Lab Questions
Big Java, Late Objects / Java for Everyone, 2e
Chapter Number: 8 Objects and Classes

1.1) Object-oriented languages like Java are designed to make it easy for programmers to implement software versions of real-world objects. In learning Java, an important skill to master is the ability to represent an object in code. Objects that we model are described using Java classes, so we have chosen to begin this lab by modeling a very simple, everyday object: a door.

Write the code to create a class that models a door object. Don't worry about the internal details just yet. Just give the class a name and an empty body for the moment. We will add more to the class shortly.

1.2) When modeling an object as a class, we also need to describe the properties it possesses. An everyday object that we wish to model always has one or more properties that describe it. For instance a door object might have a name like "Front" or "Side" to distinguish it from other doors. Another property that could describe a door is its state: "open" or "closed". Properties of objects are described in code by using nouns like "state" or "name" to create instance variables that hold values.

Add instance variables to your `Door` class for the name of the door and its state. Experience has shown that we almost always want to limit the visibility of instance variables inside the same class, so make the access modifiers of `state` and `name` private. And because the `state` and `name` properties have values like "open" or "front", let the instance variables you create be of type `String`.

1.3) Objects also have operations which can be invoked on the object and which may change the object in some way. For instance, the operations for a door object could be "open" and "close". An operation on an object corresponds to a Java method and is described in code by using a verb like "open" or "close". Invoking a method may change the value of an instance variable. For example, invoking `close()` would change the value of the `state` variable from "open" to "closed".

Declare methods for `open` and `close`. Because we usually want to allow free access to the methods a class contains, make the access modifier for each method `public`.

1.4) Now that we have a `Door` class, we would like to be able to create some `Door` objects. Java constructors are components of a class whose purpose is to create objects of the given class and to initialize the object's instance variables. Java provides a constructor with no arguments (the "default" constructor) for every class and the `Door` class is no exception. Unfortunately, the default constructor that Java provides initializes the `state` and `name` variables to null. This is unacceptable.

Add a constructor for the `Door` class that receives two arguments: the name of the door and its initial state. Because we want to use the constructor outside of the class, make the access modifier for the constructor `public`.

1.5) It is often convenient to have accessor methods that operate on a single instance variable. Here is an accessor method for the `name` variable:

```
public String getName()
{
   return name;
}
```

The word `String` in front of `getName()` indicates that the method returns a `String` when it is invoked. The body is simple and just returns the value of `name`.

Add this method to your class and write a similar accessor method for the instance variable `state`.

1.6) Many instance variables in a class will have corresponding mutator methods that allow you to change the value of the variable. Here is a mutator method for the `name` variable:

```
public void setName(String newName)
{
   name = newName;
}
```

The word `void` in front of `setName()` indicates that the method does not return a value when it is invoked. The body is simple and copies the value of the parameter variable `newName` to instance variable `name`.

Add this method to the class and write a similar mutator method for the instance variable `state`.

1.7) If you are using BlueJ, use the interactive features of the IDE to create a `Door` object. Verify that it has the correct values for `name` and `state`. Invoke the `setState` mutator method on the `Door` object and verify that the value of the object's `state` was changed. Repeat the process for the `setName` mutator. Unit test the code you have written until you are convinced it is correct.

If you are not using BlueJ, compile and run the code below:

```
/**
   A class to test the Door class.
*/
public class DoorTester
{
   /**
      Tests the methods of the Door class
      @param args not used
   */
   public static void main(String[] args)
   {
      Door frontDoor = new Door("Front", "open");
      System.out.println("The front door is " + frontDoor.getState());
      System.out.println("Expected:  open");
```



```
      Door backDoor = new Door("Back", "closed");
      System.out.println("The back door is " + backDoor.getState());
      System.out.println("Expected:  closed");
      // Use the mutator to change the state variable
      backDoor.setState("open");
      System.out.println("The back door is " + backDoor.getState());
      System.out.println("Expected:  open");
      // Add code to test the setName mutator here
   }
}
```

Create a third `Door` object called "sideDoor" with the name property "Side" and an initial state of "closed". Verify that the object was properly created. Use the mutator to change the state of object `sideDoor` to "open". Verify that the mutator is working.

1.8) Consider the variable `state` in the class `Door` and the variable `newState` in the mutator for `state`. What kind of variable is `state`? What kind of variable is `newState`? When do these variables exist?

1.9) Consider the line below which was taken from the `main` method in Lab 8.1.7 above.

```
backDoor.setState("open");
```

What is the implicit parameter that is passed by this method call? What is the explicit parameter?

2.1) In this lab, you will implement a *vending machine* that holds cans of soda. To buy a can of soda, the customer needs to insert a token into the machine. When the token is inserted, a can drops from the can reservoir into the product delivery slot. The vending machine can be filled with more cans. The goal is to determine how many cans and tokens are in the machine at any given time.

What methods would you supply for a `VendingMachine` class? Describe them informally.

