Use Inheritance and Polymorphism as a Mechanism for Reusability

Course: 420-310-DW Programming III

Instructor: Zahra Habibi

2024-11-11

Agenda

Inheritance:

- "Object" class as root class in all Java classes
- Inheritance examples & overriding the ***toString()*** method
- When and when not to use inheritance.

Polymorphism:

- Compile-time and run-time polymorphism.
- When and when not to use run-time polymorphism.
- Constructor Chaining

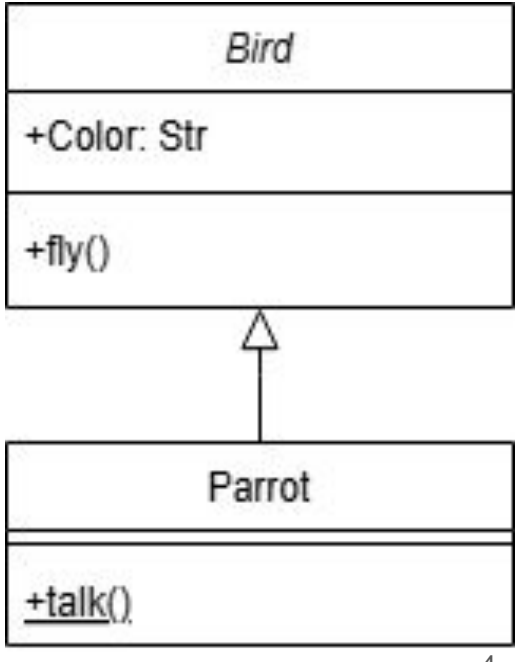
Summary:

- Case Study: Duck Hunt Simulation Game
  - Various challenges when attempting to avoid code duplication
  - Possible solutions with inheritance and polymorphism

Inheritance

Inheritance:

- One of the most important concepts in object-oriented programming.
- Inheritance allows us to define a class in terms of another class, which makes it **easier** to create, organize, and maintain an application.
- Inheritance provides the opportunity to reuse code functionality and accelerate implementation time (less time to code, test, and so on).
- When creating a new class, a programmer can simplify development by inheriting member variables and methods from an existing class, rather than writing them from scratch.
- Existing class is called the **base class** or **superclass**, and the new class is referred to as the **derived class** or a **subclass**.

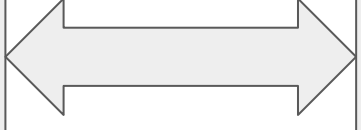


Inheritance: Object Class

- The Object class is the root class of the Java class hierarchy.
- It's part of the ***java.lang*** package, which is automatically imported into every Java program.
- This class provides several fundamental methods, like ***equals(), toString()***.

- **Implicit Inheritance Mechanism:** When you create a new class in Java, if you don't specify an explicit superclass using extends, Java automatically makes your class extend Object class.

```
public class MyClass {
        // class code
}
```



```
public class MyClass extends Object {
        // class code
}
```

Inheritance: Overriding *toString()*

- ***toString()*** method is defined in the Object class, and by default returns a string that includes the class name followed by the "@" symbol and memory address.
- Often overridden to provide a meaningful description of an object's state. Simply put, it returns a string representation of an object.

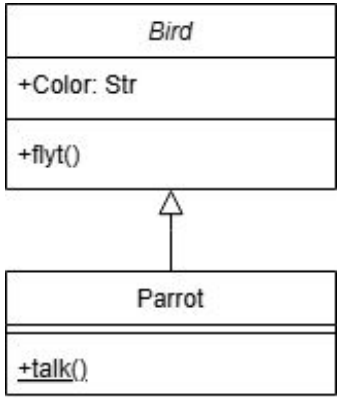
```
public class Main {
    public static void main(String[] args) {

        Person person = new Person("Alice", 30);
        System.out.println(person);
    }
}

 // Output: Person {name='Alice', age=30}
```

```
public class Person {

    private String name;
    private int age;

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

    @Override
    public String toString() {
        return "Person {name=' "+ name +" ', age=" + age + "}";
    }
}
```

