CPSC 1181
Lab 6: Inheritance

Objectives:

To understand principles of inheritance and its role in object-oriented programming

To be able to use inheritance as a tool for modeling

Introduction:

"In object-oriented design, **inheritance** is a relationship between a more general class (called the **superclass**) and a more specialized class (called the **subclass**)"(Horstmann 422). Over the course of this lab, you will explore the concept of inheritance through programmed and written examples. Reference for this lab can be found in *Big Java Chapter 9: Inheritance.*

Procedure:

You first saw Inheritance when displaying the House in lab 2. In that situation, you extended the class Rectangle with your own class Door because the door in question had the properties of a rectangle. In the strictest sense, this is not true object oriented inheritance because not all doors are necessarily rectangles. A more apt name for the class would have been RectangularDoor, because it describes the inherited quality strictly. In the following question, you will be walked through the steps in creating a basic hierarchy of classes exhibiting inheritance.

1.1) Consider the following *Card* class

```
public class Card
{
   private String name;

   public Card()
   {
      name = "";
   }

   public Card(String n)
   {
      name = n;
   }

   public String getName()
   {
      return name;
   }

   public boolean isExpired()
   {
      return false;
   }

   public String format()
   {
      return "Card holder: " + name;
   }
}
```

Use this class as a superclass to implement a hierarchy of related subclasses. Identify the subclasses as follows, and supply private instance variables for the data described.

Leave the body of the constructors and format methods for these subclasses blank for now, but implement getters for the private instance variables. One of these subclasses should inherit from another one. Pay close attention, and implement the hierarchy accordingly.

Class	Data
IDCard	ID number
CallingCard	Card number, PIN
DriverLicense	Expiration year

1.2) Implement a constructor for each of these subclasses. Each constructor should call the superclass' constructor to set the name. Here is one example:

```
public IDCard(String n, String id)
{
   super(n);
   idNumber = id;
}
```

1.3) Replace the implementation of the format method for the three subclasses. The methods should produce a formatted description of the card details. The subclass methods should call the superclass format method to get the formatted name of the cardholder.

1.4) Devise another class, `Billfold` which contains slots for two cards, `card1` and `card2`, a method `void addCard(Card)`, and a method `String formatCards()`.

The `addCard` method checks whether `card1` is `null`. If so, it sets `card1` to the new card. If not, it checks `card2`. If both cards are set already, the method has no effect.

Of course, formatCards invokes the format method on each non-null card and produces a string with the format

```
[card1|card2]
```

1.5) Write a tester program that adds two objects of different subclasses of Card to a Billfold object. Test the results of the formatCards methods.

1.6) Explain why the output of your BillfoldTester program demonstrates polymorphism.

1.7) The Card superclass defines a method isExpired, which always returns false. This method is not appropriate for the driver license. Supply a method header and body for DriverLicense.isExpired() that checks whether the driver license is already expired (i.e., the expiration year is less than the current year).

To work with dates, you can use the methods and constants supplied in abstract class Calendar which are inherited by the concrete class GregorianCalendar. You create a Calendar as follows:

```
GregorianCalendar calendar = new GregorianCalendar();
```

Then, you can obtain the current year using the constant Calendar.YEAR and method get in GregorianCalendar. The constant indicates that the method should return the current year. By comparing the returned value with the expYear field in DriverLicense, you can determine if the card is expired. The code below will retrieve the current year:

```
calendar.get(Calendar.YEAR)
```

1.8) The ID card and the phone card don't expire. What should you do to reflect this fact in your implementation?

1.9) Add a method getExpiredCardCount, which counts the number of expired cards in the billfold, to the Billfold class.

1.10) Write a BillfoldTester class that populates a billfold with a phone calling card and an expired driver license. Then call the getExpiredCardCount method. Run your tester to verify that your method works correctly.

1.11) Implement toString methods for the Card class and its three subclasses. The methods should print:

the name of the class
the values of all instance variables (including inherited instance variables)

Typical formats are:

```
Card[name=Edsger W. Dijkstra]
CallingCard[name=Bjarne Stroustrup][number=4156646425,pin=2234]
```

Write the code for your toString methods.

1.12) Implement equals methods for the Card class and its three subclasses. Cards are the same if the objects belong to the same class, and if the names and other information (such as the expiration year for driver licenses) match.

Give the code for your equals methods.

Submit your source code for Card, IDCard, CallingCard, DriverLicense, Billfold, and your test unit for marking along with your long answers for 1.6 & 1.8.

2) Start by building a package that contains the class files AClass, BClass, and ASubClass. All the classes you need for this assignment are already written and can be copied from the listings below. The package should contain a subpackage called temp that contains the class file CClass.

Inside AClass and ASubClass there is a method called addem which contains a number of lines that are commented out. Uncomment one line at a time. Write a comment under each line that describes why the line above it will or will not compile. Be sure to keep the import and package statements in the original code. Comment out any lines that won't compile in the final submission.

If you forget to code an access specifier on a class variable (that is, you don't code public, protected, or private), Java assigns the variable "package access" which opens it up to modification from any class in the same package as the class that contains the unmodified variable. Most programmers consider this a bad idea as it breaks the object-oriented principle of data encapsulation. This subtle effect occurs when you are forgetful and leave off the access modifier accidentally. For that reason, you need to be aware of this default behavior in case you are careless.

