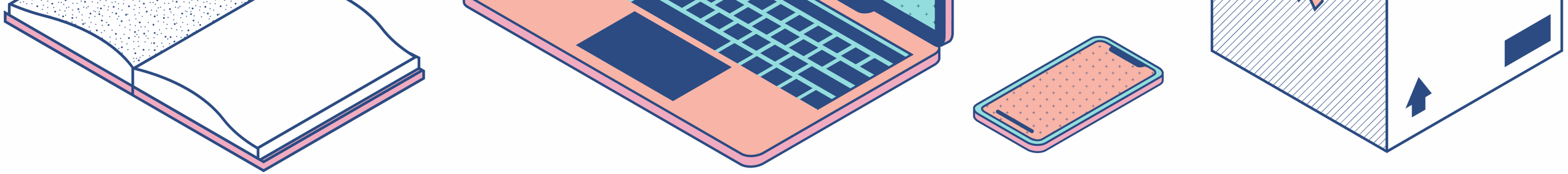
Quiz+

Abstract Class vs Interface

Majd Riyad



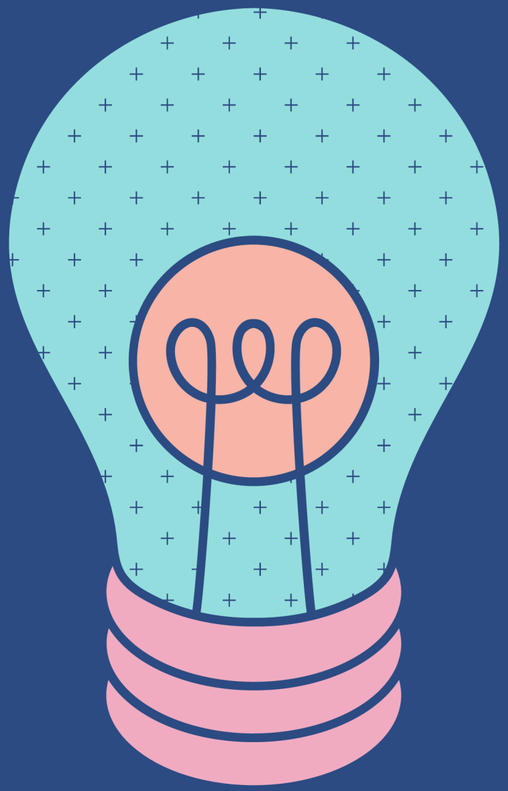
Abstract Class

- **Abstract** and **concrete** methods
- Can have **instance** variables
- **Single** inheritance (can extend one class)
- Can have any **access modifier**
- Can have **constructors**

Interface

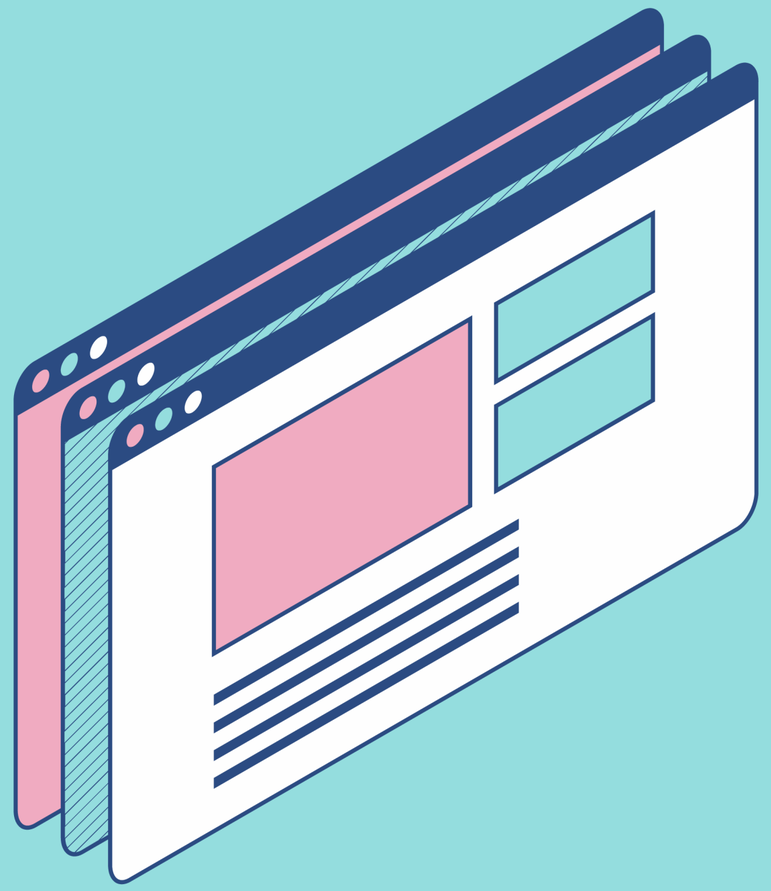
- **Abstract** methods
- Only static **final variables** (constants)
- **Multiple** inheritance (can implement multiple interfaces)
- Methods are **public** by default
- **Cannot** have **constructors**

