Java Interfaces
Abstract Classes
I Need to Write a Strategy!?!?!?

Keep in mind that even a simple strategy is a strategy. For a card game, it could be as simple as:

1. Scan my hand of cards
2. Play first playable card

Or as complex as:

1. Keep track of all cards that have been played
2. Use the current conditions and the cards I’ve seen to select a card
Java Interface

https://courses.grainger.illinois.edu/cs126/sp2022/assignments/mineopoly/

In the current assignment, you are called upon to “implement this interface without modifications”.

What, exactly, is an interface?
The Following Slides Contain Links
Java Interface

An **interface in Java** is a blueprint of a class. It has

- static constants
- abstract methods
Abstract Class

A class which contains the abstract keyword in its declaration is known as an abstract class.

- Abstract classes may or may not contain *abstract methods*, i.e., methods without body.
- If a class has at least one abstract method then the class must be declared abstract.
- Abstract classes cannot be instantiated.
- To use one, you must inherit it from another class (this will require you to provide implementations to its abstract methods).
Java Interface vs Abstract Class

There are a number of key differences between the two. They are similar:

- Both are used to achieve abstraction.
- Both require you to complete the implementation of methods.
- Neither can be instantiated.

You can find a table with side-by-side comparisons here:

https://www.javatpoint.com/difference-between-abstract-class-and-interface
Interfaces Are Like Classes Except...

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- The only fields that can appear in an interface must be declared both static and final.
- They are implemented by a class.
- An interface can extend multiple interfaces.
Interfaces Are Like Classes Except...

OK, so maybe they’re really not that much like classes.

- They are blueprints for classes or parts of classes. You might think of them as a
  - set of required methods.
- Whereas, an Abstract Classes contain everything you need to create a class except for some implementations. You might think of them as an
  - incomplete class
Interface Properties

- Interface itself:
  a. implicitly abstract

- Each method in an interface:
  a. implicitly abstract
  b. Implicitly public
Extend or Implement?

It may be helpful to keep the meanings of these words in mind.

- **Extend** - “cause to cover a larger area; make longer or wider”
- **Implement** - “put (a decision, plan, agreement, etc.) into effect”

An *interface* can only extend another interface.

An *abstract class* can do either or both (extend or implement).
Form of an Interface Declaration

public interface NameOfInterface {
    // Any number of final, static fields
    // Any number of abstract method declarations
}

Interface Declaration Example

```java
public interface Animal {
    public static final String methods = "Methods: void eat(String food), void move()";
    void eat(String food);
    void move();
}

*notice that the methods are not declared “public” nor “abstract”

**I can’t really think of a good reason to have the above field “methods”
public interface Animal {
    String methods = "Methods: void eat(String food), void move()";
    void eat(String food);
    void move();
}

*what’s the difference? Go ahead and try these yourself*
Example

```java
public interface Animal {
    void eat(String food);
    void move();
}

public interface Mammal {
    void twitchNose();
    void gestationPeriod();
}
```
Extending an Interface means...

```java
public interface Mammal extends Animal {
    void twitchNose();
    void gestationPeriod();
}

*Mammal inherits from Animal

**Now a Mammal IS-A Animal
Adding On

An interface “extends” another interface.

Extend - “cause to cover a larger area; make longer or wider”

- An interface can extend more than one interface but a class can not.
- A class can IMPLEMENT more than one interface.

```java
public interface Mammal extends Animal {
    void twitchNose();
    void gestationPeriod();
}
```
Extending an Interface means...

Public class Cat implements Mammal {

}

“Mammal extends Animal” means that a class implementing Mammal requires the class to implement ALL of these methods:

    void eat(String food);
    void move();
    void twitchNose();
    void gestationPeriod();
OR...a class can implement more than one interface:

```java
public interface Reptile {
    void twitchTail();
    void eggsInClutch();
}

public class Dragon implements Reptile, Animal {
    // note: this is a class and can do what classes can do
    // but it MUST implement the methods of both Mammal and Animal
}
```
OR...a class can implement more than one interface:

public class Dragon implements Reptile, Animal {
    // note: this is a class and can do what classes can do
    // but it MUST implement the methods of both Mammal and Animal
    void eat(String food);
    void move();
    void twitchTail();
    void numberOfEggs();
}

*Dragon is **composed of** Reptile and Animal*
Wait...what’s the difference?

Mammal EXTENDS Animal so they are now linked together.
Reptile and Animal are not linked together.
Dragon IMPLEMENTS both but they are not combined otherwise.

A class can implement multiple Interfaces.
It might make more sense to...

```java
public interface Reptile extends Animal {
   void twitchTail();
   void eggsInClutch();
}

public class Dragon implements Reptile {
   // note: this is a class and can do what classes can do
   // but it MUST implement the methods of Mammal
}
```
Inheritance vs Composition

Is-A - inheritance

Has-A - Composition

Again, if we look at the definitions of the English words, we can aid our insight.

- Inherit - derive (a quality, characteristic, or predisposition) genetically from one's parents or ancestors
- Compose - constitute or make up
Inheritance

Usually, composition is preferred to inheritance.

Inheritance:

- **Inherited** functions work slower than normal function as there is indirection.
- Often, data members in the base class are left unused which may result in wasted memory.
- **Inheritance** increases the coupling between base class and derived class.
Composition Over Inheritance

“is the principle that classes should achieve polymorphic behavior and code reuse by their composition rather than inheritance”

https://en.wikipedia.org/wiki/Composition_over_inheritance

Recommended Reading:

https://stackoverflow.com/questions/49002/prefer-composition-over-inheritance
Inheritance or Composition?

public interface Mammal extends Animal {
  void twitchNose();
  void gestationPeriod();
}

This is:

1. Inheritance
2. Composition
3. Neither
Inheritance or Composition?

public class Cat implements Mammal, Animal {
    public Animal myAnimal;
    public Mammal myMammal;
}

This is:

1. Inheritance
2. Composition
3. Neither
Inheritance or Composition?

public class Cat implements Mammal, Animal {
    public Animal myAnimal;
    public Mammal myMammal;
}

*myAnimal and myMammal are only references to types (with no values) and they won’t help you implement the methods of Animal nor of Mammal. They won’t achieve an implementation nor an abstraction.
Recommended References

Is-A, Has-A - Inheritance vs Composition


https://www.w3resource.com/java-tutorial/inheritance-composition-relationship.php
Code Examples

Abstract Class: Animal

Class Dog: Extends Animal

Interface: AnimalInt

Interface: ReptileInt

Interface: MammalInt Extends AnimalInt

Class: Cat Implements MammalInt

Class: Dragon implements ReptileInt, AnimalInt
Strategy Pattern

A couple of points:

1. There is no requirement that you use this pattern for this assignment.
2. In fact, you may have to go to extra trouble to use this.
Java Interface vs Abstract Class

Table with side-by-side comparisons here:
https://www.javatpoint.com/difference-between-abstract-class-and-interface
Composition Over Inheritance

“is the principle that classes should achieve polymorphic behavior and code reuse by their composition rather than inheritance”

https://en.wikipedia.org/wiki/Composition_over_inheritance

Recommended Reading:

https://stackoverflow.com/questions/49002/prefer-composition-over-inheritance
Recommended References

Is-A, Has-A - Inheritance vs Composition


https://www.w3resource.com/java-tutorial/inheritance-composition-relationship.php
The Strategy Pattern References

From simplest to more robust:

1. https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm
2. https://refactoring.guru/design-patterns/strategy
3. https://sourcemaking.com/design_patterns/strategy