Interfaces

An interface is a special object-oriented structure that defines a set of behaviors (i.e., methods) which implementing classes must provide. Interfaces support abstraction and code reusability by allowing different classes to share a common set of behaviors without being tied to a specific class hierarchy.

Content Learning Targets

After completing this activity, you should be able to say:

- I can define an interface and distinguish it from an abstract class.
- I can explain some use cases of interfaces in object-oriented design.
- I can identify common methods in a set of classes and refactor them into an interface.

Process Skill Goals

During the activity, you should make progress toward:

- Interpreting UML class diagrams. (Information Processing)

Facilitation Notes

Model 1 Introduction to Interfaces

Let's look at an example of an interface in Java. Suppose we are designing a software system (e.g., a cartoon racing game) that involves various types of vehicles. Every vehicle type provides the ability to turn and accelerate, but the specific implementation of these behaviors may differ between vehicle types (e.g., a car turns differently than a bike). We can define an interface called `Drivable` that specifies these common behaviors:

```
public interface Drivable {
    void turn(double dir);
    void accelerate(double force);
}
```

Each vehicle class can then implement the `Drivable` interface, providing its own specific implementation of the `turn` and `accelerate` methods:

```
public class Car implements Drivable {
    @Override
    public void turn(double dir) {
        // Car-specific turning logic
    }

    @Override
    public void accelerate(double force) {
        // Car-specific acceleration logic
    }
}

public class Bike implements Drivable {
    @Override
    public void turn(double dir) { // Bike turning logic
    }

    @Override
    public void accelerate(double force) { // Bike acceleration logic
    }
}

public class Hovercraft implements Drivable {
    @Override
    public void turn(double dir) { // Hovercraft turning logic
    }

    @Override
    public void accelerate(double force) { // Hovercraft acceleration logic
    }
}
```

Questions (35 min) Start time:

1. First, let's look at the interface syntax. What keyword is used to define an interface in Java? How do you declare that a class uses an interface? What annotation do we add to the methods in the implementing class to indicate that these methods are from the interface?

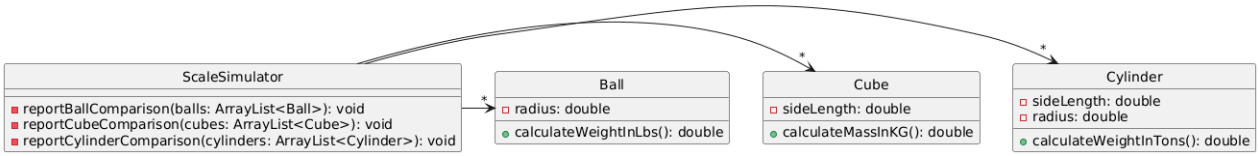
The keyword `interface` is used to define an interface. A class uses an interface by using the `implements` keyword followed by the interface name. The annotation `@Override` is added to the methods in the implementing class to indicate that these methods are from the interface.

2. What are some potential benefits of using interfaces in Java?

Interfaces allow for abstraction, enabling different classes to share a common set of behaviors without being tied to a specific class hierarchy. This promotes code reusability and flexibility, as new classes can implement the same interface without needing to inherit from a common superclass. The Java compiler will make sure all classes which implement the interface have overridden its methods. Interfaces also facilitate ***polymorphism***, allowing objects of different classes to be treated uniformly based on the shared interface. We will see examples of this later.

Live coding: Refactoring to use an interface

3. Open the `src/simpleExample` folder in your IDE. The current code follows the following UML design (PlantUML link):



What common behaviors do the classes `Cylinder`, `Cube`, and `Ball` share?