What do I use, Abstract Class or Interface?



Choosing Between Abstract Class and Interface in Object-Oriented Programming

The choice between an abstract class and an interface in OOP is determined by the specific requirements and design constraints of your application. So the question is which is better for your use case?



Here are some cases in which an abstract class might be a better choice

1. KNOW SOMETHING ABOUT BEHAVIOR
2. TEMPLATE METHOD PATTERN
3. SHARING STATE AND BEHAVIOR
4. NEED TO CONCRETE METHOD

1- Know something about behavior

```java
//1- Example for know something about behavior
abstract class  Shape{  2 usages  1 inheritor
    public abstract void draw();  1 usage  1 implementation

    public void display(){  no usages
        System.out.println("Displaying shape...");
        this.draw();
    }
}
class  Ciecle extends Shape{  3 usages
    @Override  1 usage
    public void draw() {
        System.out.println("Drawing Ciecle...");
    }
}
```

2- Template Method Pattern

```java
//You want to use a method to define the general structure of an algorithm, leaving some steps to subclasses.
abstract class Game{  1 usage  1 inheritor
    public final void play(){ //  A final method cannot be overridden in any subclass.  no usages
        initialize();
        startPlay();
        endPlay();
    }
    public abstract void initialize();  1 usage  1 implementation
    public abstract void startPlay();  1 usage  1 implementation
    public abstract void endPlay();  1 usage  1 implementation
}

class Chess extends Game{  2 usages
    @Override  1 usage
    public void initialize() {
        System.out.println("Initializing Chess");
        System.out.println("********************");
    }

    @Override  1 usage
    public void startPlay() {
        System.out.println("Starting Chess");
        System.out.println("********************");
    }
    public void endPlay() {  1 usage
        System.out.println("Ending Chess");
        System.out.println("********************");
    }
```

3- Sharing State and Behavior && need to concrete method

```java
// 4-  Sharing State and Behavior && need to concrete method
abstract class Employee {  4 usages  2 inheritors
    private String name;  2 usages


    public Employee(){  2 usages
        this.name = "Majd";

    }
    public abstract int getSalary();  1 usage  2 implementations

    public String getName() {  1 usage
        return name;
    }
}
```

```java
class HourlyEmployee extends Employee{  1 usage
    private int hours;  2 usages
    private  int price ;  2 usages
    public HourlyEmployee(){  1 usage
        super();
        hours=4;
        price=50;
    }

    @Override  1 usage
    public int getSalary() {
        return this.hours* this.price;
    }
}
class SalaryEmployee extends Employee{  1 usage
    private int salary;  2 usages
    public SalaryEmployee(){  1 usage
        super();
        salary=500;
    }
    public int getSalary() {  1 usage
        return salary;
    }
}
```