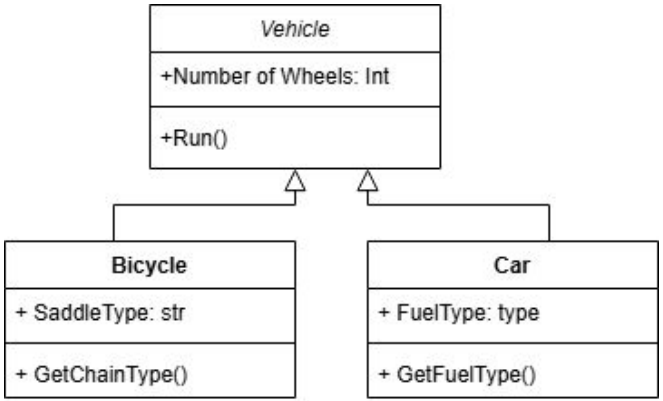
Inheritance: when to use?

- Inheritance allows us to define a class in terms of another class.
- We can use inheritance for:
  - **“Is-A” Relationship:** a ***Parrot*** is a ***Bird***, so it could inherit from the base ***Bird*** class.
  - **Code usability:** ***Car*** and ***Bike*** classes have similar methods and attributes, we can extract common code to a ***Vehicle*** superclass.
  - **Extending behavior:** ***Manager*** extends ***BaseEmployee*** while includes additional permissions or responsibilities.

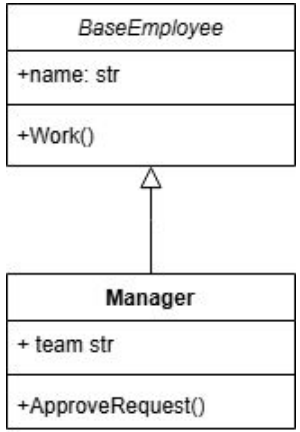
Is-A Relationship



Code Usability



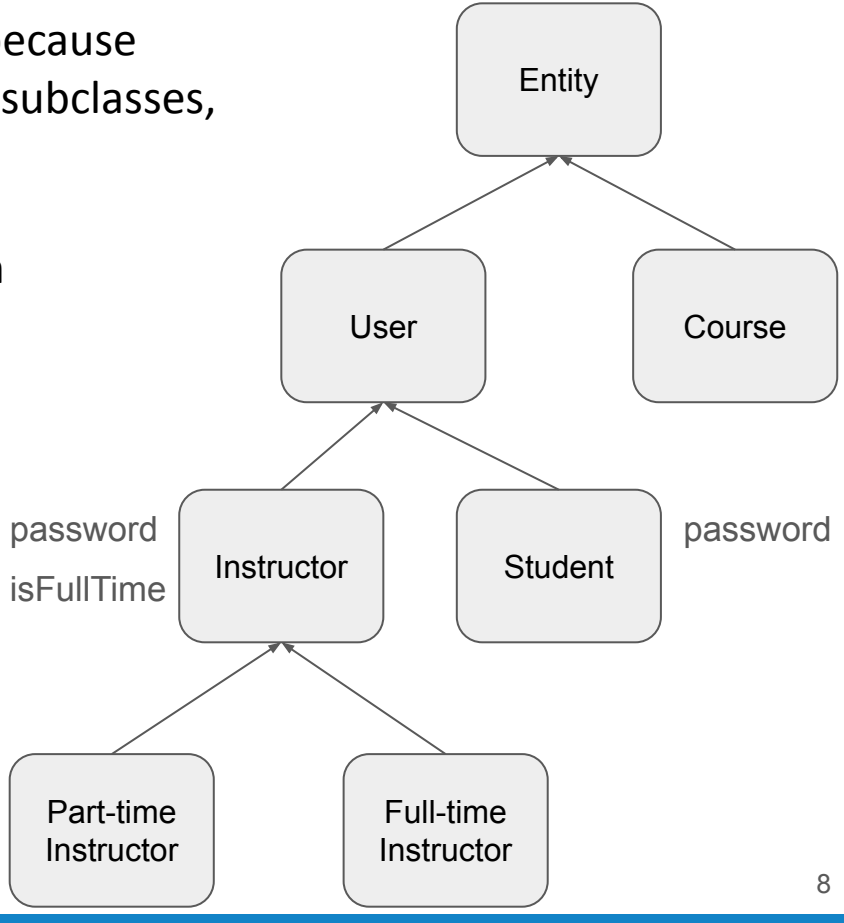
Extend Behavior



Inheritance: when not to use?

