantages of Method Overriding:

- **Flexibility:** Allows subclasses to provide specialized implementations of methods defined in their superclass, catering to specific requirements.
- **Code Reusability:** Promotes reusability of code by inheriting and modifying existing behavior as needed.
- **Dynamic Dispatch:** Enables dynamic binding of method calls at runtime, facilitating polymorphic behavior and flexibility in object-oriented design.

19)Discuss the role of abstract classes and methods in achieving polymorphism, with an example code.

Role of Abstract Classes and Methods in Achieving Polymorphism

In Java, **abstract classes** and **abstract methods** play a crucial role in achieving polymorphism by allowing for the definition of common behaviors and enforcing subclasses to provide specific implementations. Let's delve into their roles with an example code.

Abstract Classes

An abstract class in Java is a class that cannot be instantiated on its own but can have abstract methods, concrete methods, or both. Abstract classes serve as templates for subclasses, providing a common interface and shared functionality.

Abstract Methods

An abstract method is a method declared without an implementation in an abstract class. Subclasses must provide concrete implementations for all abstract methods defined in their abstract superclass.

Example: Abstract Class and Polymorphism

Let's consider an example involving an abstract class `Shape` with abstract method `calculateArea()`, and subclasses `Circle` and `Rectangle` providing specific implementations of `calculateArea()`.

```
// Abstract class Shape
abstract class Shape {
    abstract double calculateArea(); // Abstract method
}

// Subclass Circle extending Shape
class Circle extends Shape {
    private double radius;
```

```
    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double calculateArea() {
        return Math.PI * radius * radius;
    }
}

// Subclass Rectangle extending Shape
class Rectangle extends Shape {
    private double length;
    private double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    double calculateArea() {
        return length * width;
    }
}

public class ShapeDemo {
    public static void main(String[] args) {
        Shape shape1 = new Circle(5); // Polymorphic behavior
        Shape shape2 = new Rectangle(4, 6); // Polymorphic behavior

        // Calculate and display areas
        System.out.println("Area of Circle: " + shape1.calculateArea());
        System.out.println("Area of Rectangle: " + shape2.calculateArea());
    }
}
```

Explanation:

1. **Shape Class:** Abstract class defining `calculateArea()` as an abstract method, intended to be implemented by subclasses.
2. **Circle and Rectangle Classes:** Concrete subclasses extending `Shape` and providing specific implementations of `calculateArea()` based on their respective formulas.
3. **ShapeDemo Class:** Demonstrates polymorphic behavior by creating instances of `Circle` and `Rectangle` as `Shape` references. This allows treating objects of different subclasses uniformly through the `Shape` interface.

Output:

```
Area of Circle: 78.53981633974483
Area of Rectangle: 24.0
```

Advantages of Abstract Classes and Methods for Polymorphism:

- **Code Organization:** Abstract classes define a contract for subclasses, promoting code consistency and organization by enforcing method implementations.

- **Flexibility and Extensibility:** Enables polymorphic behavior, where different subclasses can be treated interchangeably through a common superclass interface (`Shape` in this case).
- **Encapsulation of Common Behavior:** Abstract classes encapsulate common behavior and attributes, allowing subclasses to inherit and extend functionality while maintaining a cohesive design.

20)Write a java program for Abstract class having Constructor, Data Member and Methods?

```
// Abstract class Shape
abstract class Shape {
    protected String name;
    protected double area;

    // Constructor
    public Shape(String name) {
        this.name = name;
        this.area = 0.0; // Default area
    }

    // Abstract method
    abstract void calculateArea();

    // Method to display shape details
    void display() {
        System.out.println("Shape: " + name);
        System.out.println("Area: " + area);
    }
}

// Concrete subclass Circle
class Circle extends Shape {
    private double radius;

    // Constructor
    public Circle(String name, double radius) {
        super(name);
        this.radius = radius;
    }

    // Override abstract method
    @Override
    void calculateArea() {
        area = Math.PI * radius * radius;
    }
}

// Concrete subclass Rectangle
class Rectangle extends Shape {
    private double length;
    private double width;

    // Constructor
    public Rectangle(String name, double length, double width) {
        super(name);
        this.length = length;
        this.width = width;
    }
```

```
    // Override abstract method
    @Override
    void calculateArea() {
        area = length * width;
    }
}

// Main class to test abstract class and subclasses
public class AbstractClassExample {
    public static void main(String[] args) {
        Circle circle = new Circle("Circle", 5.0);
        Rectangle rectangle = new Rectangle("Rectangle", 4.0, 6.0);

        // Calculate areas
        circle.calculateArea();
        rectangle.calculateArea();

        // Display shape details
        circle.display();
        System.out.println(); // Blank line for separation
        rectangle.display();
    }
}
```

Explanation:

1. **Abstract Class `Shape`:**
   - Declares an abstract method `calculateArea()` that subclasses must override.
   - Contains a constructor `Shape(String name)` to initialize the `name` of the shape.
   - Defines a method `display()` to print shape details (`name` and `area`).
2. **Concrete Subclasses (`Circle` and `Rectangle`):**
   - Extend the abstract class `Shape` and provide implementations for `calculateArea()` based on their respective formulas (`Circle`: π * r^2, `Rectangle`: length * width).
   - Constructors (`Circle(String name, double radius)` and `Rectangle(String name, double length, double width)`) initialize their specific data members (`radius` and `length`, `width`) and call the superclass constructor using `super(name)`.
3. **Main Class `AbstractClassExample`:**
   - Creates instances of `Circle` and `Rectangle`.
   - Calls `calculateArea()` for each shape to compute its area.
   - Calls `display()` to print details of each shape (`name` and `area`).

Output:

```
Shape: Circle
Area: 78.53981633974483

Shape: Rectangle
Area: 24.0
```