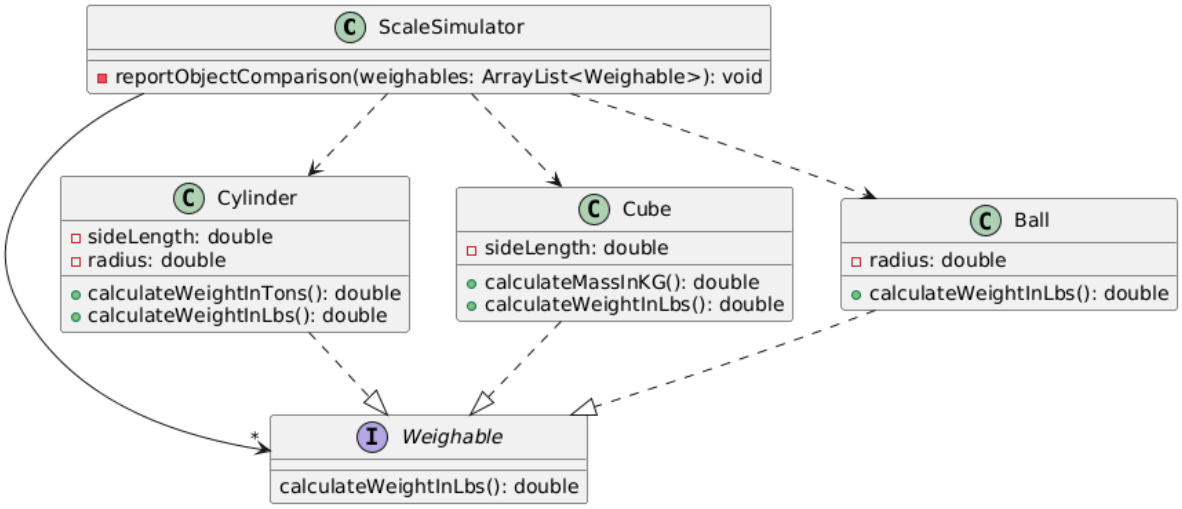
Some sort of weight measurement, though in different units for each.

4. Let's refactor the code to introduce an interface called `Weighable` that declares a method for measuring weight in pounds.

5. Next, each of the three classes should implement this interface, providing their own specific implementation of the weight measurement method.

6. Finally, let's update the `ScaleSimulator` class to use the `Weighable` interface instead of the specific classes. This will allow the scale simulator to work with any object that implements the `Weighable` interface, promoting flexibility and code reusability. Notice that Java allows us to store different types of objects in the same ArrayList, as long as they implement the same interface!

7. The final UML design should look like this (PlantUML link):



Notice the two new arrow types: the dashed arrow with an open triangle head indicates an "is-a" relationship where the class is implementing the interface. For example, `Cylinder` ***is-a*** `Weighable`. The dashed arrow with a plain head indicates a "depends-on" relationship where one class uses another ***somewhere***, but not as an instance variable.

8. Where does the `ScaleSimulator` class use the `Cylinder`, `Cube`, and `Ball` classes?

The `ScaleSimulator` class uses the constructors of these classes to create instances of them, which are stored in the ArrayList of `Weighable` objects.

9. Based on the examples we have seen so far, how would you distinguish between an abstract class and an interface? When might you choose to use one over the other?

An abstract class can have instance variables and concrete methods (with implementations), while an interface can only declare method signatures (without implementations) and constants (static final variables). A class can implement multiple interfaces but can only inherit from one abstract class. You might choose to use an abstract class when you want to provide a common base with shared behaviors (methods) and data (fields) for related classes. You might choose to use an interface when you want to define a contract for behavior that can be implemented by unrelated classes (maybe with different superclasses), promoting flexibility and code reusability.

Model 2 Zookeeper Problem

In this activity, we will refactor a system that models a pet zoo to use an appropriate interface.

Questions (35 min) Start time:

10. To check your understanding, is the following a valid Java interface? Why or why not?

```
public interface Pet {
    private String name;

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

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

No, this is not a valid Java interface. Interfaces can't have instance variables or constructors. They can only declare method signatures (no bodies) and static final constants. Interface methods are implicitly abstract and public, so they can't have a body unless they are default or static methods (which is not the case here). A valid interface would look like:

```
public interface Pet {
    void eatFood();
}
```

11. Fill in the blanks to complete the `Cat` class so that it correctly uses the `Pet` interface.

```
public class Cat __________ Pet {
    private String name;