- **Avoid** inheritance with tightly coupled classes, because making change to superclass, leads to change to subclasses, which is not efficient for code maintenance.

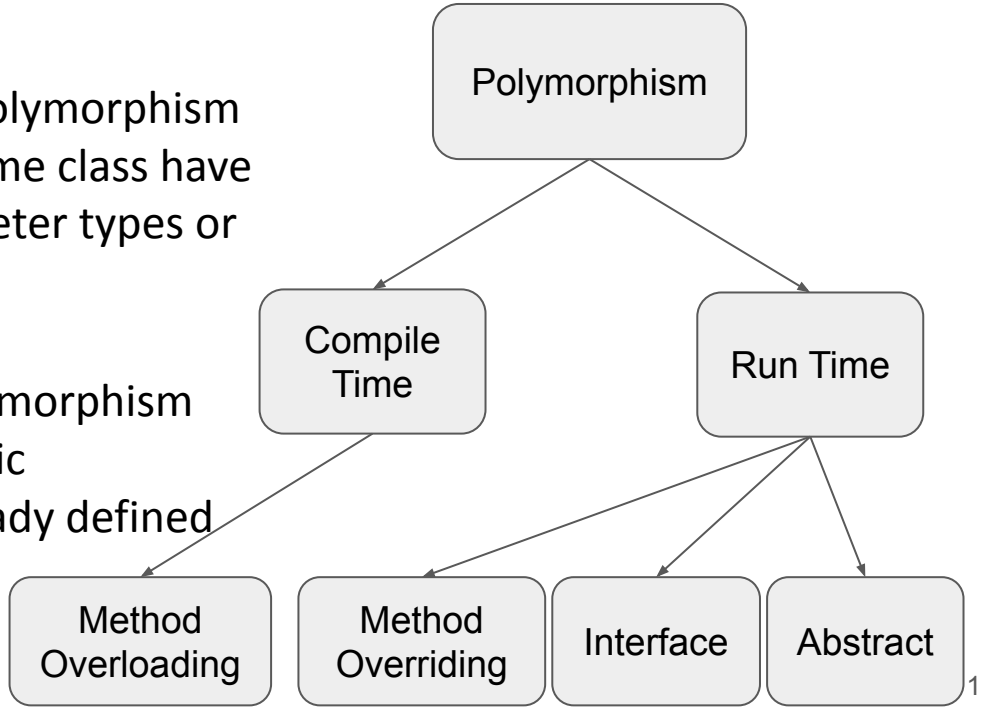
- A new method in ***Entity*** class, would be visible in all child classes (code pollution)
- Inheritance is usually good to one or two levels!
- Solution?
  - Remove ***User*** Class & move codes to subclasses (code duplication).
  - Remove Part-time & full-time instructor subclasses and use a boolean flag in parent Instructor class .



Polymorphism

Polymorphism:

- Polymorphism means “many forms”.
- It allows methods or objects to behave in multiple ways by enabling a single action to operate differently.
- **Compile-time** (**early binding** or **static**) polymorphism occurs when multiple methods in the same class have the same name but differ in their parameter types or numbers.
- **Run-time** (**late binding** or **dynamic**) polymorphism occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.



