Section 6
Overview
The goals of this section are to:
1) Learn the role abstract classes and interfaces play in programming
2) Gain a clearer understanding of inheritance (polymorphism, abstract methods, etc.)
3) Use inheritance to make add more interesting elements to a game

Introduction
Inheritance is a very tricky concept, but it is used everywhere around us. In this lab, we will be working a small and simple game based in the command-line, with the help of abstract classes.

Abstract Classes
https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html
Abstract classes are one of the ways Java handles inheritance. The first thing to notice, is that they are declared as “abstract” in the class definition. Furthermore, the abstract class will usually include abstract methods. These are easy to tell apart from normal methods because they will usually have the “abstract” keyword, as well as have no implementation. Abstract classes cannot be instantiated, and abstract methods cannot be called.

Why would you use an abstract class? Well, there are many perks of using an abstract class, such as the ability to categorize things. For example, imagine if you had a bunch of classes that are all vehicles (car, bicycle, motorcycle, etc.). Normally, you would not be able to link them together nicely so that they all share the same methods or fields. However, with the power of inheritance and the extends keyword, we can have them use the same methods and fields! And if we want them to have the same method, but different implementations, abstract methods are just what we are looking for.

A short example of an abstract class:
```java
public abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```
Where **inheritance** comes in, is with the use of the **extends** keyword. When class A extends class B, that means that A is forced to provide implementations of B’s abstract methods, as well as inherit B’s methods and fields.

```java
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

Here, Circle has to provide its own implementation of the draw() and resize() methods in GraphicObject, because those methods are abstract. Also, Circle would be able to access GraphicObject’s x and y fields, as well as use the moveTo() method. These fields and methods are inherited because Circle extends GraphicObject, so all Circle objects would be able to use them.

If we were to create a new class called Square that extends GraphicObject, it would be forced to implement draw() and resize() as well. This gives us the powerful ability on calling draw() on any GraphicObject, whether it’s a Circle, Square, or anything else that extends the GraphicObject class. The program doesn’t even need to know what class it really is! As long as it extends GraphicObject, it is guaranteed to have draw() and resize().

**Interfaces**

https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

While we are not going to be dealing with **interfaces** in this lab, it is important to know about them. In many ways, they are very similar to abstract classes - they cannot be instantiated and their methods can be abstract. The differences begin with the fact that ALL their methods must be abstract. Also, the variables that are declared within an interface are always public, static, and final. And finally, to inherit from an interface, the **implements** keyword is used instead of the **extends** keyword.

Why would anyone use an interface rather than an abstract class then? The biggest advantage an interface has over an abstract class is that while a class can only EXTEND a single parent class, it can IMPLEMENT multiple interfaces. This means that classes that inherit multiple interfaces are guaranteed to have implemented the abstract methods that are included in all of the interfaces.
We have coded up a nice little game for you to play. Instructions are provided in the game menu and the only animal you can currently add to your zoo is “cat”.

How does this program work? If you glance through the code in Zoo.java, you’ll see that the Zoo class never knows what specific animal it’s actually dealing with. So long as the animals we implement extend the Animal class, they are guaranteed to have the methods getType(), getName(), and getCall(), even if these methods return null. This is part of a concept called polymorphism, when an object can take on many forms. In this case, the Cat class is considered a Cat, a Carnivore, and an Animal all at once. Because we know Cat has all the methods Animal does, we can treat it like an Animal and call Animal’s methods on it.

This game is rather boring with only cats. Let’s create a new animal that extends the Carnivore class, with 1. a constructor that sets the name variable to whatever you want, and 2. overwrite Carnivore’s getCall() method by making your carnivore return a String. Choose whatever carnivorous animal you’d like. Look at Cat.java if you need help. When you’re finished, modify addAnimal() in Zoo.java so you add your animal to your zoo, then try it out.

It seems that without herbivores, no visitors want to come to your zoo. Let’s fix this by implementing a new Herbivore class that extends the abstract Animal class. Keep in mind that extending an abstract class to a non-abstract class forces you to also implement the abstract methods. For those of you using an IDE, note how an error will appear informing you that getCall() needs to be implemented. For now, 1. make getCall() return null, and 2. make the constructor set the variable type to “herbivore”. If you’re unsure on how to do that, take a look at the Carnivore class.

Remember that the Herbivore class, like the Carnivore class, is supposed to be a parent class. We don’t plan on creating instances of Herbivore in the game - though it is possible since unlike Animal, Herbivore is not abstract. Instead, like you did with the carnivorous animal you created, make an animal class that extends Herbivore and add it to the game.

If you’ve gotten this far, keep adding more animals, whether carnivores or herbivores, and see how large you can make your zoo! Our (not very) high score is 16. Can you beat that?
Questions
1. We looked at abstract classes as a way Java deals with inheritance. What is another way inheritance can be handled in Java?
2. Draw a high-level design inheritance model of your Animal classes. It should look something like this:

![Inheritance Diagram]

3. Do abstract classes necessarily have to have an abstract method?
4. Name some examples of things that would be an abstract class, and what would extend that class (ex: abstract class: candy, classes that extend it: tootsie rolls, smarties, starburst etc.).
5. Is it efficient to have to overwrite getCall() for every animal that you implement? What would be a better way to implement getCall()? Take a look at the getType() and getName() methods for Animal.java. In this case, where would we initialize the call String to return for getCall()?