Due: Monday, October 24, 2016 by the end of lab

Goals:

By the end of this activity you should be able to do the following:

- Implement interfaces
- Overload methods

Directions:

Part 1: Customer.java (70%)

- Open a new Java file in jGRASP and create a class called Customer. Create three instance variables:
  - String object for the customer's name
  - String object for the customer's location (town)
  - double to store the customer's balance

- Create a constructor for the Customer class that takes the Customer's name as a parameter. **Set the location variable to an empty string and the balance to zero**. You do not have to include the comments.

```
public Customer(String nameIn) {
    _____________ = nameIn; // sets name to parameter input
    _____________ = ""; // sets location to empty string
    _____________ = 0; // sets balance to 0
}
```

- Create methods that set the customer's location, change the customer balance by an amount specified by a double parameter, get the location, and get the balance. That is, provide method bodies for the method headers below. Be sure to compile the completed methods before using them in interactions.

```
public void setLocation(String locationIn) // sets location variable
public void changeBalance(double amount) // add amount to balance
public String getLocation() // returns variable for location
public double getBalance() // returns variable for balance
```

- Try the following example in the interactions pane (you can leave out the comments):

```
▶ Customer cstmr = new Customer("Lane, Jane");
▶ cstmr.changeBalance(30); // add $30 to balance
▶ cstmr.getBalance()
  30.0
▶ cstmr.changeBalance(-5); // take $5 off balance
▶ cstmr.getBalance()
  25.0
▶ cstmr.setLocation("Boston, MA");
▶ cstmr.getLocation()
  Boston, MA
```

- Suppose you want to be able to set the customer location with city and state in a single String or by entering city and state in two separate String parameters. **Overload** the setLocation method so that it takes two String parameters (do not delete the original setLocation method; rather create a second constructor).

```
public void setLocation(String city, String state) {
    location = city + ", " + state;
}
```

- Now when the setLocation method is included in your code, the compiler will check to see whether you have one String parameter or two String parameters. It will then **bind the method call** with the appropriate **declaration** of the setLocation method. Try entering the following code into the interactions pane (recompile your program first):

```
Customer cstmr = new Customer("Lane, Jane");
cstmr.setLocation("Boston, MA")
cstmr.getLocation()
Boston, MA
cstmr.setLocation("Omaha", "NE")
cstmr.getLocation()
Omaha, NE
```

- Create a toString method that shows the customer's name, their location, and their balance (you do not have to format balance).

```
Customer cstmr = new Customer("Lane, Jane");
cstmr.setLocation("Boston, MA")
cstmr.changeBalance(5)
cstmr
Lane, Jane
Boston, MA
$5.0
```

Part 2: Implementing the Comparable Interface (30%)

- Suppose that you wanted to be able to compare Customer objects based on some attribute. You can **implement** the Comparable **interface** in your customer class by indicating this in the class header as shown below:

```
public class Customer implements Comparable {
```

- The Comparable interface has a method called compareTo that takes another object and compares this object to the parameter based on some value. You want to sort customers based on their balance, so the compareTo method is defined as follows:

```
public int compareTo(Object obj) {
   // cast the incoming object as a Customer
   Customer c = (Customer)obj;
   if (this.balance > c.getBalance()) {
      return 1;
   }
   else if (this.balance ___ c.getBalance()) {
      return -1;
   }
   else {
      return 0;
   }
}
```

- Write a main method (either in Customer or a separate driver program) that creates two customer objects and then uses a **conditional statement** to print which object has a higher balance or that they are equal, based on the return value of the compareTo method. When composing the boolean expression for the if statement, keep in mind that by convention the decision should be based on whether the return value for compareTo is less than 0, greater than 0, or equal to 0 (i.e., you should not expect it to be exactly -1 or 1).

```
public static void main(String[] args) {
   Customer cstmr1 = new Customer("John");
   cstmr1.changeBalance(10);
   cstmr1.setLocation("Boston, MA");
   System.out.println(cstmr1);

   Customer cstmr2 = new Customer("JoAnn");
   cstmr2.changeBalance(73);
   cstmr2.setLocation("Auburn, AL");
   System.out.println(cstmr2 + "\r\n");

   // use the compareTo method in the if statement below
   if (_________________) {
      System.out.println("Higher balance: " + cstmr1);
   }
   else if (_________________) {
      System.out.println("Higher balance: " + cstmr2);
   }
   else {
      System.out.println("Balances are equal.");
   }
}
```

- Use a generic type with the Comparable interface. Suppose you wanted to ensure that only Customer objects are compared (sending in a String object right now would cause a run-time error):

```
public class Customer implements Comparable<Customer> {
```

- Modify the compareTo method to use the Customer object directly without casting:

```
public int compareTo(Customer obj) {
    if (this.balance > obj.getBalance()) {
        return 1;
    }
    else if (this.balance ___ obj.getBalance()) {
        return -1;
    }
    else {
        return 0;
    }
}
```

- Execute the main method to ensure that the output is still correct.