Polymorphism: Example 1 - Overloading

- The ***add()*** method in the same class is overloaded with different parameter lists, allowing the same method name to be used for different actions.
- The return type of a function has no effect on function overloading.

```
public class Main {
    public static void main(String[] args) {

        Calculator calc = new Calculator();

         // Calls add(int, int)
        System.out.println (calc.add(5, 10));

         // Calls add(int, int, int)
        System.out.println (calc.add(5, 10, 15));
    }
}
```

```
class Calculator {

    // Method to add two integers
    int add (int a, int b) {
        return a + b;
    }

    // Method to add three integers
    int add (int a, int b, int c) {
        return a + b + c;
    }
}
```

Polymorphism: Example 2 - Overriding

- It involves with inheritance.
- When ***myDog.sound()*** is called, Java dynamically determines the Dog class's sound method at runtime.

```
public class Main {

    public static void main(String[] args) {

        // Animal reference but Dog object
        Animal myDog = new Dog();

        // Calls Dog's overridden sound method
        myDog.sound();
    }
}
```

```
// base class
class Animal {

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

```
// subclass
class Dog extends Animal {

        @Override
        void sound() {
            System.out.println ("Dog barks");
        }
}
```

When and When not to Use Run-Time Polymorphism

- **Dynamic** polymorphism has slower execution compare to **static** polymorphism.
- However, dynamic polymorphism provides **flexibility**:
  - Work with objects without needing to know their specific class in advance.
  - Eliminating if-else or switch statements.

```
public class Game {
    public static void main (String[] args) {

        Character[] characters = new
                                    Character[w];

        characters[0] = new Warrior();
        characters[1] = new Archer();

        for (Character character : characters) {
            // Runtime polymorphism to
            // determine the type of object
            character.attack();
        }
    }
}
```

```
class Character {
    // better to use abstract class
    void attack() {
        System.out.println ("Character
            performs a generic attack.");
    }
}
```

```
class Archer extends Character {

    @Override
    public void attack() {
        System.out.println("Archer
            shoots a precise arrow!");
    }
}
```

```
class Warrior extends Character {

    @Override
    public void attack() {
        System.out.println("Warrior
            swings a mighty sword!");
    }
}
```

Constructor Chaining by Overloading

- **Constructor Chaining** is a technique where one constructor calls another constructor within the same class or from its superclass. Providing multiple options for creating an object.
- Within same class by using ***this( )*** keyword for constructors in the same class.

```
public class Example {
        public static void main (String[] args) {

                //calls constructor 1
                Person person1 = new Person("Alice", 25);
                // calls constructor 2
                Person person2 = new Person("Bob");

                System.out.println (person1); //outputs: Alice, 25
                System.out.println (person2); //outputs: Bob, 20
            }
}
```

```
public class Person {
    private String name;
    private int age;

    // Constructor 1: Name, age
    public Person (String name, int age) {
        this.name = name;  this.age = age;
    }
    // Constructor 2: Only name
    public Person (String name) {
        this (name, 20);   // calls constructor 1
    }
    @Override
    public String toString() {
        return name + ", " + age;
    }
}
```

Constructor Chaining To Invoke Superclass Constructor

- **Constructor Chaining** between subclass and superclass using ***super()***.
- Use ***super()*** **only** for constructor chaining in subclass.

```
// Main class
public class Main {

  public static void main (String[] args) {

      Dog dog = new Dog ("Buddy", 5, "Golden");

      // Pet constructor called for Dog
      dog.eat();   // output: Buddy is eating
    }
}
```

```
// Base class
class Pet {
    private String name;
    private int age;

    // Constructor for Pet
    public Pet (String name, int age) {
        this.name = name; this.age = age;
    }

    public void eat() {
         System.out.println (name + " is eating.");
} }
```

```
 // Subclass
class Dog extends Pet {
    private String breed;

