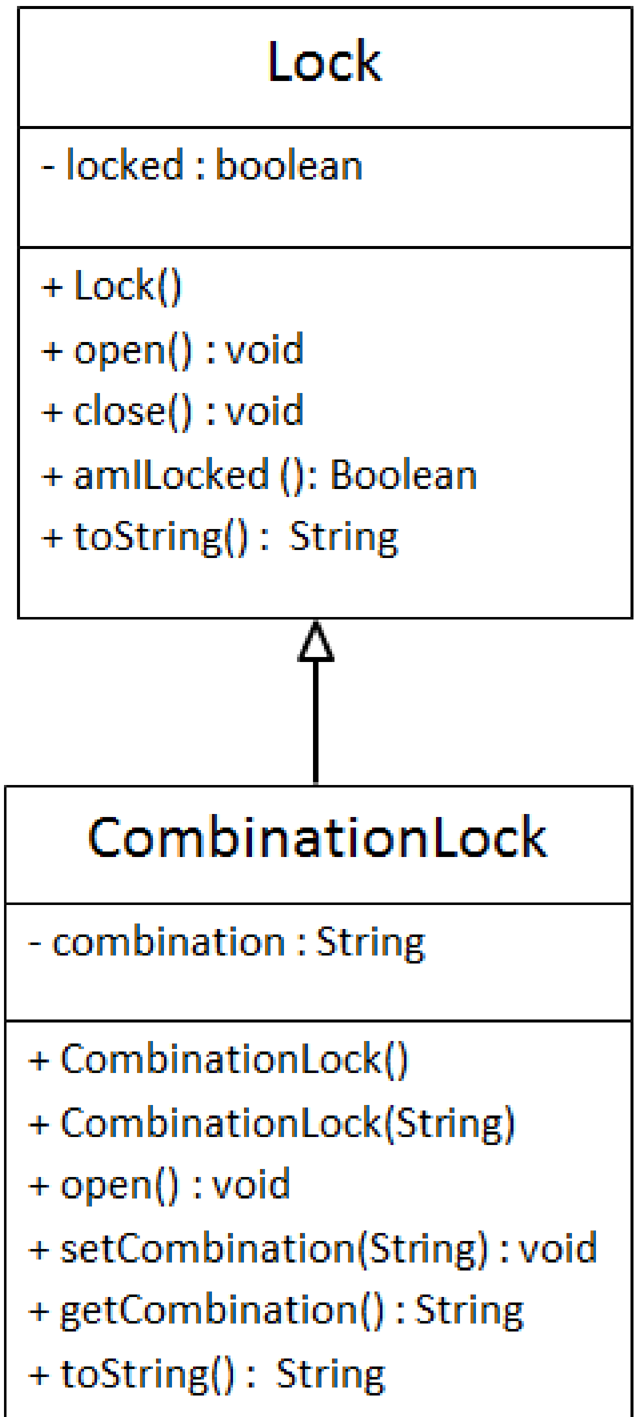
Design

There are many types of locks including combination locks, pad locks, and dead bolt locks, just to name a few. What do all these locks have in common? Answer - they are either in the state of locked or unlocked. Therefore we can design a class that represents the behavior that all locks possess, hence a more general description. A combination lock is a type of lock but it is more specific in how it state changes; namely entering a combination. Therefore we will design a second class that represents the behavior of a combination lock and its more specific description. Through inheritance the combination lock class will inherit the behavior of the general lock class.

UML Class Diagram

UML (Unified Modeling Language) is a pictorial language that allows you to describe a software system in visual way. A **class diagram** is a type of UML diagram that describes the attributes and operations of a class. A class diagram also shows the relationship between classes within a software system and it shows the responsibility each class has within the system.

Below is a class diagram for the Lock and CombinationLock classes.



Lock Class

Lock Class - Instance Variables

Here is the beginning implementation of the Lock class:

```
public class Lock
{
    // Instance Variable
    private boolean locked;



}
```

The Lock class contains one instance variable that represents the state of a lock. The variable is declared type boolean. Recall that a boolean variable can only have one of two possible states: true or false. This will work nicely with our class since a lock only has two possible states: unlocked or locked.

Lock Class - Mutator Methods

"What behavior do all locks have in common?" Answer - they can be locked or unlocked. Recall that behaviors in a class are represented by methods. So we will define two methods in our Lock class that will give us the ability to change the state of a lock to either locked or unlocked. Here is the modified implementation of the Lock class with these two methods added:

```
public class Lock
{
    // Instance Variable
    private boolean locked;

    // Mutator Methods
    public void open()
    {
        locked = false;
    }

    public void close()
    {
        locked = true;
    }
}
```

The two methods are named **open** and **close**. When a client sends a lock object the open message the locked state is changed to false which represents an open lock. If a lock object is sent a close message then the locked state is changed to true which represents a closed lock. Since both of these methods change the value of the instance variable they are called **mutator methods.**

Lock Class - Accessor Methods

It is often necessary to provide a client the ability to access or view the state of an object. We call these kinds of methods **accessor methods**. Since there is only one instance variable in the Lock class we will only define one accessor method. Often accessor method names are preceded with the prefix "get" but that is not a requirement. In this class, we will name our accessor method - **amILocked**. Here is its implementation:

```
public class Lock
{
    // Instance Variable
    private boolean locked;

    // Mutator Methods
    public void open()
    {
        locked = false;
    }

    public void close()
    {
        locked = true;
    }

    // Accessor Method
    public boolean amILocked()
    {
        return locked;
    }
```

Lock Class - toString Method

It is usually a good idea to implement a toString method for your class. Here is the implementation of the Lock class with a toString method added.

```
public class Lock
{
    // Instance Variable
    private boolean locked;

    // Mutator Methods
    public void open()
    {
        locked = false;
    }

    public void close()
    {
        locked = true;
    }

    // Accessor Method
    public boolean amILocked()
    {
        return locked;
    }

    // toString Method
    public String toString()
    {
        if(locked == false)
          return "Lock is open";
        else
          return "Lock is closed";
    }
}
```

Lock Class - Constructors

When we construct a lock object we must decide if we want the lock to start in an open or closed state. Recall that if you do not create a constructor for a class the compiler will create one for you and assign the instance variables their default values. The default value for a boolean variable is false. That means, if we leave the implementation of our constructor up to the compiler our lock object will start with an open state (locked = false).

