Lab 2: Interfaces and Polymorphism

Reading

Intro to Interfaces!

Interfaces are like blank classes with only method declarations — no definitions. In essence, an interface is like a "contract" for a class, specifying the methods that a class implementing it must contain. However, while a class can only inherit from a single superclass, it can implement any number of interfaces. Any class that implements an interface must implement all the methods of that interface. Most of the time, interfaces are named after the capabilities they enforce: amongst the many possibilities are `Sizable`, `Movable`, `Cookable`, and in this lab's case... `VideoProvider`. Interfaces are useful because they enforce commonality among different classes and enable generic programming.

Syntax for Interfaces

Interfaces are created in their own file in a program, much like creating classes. However, instead of using the keyword `class`, you will write `interface`. Unlike classes, interfaces never have constructors. The only things that should be contained in your interface are declarations for the methods you want the interface to enforce. A `Movable` interface could declare `move` as its method, or a `Cookable` interface could declare `bake` and `grill` as its methods. The syntax for declaring a method within an interface is simple, just the return type of the method and the name of the method. The syntax for the `Movable` interface's method declaration would be `public void Move();`. Remember, the return type could be other things, such as Strings, ints, or any other class!

Abstract Methods

We mentioned earlier that interfaces contain only **abstract** methods. This means that the method has been declared in the interface file, but it contains no code to implement the method. So, when we create a class that implements an interface, we will need to **override** those abstract methods and add code to implement them. How is this done? We use the annotation `@Override` above a method to the Java compiler that it should override the existing method. It is important that the method signature (the keywords that appear before the name of the method) are identical to the signature of the method that is being overridden. Returning to the example of the `Moveable` interface, we would override the abstract `move()` method with:

```
@Override
public void move(){
```

```
        // Code to implement the move() method
}
```

More on Polymorphism

Understanding polymorphism is vital to good object-oriented programming, so it is important to come to this lab with a grasp on the concept. If you're still feeling confused, here are some resources that may have an explanation you find more effective:

- **Lecture Slides/Lecture Capture**: It's often very helpful to revisit past lectures to hear examples again, pick up on information you may have missed, or to generally review concepts.
- **Piazza**: If you have conceptual questions about polymorphism, Piazza is a great place to ask them! TAs respond quickly to Piazza posts, and you'll often find that others share your questions.
- **TA Hours**: TAs are more than happy to answer conceptual (non-code related) questions during TA hours, so this is a great option for a more 1-on-1 conversation.