     // Constructor for Dog
    public Dog (String name, int age, String breed) {
       super (name, age);   // calles Pet constructor
       this.breed = breed;}
}
```

Case Study:
Duck Hunt Simulation Game

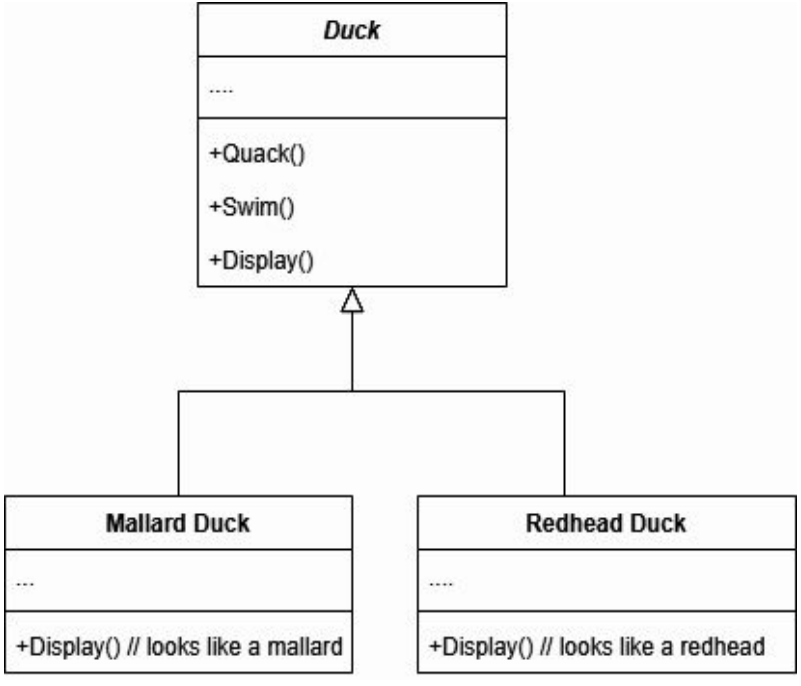
Case Study: Duck Hunt Simulation Game

- Suppose a company is building a **duck hunt simulation game** …
- Different options for modelling of object oriented programming for the ***duck*** class is suggested.

https://www.everything80spodcast.com/duck-hunt/

Modeling Duck: Polymorphism

- Assume we have 2 classes: ***RedHead Duck*** & ***Mallard Duck***.
- Both Redhead & Mallard have the same behavior: ***Quack()*** & ***Fly()***.
- We extract common code from the subclasses to superclass "***Duck***", and only override the ***display()*** method.
- ***Duck()*** superclass takes care of implementation of ***Quack()*** and ***Swim()*** Behaviors. Common behaviors between all ducks.
- Mallard & Redhead duck subclasses takes care of ***display()*** overridden method.
- Seems easy solution, right?



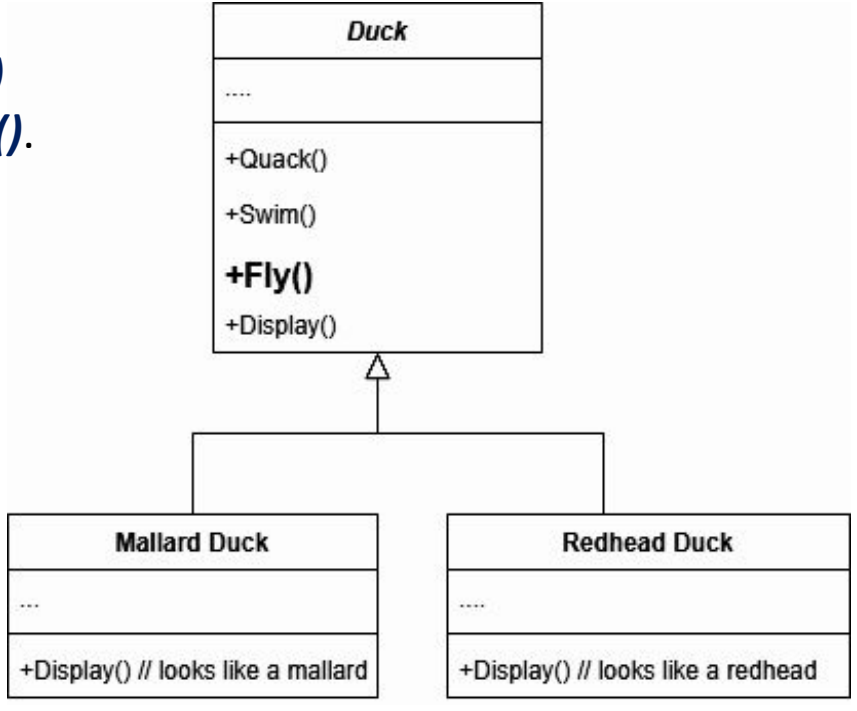