Since I prefer that the lock start in a locked state (locked = true) we will create our own default constructor. Here is the completed Lock class with a default constructor included:

```
public class Lock
{
    // Instance Variable
    private boolean locked;

    // Default Constructor
    public Lock()
    {
      locked = true;  // lock starts in a closed state
    }

    // Mutator Methods
    public void open()
    {
        locked = false;
    }

    public void close()
    {
        locked = true;
    }

    // Accessor Method
    public boolean amILocked()
    {
        return locked;
    }

    // toString Method
    public String toString()
    {
        if(locked == false)
          return "Lock is open";
        else
          return "Lock is closed";
    }
}
```

We have finished the implementation of our Lock class. Copy the code from the completed Lock class above into a source file named **Lock.java**.

CombinationLock Class

The Lock class is now our base or super class. The next step is to derive a class named CombinationLock from our base class. This class will inherit the behavior of the Lock class but also extend its behavior to include a combination.

1. ### CombinationLock - Instance Variables

   We start by defining its instance variables. To help us decide what variables are needed we ask ourselves this question, "What are the states of a combination lock?" Well, to start, a combination lock is either locked or unlocked. But since we are going to inherit this state from our superclass, Lock, we do not need to include it in our definition. However, a state that is specific to a combination lock is the combination itself. In order for a combination lock to be opened, it must know its combination. In our implementation we declare an instance variable of type String named combination. Here is the beginning of our CombinationLock class:

   ```
   import java.util.*;  // needed for Scanner

   public class CombinationLock extends Lock
   {
      // Instance Variables
      private String combination;



   }
   ```

   Copy the following code in a source file named **CombinationLock.java**.

2. ### CombinationLock Class - open Method

   We now turn to the specific behavior of a combination lock. The next question to be asked is, "What behavior or action does a combination lock perform?" Answer - It locks and unlocks. It turns out that this is the same behavior that is performed by our Lock class in the two methods open and close. Since both of these methods were inherited from the Lock class, we must now decide if there behavior needs to be modified in order to more clearly reflect the behavior of a combination lock. Let's start with the open method. How do you open a combination lock? Answer - you enter the combination. Since this is not the action performed by a general lock, we will need to override the open method to include this behavior.

   Override the **open** method using the following pseudocode:

   ```
   prompt user to enter combination
   input COMBO
   if COMBINATION = COMBO then
      super.open()
   ```

Notice the use of the keyword **super**. If the combinations match the method needs to unlock the lock. This happens to be the behavior of the open method in the Lock class. The super keyword allows a method to call the method it is overriding in the super class.

3. CombinationLock Class - close Method

We now consider the implementation of the close method. The close method in the Lock class has the same behavior that should be used in the CombinationLock class. Therefore, we do not need to override the close method, we will just use the one inherited from the Lock class.

4. CombinationLock Class - toString Method

The CombinationLock class inherts the toString method from the Lock class, but it does not include information about the combination instance variable. Therefore, we will override the toString method. Note: Normally you would not want to include code that allows a client access to a password or combination but since we are attempting to learn some of the principles of inheritance we will go ahead and overlook that here. Here is its implementation:

```
public String toString()
{
    String str = super.toString() + "\n" +
                 "Combination = " + combination + "\n";
    return str;
}
```

Notice the statement super.toString(). Since the toString method of the Lock class already returns a string that includes information about an object's locked state, this is the most efficient way to include this information in the overridden toString method.

Add the code for the **toString** method to the CombinationLock class.

5. Mutator and Accessor Methods

Add a mutator and access method to the class for the instance variable combination named **setCombination** and **getCombination respectively**.

6. Constructors

We are going to add two constructors to our CombinationLock class. The first will be a default constructor with no parameters and the second will contain one parameter.

The default constructor will initialize the combination variable to a null string "". Something else we must consider is how does the variable **locked** inherted from the Lock class get its initial value. We can access this variable by using the keyword super to invoke the Lock class's constructor. This operation must be the first line in the subclasses constructor. Here is the implementation of the default constructor including a call to Lock class's constructor:

```
public CombinationLock()
{
   super();    // call the default constructor of the Lock class
   combination = "";
}
```

Add the **default constructor** code to the CombinationLock class.

A second constructor will allow a user to assign a combination to the lock at the time it is instantiated. Again, we will use the keyword super to call the constructor in the Lock class. Here is its implementation:

```
public CombinationLock(String combo)
{
    super();
    combination = combo;
}
```

The call to super() is not really needed here. By default if you do not call a superclass's constructor in the subclass constructor, the compiler will call the default constructor of the superclass for you. However, if you wish to call a constructor in the superclass that is not the default constructor, then you must explicitly call the constructor yourself using super.

Example:

```
super(n);
```

Add the code for the **second constructor** to the CombinationLock class.