Explain how package access variables differ from protected access variables.

```
import temp.*;

public class AClass
{
   private int aprivate;
   protected int aprotected;
   public int apublic;
   int apackage;
   static int noAObjects = 0;

   /**
      Think about these declarations
   */
   BClass bobj = new BClass();
   CClass cobj = new CClass();

   /**
      Constructor for objects of class AClass.
   */
   public AClass()
   {
      // Initialize instance variables
      aprivate = 1;
      aprotected = 2;
      apublic = 3;
      apackage = 4;
      noAObjects ++;
   }

   public int addem()
   {
      //System.out.println(bobj.bprivate);
      //System.out.println(bobj.bprotected);
      //System.out.println(bobj.bpublic);
      //System.out.println(bobj.bpackage);
      //System.out.println(cobj.cprivate);
      //System.out.println(cobj.cprotected);
      //System.out.println(cobj.cpublic);
      //System.out.println(cobj.cpackage);
      return aprivate + aprotected + apublic + apackage;
   }
}
```

```
public class BClass
{
   private int bprivate;
   protected int bprotected;
   public int bpublic;
   int bpackage;

   /**
      Constructor for objects of class BClass.
   */
   public BClass()
   {
      // Initialize instance variables
      bprivate = 1;
      bprotected = 2;
      bpublic = 3;
      bpackage = 4;
   }

   public int addem()
   {
      return bprivate + bprotected + bpublic + bpackage;
   }
}
```

```
import temp.*;

public class ASubClass extends AClass
{
   private int asprivate;
   protected int asprotected;
   public int aspublic;
   int aspackage;

   /**
      Think about these declarations
   */
   BClass bobj = new BClass();
   CClass cobj = new CClass();

   /**
      Constructor for objects of class ASubClass
   */
   public ASubClass()
   {
      // Initialize instance variables
      asprivate = 1;
      asprotected = 2;
      aspublic = 3;
      aspackage = 4;
   }

   public int addem()
   {
      // System.out.println(bobj.bprivate);
      // System.out.println(bobj.bprotected);
      // System.out.println(bobj.bpublic);
      // System.out.println(bobj.bpackage);
      // System.out.println(cobj.cprivate);
      // System.out.println(cobj.cprotected);
      // System.out.println(cobj.cpublic);
      // System.out.println(cobj.cpackage);
      // System.out.println(aprivate);
      // System.out.println(aprotected);
      // System.out.println(apublic);
      // System.out.println(apackage);
      return adprivate + asprotected + aspublic + aspackage
         + aprotected + apublic + apackage;
   }
}
```

```
package temp;

public class CClass
{
   private int cprivate;
   protected int cprotected;
   public int cpublic;
   int cpackage;

   /**
      Constructor for objects of class CClass
   */
   public CClass()
   {
      // Initialize instance variables
      cprivate = 1;
      cprotected = 2;
      cpublic = 3;
      cpackage = 4;
   }

   public int addem()
   {
      return cprivate + cprotected + cpublic + cpackage;
   }
}
```

Submit your commented code for each of the classes to your lab instructor.

3) It is often the case that two or more classes share a common set of methods. For programming purposes we might wish to treat the objects of those classes in a similar way by invoking some of their common routines.

For example, the Dog and Cat classes below agree on the void method speak. Because Dog and Cat objects have the ability to "speak", it is natural to think of putting both types of objects in an ArrayList and invoking speak on every object in the list. Is this possible? Certainly we could create an ArrayList of Dog that would hold all the Dog objects, but can we then add a Cat object to an ArrayList of Dog?

Try running the main program below as it is written.

Run it a second time after uncommenting the line that instantiates a Cat object and tries to add it to the ArrayList.

```
import java.util.*;

```

```java
public class AnimalRunner
{
   public static void main(String[] args)
   {
      ArrayList<Dog> dogcatList = new ArrayList<Dog>();
      dogcatList.add(new Dog("Fred"));
      // dogcatList.add(new Cat("Wanda"));
   }
}
```

```java
public class Dog
{
   private String name;

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

   public void speak()
   {
     System.out.println("Woof! Woof!");
   }

   public String toString()
   {
      return "Dog:  " + name;
   }
}
```

```java
public class Cat
{
   private String name;

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

   public void speak()
   {
     System.out.println("Meow! Meow!");
   }

   public String toString()
   {
      return "Cat:  " + name;
   }
}
```

Our experiment to add Cat objects to an ArrayList of Dog objects failed. Perhaps we should try using the default Java ArrayList without generics? Try running the code below as it is written along with the Dog and Cat classes defined above. Run it a second time after uncommenting the line that invokes speak.

```
import java.util.*;

public class AnimalRunner
{
   public static void main(String[] args)
   {
      ArrayList dogcatList = new ArrayList();
      dogcatList.add(new Dog("Fred"));
      // dogList.add(new Cat("Wanda"));
      for (Object obj : dogcatList)
      {
         // obj.speak();
      }
   }
}
```

The experiment shows that we are now able to add Dog and Cat objects to the ArrayList, but there is a compile error on the line obj.speak because obj is an Object reference variable and the class Object doesn't contain a speak method. We need a reference variable that can refer to Dog and Cat objects and which also allows us to invoke speak. The solution to the problem uses interfaces.

First create an **interface** called Speakable that specifies a void speak() method. Be sure to modify the Dog and Cat classes to indicate that they implement the Speakable interface. For example, in the case of the Dog class, we will code

```
public class Dog implements Speakable
```

Be sure to make a similar change in the declaration of the Cat class.

The term Speakable can be used to create Speakable references. Using generics, create an ArrayList of Speakable objects in the main method. Modify the loop so that it iterates over Speakable objects. Try adding the Dog and Cat objects and invoking the speak method on each object. Check to see if this works.

Save and submit your finished code for the Dog, Cat, AnimalRunner, and Speakable classes.