Modeling Duck: New Behavior

- After a few month, company decides to add a new behavior, ***Fly().***

- Both Mallard and Redhead ducks fly, so this ***Fly()*** method will be implemented in superclass ***Duck()***.

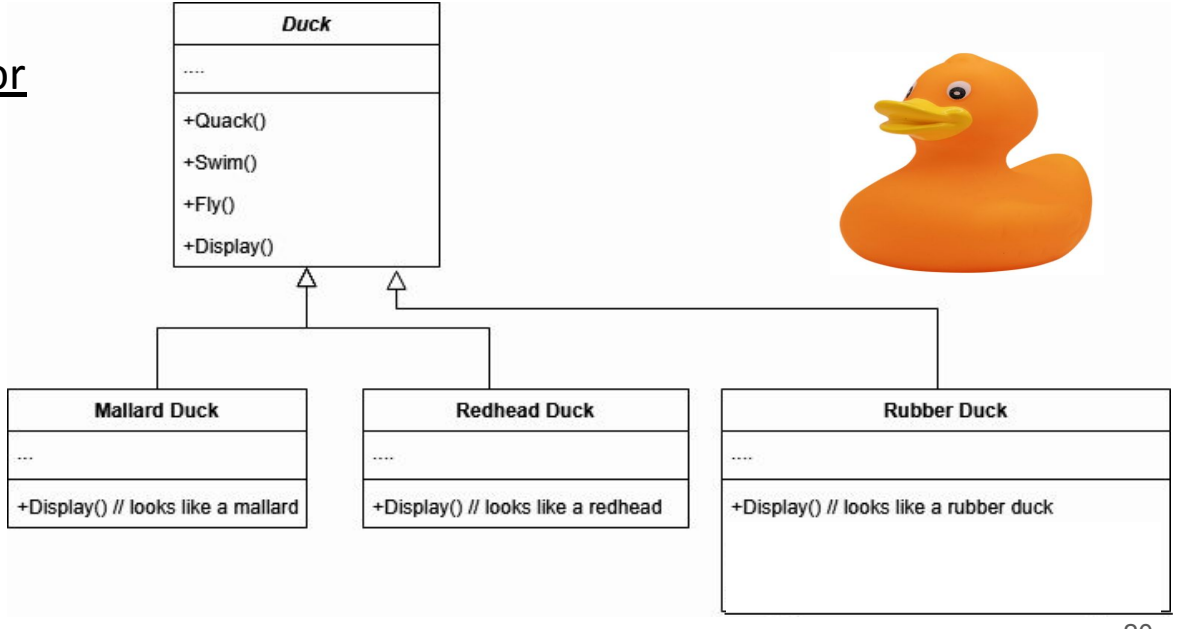
- **Recall:** Classes should be open for extension but close for modification.

- Still seems ok.



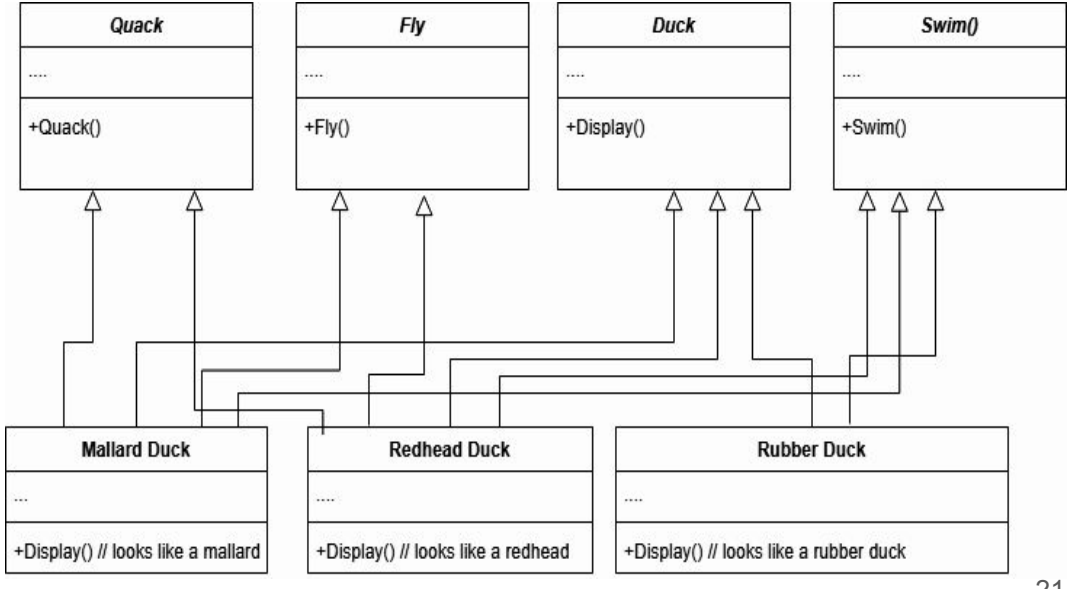
Modeling Duck: Problem

- Once again, after a few months, the company decides to add a new duck to the game—a rubber duck.
- **Problem:** Rubber Duck does not have ***Fly()*** nor ***Quack()*** behavior. But these methods (Fly & Quack) have already been inherited to the Rub duck class!
- **Solution:** Override these behavior in the rubber duck class and do nothing!
- Inheritance works well here, but it can get complicated if the company decides to add more classes with different behaviors.
- This solution is **not** scalable!



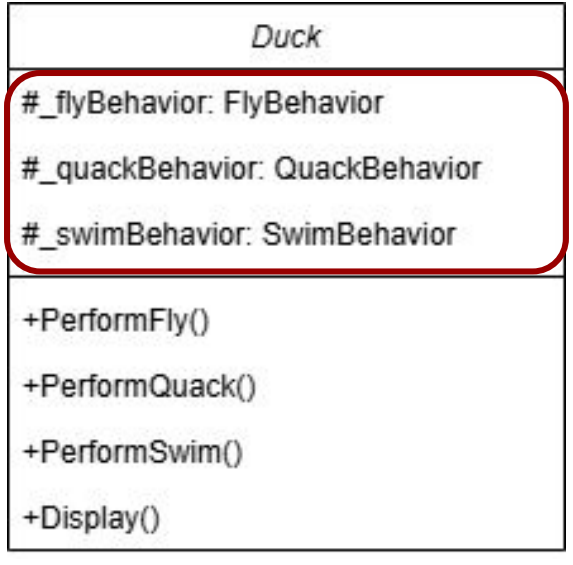
Modeling Duck: Solution 1 - Multiple Inheritance

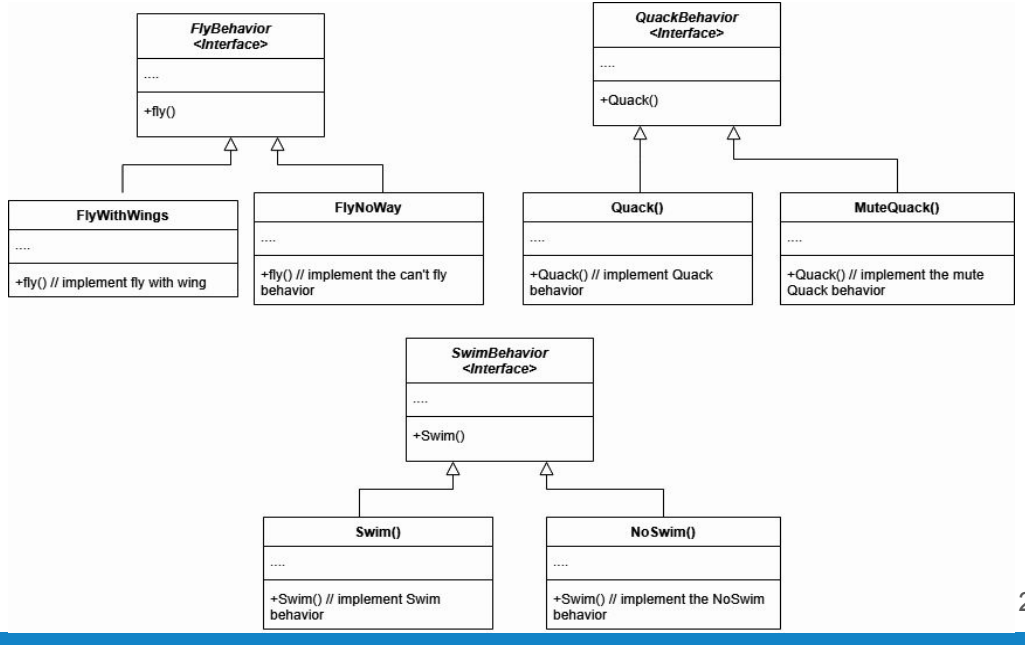
- Other **solution** is to separate the duck behaviors into distinct classes and apply **multiple inheritance** for subclasses like Mallard and Redhead ducks.
- But, Java does not support multiple inheritance directly through classes. We can use either interface or use "***Default Methods***" in Interfaces (Java 8+).
- Is the problem solved?
  - Increased complexity and reduced Readability.
  - "Diamond" problem, we may have conflict in which ***method()*** should be inherited.
- That's why Java has limited multiple inheritance.



Modeling Duck: Solution 2 - Interface / Abstract Class

- Another **solution** is to encapsulate the duck behaviors through creating **abstract classes** or **interfaces** for duck behaviors, ***Quack()***, ***Fly()***, ***Swim()***.
- We will store **fly** and **quack** behaviors as instance variables in the ***Duck*** class.
- A duck will delegate its flying and quacking behaviours (rather than implementing them itself).





Modeling Duck: Solution 2 - Interface / Abstract Class

DAWSON COLLEGE

- Behavior variables like "***_flyBehavior***" and "***_quackBehavior***" are declared as the behavior interface type.


