Encapsulation with Inheritance

Inheritance allows a subclass to inherit properties and behaviors (methods and fields) from a superclass. Encapsulation, on the other hand, is the practice of bundling the data (attributes) and methods that operate on that data into a single unit (class). Combining encapsulation with inheritance helps maintain data integrity and access control in a hierarchical class structure.

Example:

```java
class Person {
    private String name; // Encapsulated attribute

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

class Student extends Person {
    private int studentId;

    public Student(String name, int studentId) {
        super(name);
        this.studentId = studentId;
    }

    public int getStudentId() {
        return studentId;
    }
}
```

In this example, the `Student` class inherits from the `Person` class. The `Person` class encapsulates the `name` attribute, providing a getter method to access it. The `Student` class encapsulates the `studentId` attribute in addition to `name`.

By using inheritance, the `Student` class automatically gains access to the encapsulated `name` attribute through the `Person` superclass. This allows the `Student` class to reuse the `Person` class's code and data without having to reimplement it.

Encapsulation with Polymorphism

Polymorphism is the ability of objects of different classes to be treated as objects of a common superclass. When encapsulation is combined with polymorphism, you can ensure consistent and controlled access to encapsulated data across different types of objects.

Example:

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

class Circle implements Shape {
    private double radius;

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

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

class Rectangle implements 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;
    }
}
```

In this example, the `Shape` interface defines the `calculateArea()` method. The `Circle` and `Rectangle` classes encapsulate their respective attributes (`radius` and `length`, `width`) and implement the `calculateArea()` method.

By using polymorphism, you can create a collection of `Shape` objects and call `calculateArea()` on them without knowing their specific types. This is because the `Shape` interface defines a contract that all implementing classes must adhere to.

For example, the following code demonstrates how to use polymorphism to calculate the area of a circle and a rectangle:

Java
```java
Shape circle = new Circle(5.0);
Shape rectangle = new Rectangle(4.0, 6.0);

System.out.println("Circle Area: " + circle.calculateArea());
System.out.println("Rectangle Area: " + rectangle.calculateArea());
```

In this code, `circle` and `rectangle` are both treated as `Shape` objects. This allows us to call the `calculateArea()` method on both objects, even though they are of different types.

Benefits of Combining Encapsulation with Inheritance and Polymorphism

Combining encapsulation with inheritance and polymorphism provides several benefits, including:

- Data integrity: Encapsulation helps to protect data from unauthorized access and modification. Inheritance allows us to reuse the encapsulated data in subclasses without having to reimplement it. This helps to ensure that the data is always consistent and accurate.
- Code reuse: Inheritance allows us to reuse code and data from superclasses in subclasses. This can save us a significant amount of time and effort, and it can also help to improve the quality of our code.
- Flexibility: Polymorphism allows us to write code that is more flexible and reusable. For example, we can create a collection of objects of different types and then call a common method on all of them without having to worry about their specific types. This can make our code easier to maintain and extend.

Overall, combining encapsulation with inheritance and polymorphism is a powerful design technique that can help us to write more robust, maintainable, and reusable code.