Here are some cases in which an interface might be a better choice

1. KNOW NOTHING ABOUT BEHAVIOR
2. MULTIPLE INHERITANCE
3. UNRELATED CLASSES
4. DEFINING A CONTRACT

1- Know nothing about behavior

```
132     //1- Know nothing about behavior
133
134     interface edible {  2 usages  2 implementations
135         void eat() ;  no usages  2 implementations
136     }
137     class IslamicPerson implements edible{  1 usage
138         @Override  no usages
139         public void eat() {
140             System.out.println("Halal...");
141         }
142     }
143     class ChinesePerson implements edible{  1 usage
144
145         @Override  no usages
146         public void eat() {
147             System.out.println("eats everything...");
148         }
149     }
```

2- Multiple Inheritance

```java
//2- Multiple Inheritance

interface Flyable {  1 usage  1 implementation
    void fly();  no usages  1 implementation
}

interface Swimmable {  1 usage  1 implementation
    void swim();  no usages  1 implementation
}

class Duck implements Flyable, Swimmable {  no usages
    @Override  no usages
    public void fly() {
        System.out.println("Duck is flying...");
    }

    @Override  no usages
    public void swim() {
        System.out.println("Duck is swimming...");
    }
}
```

3- Unrelated Classes

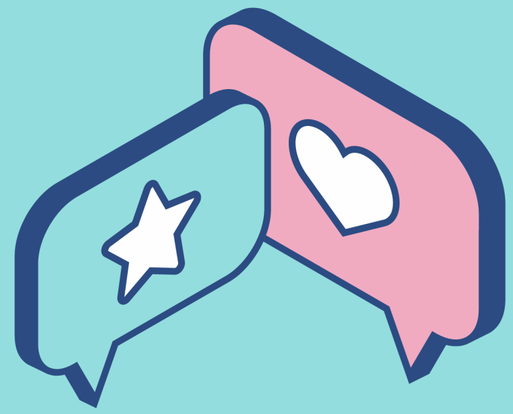
```
171
172     //3- Unrelated Classes
173     //If you have several classes that are not related by inheritance but share common functionality.
174
175     interface Audible {  2 usages  2 implementations
176         void makeSound();  no usages  2 implementations
177     }
178
179     class Dog implements Audible {  no usages
180         @Override  no usages
181         public void makeSound() {
182             System.out.println("Bark");
183         }
184     }
185
186     class AlarmClock implements Audible {  no usages
187         @Override  no usages
188         public void makeSound() {
189             System.out.println("Ring");
190         }
191     }
```

4- Defining a Contract

```java
//4- Defining a Contract
/*
    "defining a contract" refers to creating a set of methods that a class needs to implement.
    This idea makes sure that all the methods listed in an interface have concrete implementations for any class that claims to follow it.
    Without specifying how the techniques must perform, this contract defines what they must do.
 */
interface Animal {  2 usages  1 implementation
    void makeSound();  no usages  1 implementation
    void move();  no usages  1 implementation
}
class Wolf implements Animal {  1 usage
    @Override  no usages
    public void makeSound() {
        System.out.println("Wolf says: Woof!");
    }

    @Override  no usages
    public void move() {
        System.out.println("Wolf is running");
    }
}

class Bird implements Animal {  1 usage
    @Override  no usages
    public void makeSound() {
        System.out.println("Bird says: Tweet!");
    }

    @Override  no usages
    public void move() {
        System.out.println("Bird is flying");
    }
}
```

Another important factor in determining my choice is: Design patterns.

DESIGN PATTERNS OFTEN USE THESE STRUCTURES TO SOLVE SPECIFIC DESIGN PROBLEMS EFFECTIVELY. BY UNDERSTANDING THE NEEDS OF A PATTERN AND HOW ABSTRACT CLASSES OR INTERFACES FIT THOSE NEEDS, YOU CAN MAKE MORE INFORMED DECISIONS ABOUT WHICH TO USE.



THANK YOU