Abstraction, Inheritance and Polymorphism

What is Abstraction?

Abstraction is an OOP principle that focuses on exposing only essential features while hiding implementation details.

Why Abstraction Matters

- Simplifies complex systems
- Reduces code duplication
- Improves maintainability
- Enhances flexibility

Motivation Example

You are implementing a program that manages geometric shapes. Each shape has a method to calculate its area, but the way the area is calculated differs for each shape (e.g., circle, rectangle, triangle).

Be aware that many thing would be wrong with the following example…

Motivation Example II

```java
class Figure {
    String type; // "circle", "rectangle", "triangle"
    double radius; // for circle
    double width, height; // for rectangle
    double base, heightTriangle; // for triangle
}
```

Motivation Example III

```java
for(Figure f: figures) {
    System.out.println("Calculating area of " + f.type);

    if(f.type.equals("circle")) {
        System.out.println("Circle area: " + (Math.PI * f.radius * f.radius));
    } else if(f.type.equals("rectangle")) {
        System.out.println("Rectangle area: " + (f.width * f.height));
    } else if(f.type.equals("triangle")) {
        System.out.println("Triangle area: " + (0.5 * f.base * f.height));
    }
}
```

Problems with the Example

- This code shows **details** of how each shape calculates its area.
- Details of each shape are mixed together, making it hard to read and maintain.
- Details should be hidden when possible.
- In addition, adding a new shape requires modifying existing code.

Key Principles of Abstraction

- Hide unnecessary details
- Expose only relevant operations
- Use interfaces and abstract classes
- An interface is a **contract** that defines methods a class must implement, without specifying how.

Interfaces in Java

- Cannot be instantiated
- Only contains abstract methods, i.e their signatures
- No protection levels for methods (all are public)
    - Remember it's a contract

```java
interface Drawable {
    void draw();
}
```

Example: Interface

```java
interface Shape {
    double area();
}

class Circle implements Shape {
    double radius;
    Circle(double r) { radius = r; }
    double area() { return Math.PI * radius * radius; }
}
```

Example: Interface (II)

```
class Rectangle implements Shape {
    double width, height;
    Rectangle(double w, double h) { width = w; height = h; }
    double area() { return width * height; }
}
```

Example: Interface (III)

Now our motivation example can be rewritten as:

```java
Shape[] shapes = { new Circle(5), new Rectangle(4, 6) };
for (Shape s : shapes) {
    System.out.println("Area: " + s.area());
}
```

Benefits of Abstraction

- Promotes loose coupling
- Facilitates code reuse
- Enables polymorphism

Abstraction vs. Encapsulation

- **Abstraction**: Focuses on what an object does
- **Encapsulation**: Focuses on how it achieves it

Common Mistakes

- Exposing too many details
- Expose only the necessary operations

Inheritance

So, what's inheritance?

Inheritance is an OOP principle that allows a class to inherit properties and methods from another class. This promotes **code reuse** and establishes a natural hierarchy between classes.

Inheritance vs Abstraction

Notice that in our example we have two classes (`Circle` and `Rectangle`) that share a common interface (`Shape`). This is a form of inheritance, as both classes inherit the contract defined by the `Shape` interface.

But typically, inheritance is implemented using classes, not interfaces because we want to **reuse code**. For example:

Common Base Class

```java
class Shape {
    public int x;
    public int y;

    public void moveTo(int newX, int newY) {
        x = newX;
        y = newY;
    }

    public void moveBy(int deltaX, int deltaY) {
        x += deltaX;
        y += deltaY;
    }

    public int[] getCenter() {
        return new int[] { x, y };
    }
}
```

Class extension

```
class Circle extends Shape {
    double radius;
    Circle(double r) { radius = r; }
    double area() { return Math.PI * radius * radius; }
}

class Rectangle extends Shape {
    double width, height;
    Rectangle(double w, double h) { width = w; height = h; }
    double area() { return width * height; }
}
```

`extends` keyword is used to indicate that a class inherits from another class. All the public and protected members of the parent class are available in the child class.

Class extension (II)

In this setup, both `Circle` and `Rectangle` inherit the properties and methods of `Shape`, allowing us to reuse code and maintain a clear structure.

It's like "cutting and pasting" the code from `Shape` into `Circle` and `Rectangle`, but done automatically by the language.

Notice that interfaces cannot provide code reuse, only a contract and classes `implements` them. A class `extends` other classes to reuse code.

Class extension (III)

- A class can implement multiple contracts, i.e interfaces, but can only extend one class (single inheritance).
- At the same time, i.e:

```
class MultiShape extends Shape implements Drawable, Cloneable, Serializable {
    // class body
}
```

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It is used in conjunction with inheritance.

For example, both `Circle` and `Rectangle` can be treated as `Shape` objects.

Polymorphism (II)

```
Shape s1 = new Circle(5);
Shape s2 = new Rectangle(4, 6);

System.out.println("Circle area: " + s1.area());
System.out.println("Rectangle area: " + s2.area());
```

We are storing different types of shapes in variables of type `Shape`. When we call `area()`, the correct method for each shape is invoked.

Polymorphism (II)

```java
Shape[] shapes = { new Circle(5), new Rectangle(4, 6) };

for (Shape s : shapes) {
    System.out.println("Area: " + s.area());
}
```

This is our introduction example again, notice we are already using polymorphism to treat different shapes uniformly.

Best Practices

- Use interfaces for contracts
- Abstract classes for shared code
- Hide implementation details

Summary

- **Abstraction** helps manage complexity and build flexible, maintainable systems. Use abstract classes and interfaces to define clear boundaries.
- **Inheritance** promotes code reuse and establishes relationships between classes.
- **Polymorphism** enables treating different objects uniformly through a common interface.