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

    @_____________
    public void ___________ {
        System.out.println(name + " purrs.");
    }
}
```

`implements, Override, eatFood()`

12. Is each of the following statements valid or invalid? Explain why.

a) `Pet myPet = new Pet();` — Invalid. Can't instantiate an interface.

b) `Pet myPet = new Cat("Whiskers");` — Valid. "Cat is-a Pet", so this works.

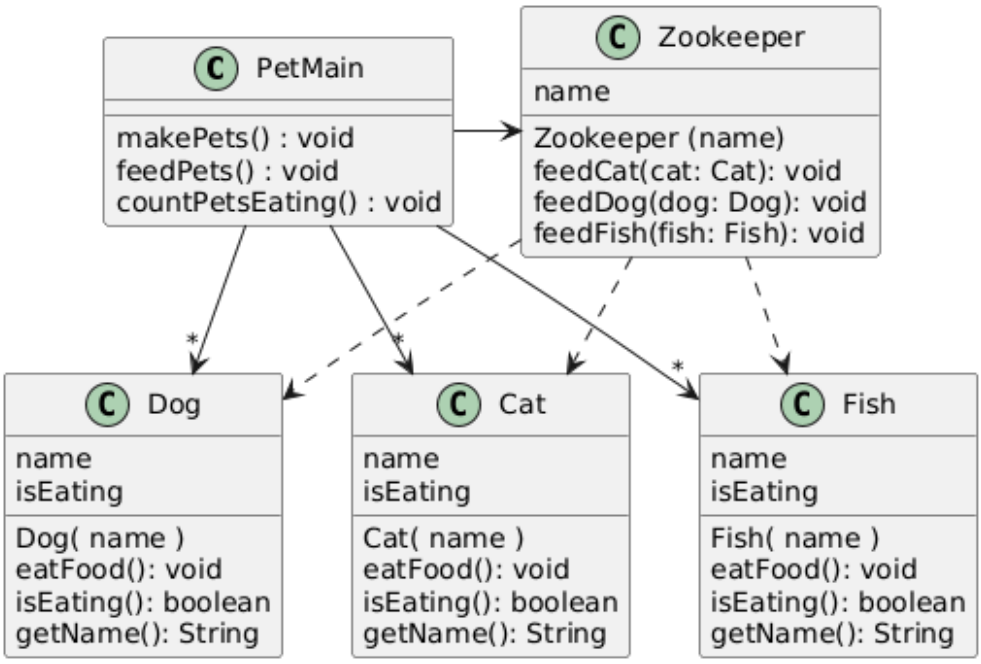
c) `Cat myCat = new Pet();` — Invalid. A Pet isn't necessarily a Cat.

d) `Cat myCat = new Cat("Whiskers");` — Valid.

Coding activity: Zookeeper system

In the following scenario we have a Pet Zoo, with a Zookeeper who is in charge of feeding different types of animals. When the simulator runs, various pets are made and fed. Also, there is a way to count the number of pets that are eating. The animals include cats, dogs, and fish. All the animals have names, and can be told to eat food, as well as report that they are eating (once fed they always report eating).

Original design: (PlantUML link)



13. Improve this UML design by introducing an appropriate interface to remove code duplication. What common behavior(s) do the classes `Cat`, `Dog`, and `Fish` share? What can we do instead of the three `feed` methods in `Zookeeper`?

Improved design with `Pet` interface: PlantUML link

14. In the improved design, why does `PetMain` still depend on the concrete classes `Cat`, `Dog`, and `Fish`? And why does `Zookeeper` depend on the `Pet` interface?

`PetMain` needs to create instances of the concrete classes, so it must depend on them. `Zookeeper` depends on the `Pet` interface because it takes an object of type `Pet` as a parameter in its `feedPet` method. This allows `Zookeeper` to work with any object that implements the `Pet` interface, promoting flexibility and code reusability.

15. Now, ***working in pairs***, implement the improved design in Java. You can start with the provided code in the `src/pets` folder.

1. Add the `Pet` interface and refactor the `Cat`, `Dog`, and `Fish` classes to implement it.
2. Refactor the `Zookeeper` class to use the `Pet` interface instead of the concrete animal classes.
3. Refactor `PetMain` to make a single list of `Pet` objects instead of three separate lists.

When you finish, compare your code to the provided solution in the `sol/pets` folder.

16. The word ***polymorphism*** comes from the Greek for "many" and "shapes" (or "forms"). An interface can take many shapes; for example, a variable with declared type `Pet` could hold a reference to a `Cat`, `Dog`, or `Fish` object. Since each of these classes implements the `Pet` interface, we know they all have the `eatFood` method. So regardless of what object type we actually have, we can safely call `eatFood` on it and we know the correct version will be called. This is a ***polymorphic method call***.

Model 3 Interface Practice

For additional practice with interfaces, see the following exercises.

- In `src/numberSequence`, follow the TODOs to implement the `Sequence` interface, allowing for different types of number sequences with the same basic functionality of getting the next number or resetting to the beginning.

- In `src/shapes`, explore the example of using an interface to draw different shapes. (This example combines Swing graphics, recursion, ***and*** interfaces.) Try adding a new type of shape (e.g., a triangle). Notice how the existing code works with very little modification.