2.2) Now translate those informal descriptions into Java method headers, such as

```
public void fillUp(int cans)
```

Give the names, parameter variables, and return types of the methods. Do not implement them yet.

2.3) What instance variables do the methods need to do their work? *Hint:* You need to track the number of cans and tokens. Declare them with their type and access modifier.

2.4) Consider what happens when a user inserts a token into the vending machine. The number of tokens is increased, and the number of cans is decreased. Implement a method:


```
public void insertToken()
{
   // Instructions for updating the token and can counts
}
```

You need to use the instance variables that you defined in the previous step.

Be sure to check that the number of cans is greater than zero before decreasing the can count and increasing the token count.

2.5) Now implement a method `fillUp(int cans)` to add more cans to the machine. Simply add the number of new cans to the can count.

2.6) Next, implement two methods, `getCanCount` and `getTokenCount`, that return the current values of the can and token counts.

2.7) You have implemented all methods of the `VendingMachine` class.
Put them together into a class, like this:

```
public class VendingMachine
{
   private your first instance variable
   private your second instance variable

   public your first method
   public your second method
   . . .
}
```

2.8) Now complete the following tester program so that it exercises all of the methods of your class.

```
public class VendingMachineTester
{
   public static void main(String[] args)
   {
      VendingMachine machine = new VendingMachine();
      machine.fillUp(10); // Fill up with ten cans
      machine.insertToken();
      machine.insertToken();
      System.out.print("Token count: ");
      System.out.println(machine.getTokenCount());
      System.out.println("Expected: . . .");
      System.out.print("Can count: ");
      System.out.println(machine.getCanCount());
      System.out.println("Expected: . . .");
   }
}
```


2.9) So far, the `VendingMachine` class does not have any constructors. Instances of a class with no constructor are always constructed with all instance variables set to zero (or `null` if they are object references). It is always a good idea to provide an explicit constructor.

Provide two constructors for the `VendingMachine` class:
a) A constructor with no arguments that initializes the vending machine with 10 soda cans
b) A constructor, `VendingMachine(int cans)`, that initializes the vending machine with the given number of cans

Both constructors should initialize the token count to 0.

3.1) Consider the following task: You are on vacation and want to send postcards to your friends. A typical postcard might look like this:

```
Dear Sue: I am having a great time on
the island of Java. The weather
is great. Wish you were here!
Love,
Janice
```

You decide to write a computer program that sends postcards to various friends, each of them with the same message, except that the first name is substituted to match each recipient.

What "black box" (class that will be used to build objects of its type) can you identify that will allow you to send your greeting?

3.2) We want to be able to write a program that will use our `Postcard` class to send postcards with the same message to different recipients.

The following class implements a `Postcard`. Notice that we do not set the recipient in the constructor because we want to be able to change the recipient, and keep the same message and sender. What method would you add to support this functionality? Implement the method.

```
public class Postcard
{
   private String message;
   private String sender;
   private String recipient;

   public Postcard(String aSender, String aMessage)
   {
      message = aMessage;
      sender = aSender;
      recipient = "";
   }
```



```
   // Your method here
}
```

3.3) Add a `print` method to the `Postcard` class to display the contents of the postcard on the screen. Use the instance variables `message`, `sender`, and `recipient` defined in the last step when you implement the method.

3.4) Try out your class with the following code:

```
public class PostcardPrinter
{
   public static void main(String[] args)
   {
      String text = "I am having a great time on\nthe island of Java. Weather\nis great.
Wish you were here!";
      Postcard postcard = new Postcard("Janice", text);
      postcard.setRecipient("Sue");
      postcard.print();
      postcard.setRecipient("Tim");
      postcard.print();
   }
}
```

What is the output of your program?

4) Using the `Person` and `PersonRunner` classes below, execute the `main` method in `PersonRunner`. Why does the program end abnormally? Comment out the first two lines and run the program again. Why does the reference to `p3` cause Frank's name to be printed when we set Frank's name with the `p2` reference?

```
public class PersonRunner
{
   public static void main(String[] args)
   {
      Person p1 = null;
      p1.speak();
      Person p2 = new Person("Beth");
      p2.speak();
      Person p3 = p2;
      p2.speak();
      p2.setName("Frank");
      p3.speak();
   }
}

public class Person
{
   private String name;

   public Person(String name)
   {
```



```
      this.name = name;
   }
   public void setName(String name)
   {
      this.name = name;
   }
   public String getName()
   {
      return name;
   }
   public void speak()
   {
      System.out.println("My name is " + name);
   }
}
```

4.2) Modify the `Person` and `PersonRunner` classes from Lab 8.4.1. Add a static `int` variable `count` in the `Person` class. Increment the `count` field each time a `Person` object is instantiated. Add a static method in `Person` called `printCount()` that prints a message that indicates the number of `Person` objects that have been created. Modify the main program so that it creates four `Person` objects and then invokes `printCount()`.

