Abstraction

is one of the four fundamental OOP (Object-Oriented Programming) principles, along with encapsulation, inheritance, and polymorphism. Abstraction involves simplifying complex systems by breaking them into smaller, more manageable parts and focusing on the essential features while hiding unnecessary details.

In Java, abstraction is achieved through abstract classes and interfaces. Abstract classes are classes that cannot be instantiated themselves but can be subclassed, while interfaces define a contract that classes must adhere to. Let's explore abstraction with two code examples—one using an abstract class and the other using an interface.

Example 1: Abstraction using an Abstract Class

```java
// Abstract class representing a shape
abstract class Shape {
    // Abstract method to calculate area (must be implemented by subclasses)
    public abstract double calculateArea();

    // Concrete method
    public void display() {
        System.out.println("This is a shape.");
    }
}

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

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

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

// Concrete subclass representing a rectangle
class Rectangle extends Shape {
    private double length;
    private double width;
```

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

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

public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle(5);
        Rectangle rectangle = new Rectangle(4, 6);

        circle.display();
        System.out.println("Area of Circle: " + circle.calculateArea());

        rectangle.display();
        System.out.println("Area of Rectangle: " + rectangle.calculateArea());
    }
}
```

In this example, we have an abstract class `Shape` with an abstract method `calculateArea()`. Subclasses `Circle` and `Rectangle` extend the `Shape` class and provide concrete implementations for the `calculateArea()` method. The abstract class serves as a blueprint for shapes, and each subclass implements its specific behavior.

Example 2: Abstraction using an Interface

```java
// Interface representing a drawable object
interface Drawable {
    void draw(); // Abstract method to draw the object
}

// Concrete class representing a Circle
class Circle implements Drawable {
    private double radius;

    public Circle(double radius) {
```

```
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a circle with radius " + radius);
    }
}

// Concrete class representing a Rectangle
class Rectangle implements Drawable {
    private double length;
    private double width;

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

    @Override
    public void draw() {
        System.out.println("Drawing a rectangle with length " + length + " and width " + width);
    }
}

public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle(5);
        Rectangle rectangle = new Rectangle(4, 6);

        circle.draw();
        rectangle.draw();
    }
}
```

In this example, we have an interface `Drawable` with an abstract method `draw()`. Both the `Circle` and `Rectangle` classes implement this interface and provide their own implementations for the `draw()` method. This allows objects of different types to be treated uniformly when it comes to drawing them, emphasizing the abstraction of drawing functionality from specific shapes.

These examples illustrate how abstraction helps in simplifying complex systems by defining a common interface or abstract class that hides implementation details, allowing you to work with objects at a higher level of abstraction.