```
public class Duck {

    protected FlyBehavior _flyBehavior;
    protected QuackBehavior _quackBehavior;

    public Duck() { }

    public void performFly() {
            _flyBehavior . fly();
    }
    public void performQuack() {
            _quackBehavior.quack();
    }
}
```

```
// Fly behavior Interface
interface FlyBehavior {
    // Abstract method (does not have a body)
    void fly();
}

// Implement the interface
class FlyWithWings implements FlyBehavior {
    public void fly() {
        System.out.println("flying...");
    }
}

// Implement the interface
class FlyNoWay implements FlyBehavior {
    public void fly() {
        System.out.println("can't fly ...");
    }
}
```

Modeling Duck: Solution 2 - Interface / Abstract Class

DAWSON COLLEGE

```
// Main class
public class Main {

    public static void main(String[] args) {

        MallardDuck mallardDuck = new MallardDuck();
        RubberDuck rubberDuck = new RubberDuck ();

        // Perform behaviors
        mallardDuck.performFly();
        mallardDuck.performQuack();
    }
}
```

```
class MallardDuck extends Duck {

    MallardDuck () {
        this._flyBehavior = new FlyWithWings();
        this._quackBehavior = new Quack();
    }
}
```

```
class RubberDuck extends Duck {

    RubberDuck () {
        this._flyBehavior = new FlyNoWay();
        this._quackBehavior = new MuteQuack();
    }
}
```

Advantages:

- Can **easily** add / modify duck types and behaviours without necessarily (or heavily) modifying our duck classes.
- **Eliminated** code duplication.

Next Session

- Interface & Abstract classes will be covered in detail.

Thank you