Goals:

By the end of this activity you should be able to do the following:
  ➢ Understand and implement static variables and methods
  ➢ Be able to create, compile, and run a JUnit test file

Description:

For this assignment, you will need to download BankLoan.java from the course website and save it to an appropriate folder.

Directions:

Don't forget to add your Javadoc comments for your classes, constructor, and methods in this activity.

Part 1: Static methods

- You don't have to read this bullet now, but make sure that you understand it before this week's quiz and project. Three of the properties of static methods:
  o Static methods can be invoked using the name of the class.
  o Static methods can be invoked before an object of the class is instantiated.
  o Static methods cannot access instance data. This is because the method can be called before an object is created, which means no instance data would exist. Consider the following examples:
    ▪ The trim method of the String class is an instance method. You must create a String object before you call the method so that the method knows what to trim:

      ```
      String s = "   Red Sox";
      System.out.println(s.trim());
      Red Sox
      ```

    ▪ The pow method of the Math class has parameters for the values it needs so it is defined as static. This means that you don't have to go through the additional step of creating a Math object before you use the method:

      ```
      System.out.println(Math.pow(2, 3));
      8.0
      ```

  If a method that you create doesn't need direct access to instance variables or instance methods in that class, then you should consider making the method static.

- Suppose that you want to add a method to BankLoan that returns true if a loan amount is valid and false otherwise. If the loan amount (a double) is taken as a parameter, then the method would not need direct access to instance data. Create the following method header:

  ```
  public static _____ isAmountValid(_____ amount)
  ```

  Add code to the method to return true if the amount is greater than or equal to 0 and false otherwise:

  ```
  return amount ____ 0;
  ```

- Make sure that BankLoan.java compiles after you add your isAmountValid method. It will be tested later in this activity.
- Now suppose that you need a method that returns true if a loan's balance is greater than 0 and false otherwise. The user will pass in a BankLoan object as a parameter, and the method will only access this object, which is considered a local variable. Thus, the method can be static.

```
public static _____ isInDebt(_____ loan) {
```

- Now add an if statement that returns true if the specified object has a balance greater than 0:

```
if (_____.getBalance() > 0) {
    return true;
 }
  return _____;
```

- Compile your program and test it in the interactions pane:

```
BankLoan b = new BankLoan("Bob", 0.08);
BankLoan.isInDebt(b)
false
b.borrowFromBank(1); // borrow $1.00
BankLoan.isInDebt(b)
true
```

Part 2: Static variables as fields – a.k.a. <u>class variables</u> as opposed to <u>instance variables</u>

- You don't have to read this bullet now, but make sure that you understand it before this week's quiz and project. Properties of static variables / constants:
  - Public constants (which are static and final) can be accessed using the name of the class if they are public.
  - Public static constants can be accessed before an object of the class is instantiated.
    ```
    System.out.println(Math.PI);
    3.141592653589793
    ```
  - Public static variables (not declared as final) violate encapsulation. Private static variables should be accessed and modified through static <u>or</u> instance methods.
  - A static variable is <u>shared among all instances of the class</u> and it can be accessed / modified by any method in the class. **Why are constants declared as static as well as final?**

  If one object modifies a static variable, then it will be modified for all other objects as well because they are all accessing the same variable (i.e., <u>a class variable</u> is shared). If a variable needs to be specific to one object (e.g., each object need its own customer name field), then it should an instance variable (not static) rather than a class variable (static).

- Add a static variable to the BankLoan class that will count the number of BankLoan objects that have ever been instantiated in a program. The value will be initialized to 0 and then be incremented each time a BankLoan object is created.

```
private static int loansCreated = 0;
```

- You want to increment the variable only when an object is instantiated, so you'll need to add the following code to the constructor:

```
loansCreated++;
```

- You'll also need to add a method to access the variable. The method will only access a static variable, so it can and should be static as well.

```
public static _____ getLoansCreated() {
   return loansCreated;
}
```

- Also add a method that will reset the number of loans created to 0. The method only modifies a static variable, so it can be static and there is no return type.

```
public static _____ resetLoansCreated() {
   _____ = _____;
}
```

Test your class in the interactions pane:

```
BankLoan.getLoansCreated()
0
BankLoan jane = new BankLoan("Jane L", 0.09);
BankLoan bob = new BankLoan("Bob S", 0.09);
```

Now open a canvas window (click 🖼 on the Debug tab), then drag *jane* and *bob* onto the canavas. Unfold *jane* in the Debug tab, and drag *loansCreated* onto the Canvas. Figure 1 shows the initial canvas. Notice that *loansCreated* is shown as *BankLoan.loansCreated* since it is a class variable rather than an instance variable.

```
BankLoan.getLoansCreated()
2
bob = new BankLoan("Bob Parker", 0.02);
```

Figure 2 shows the canvas after *bob* has been assigned a new loan and the loansCreated field has been updated. That is, we have now called the *new* operator a total of three times.

```
BankLoan.getLoansCreated()
3
BankLoan.resetLoansCreated();
BankLoan.getLoansCreated()
0
```
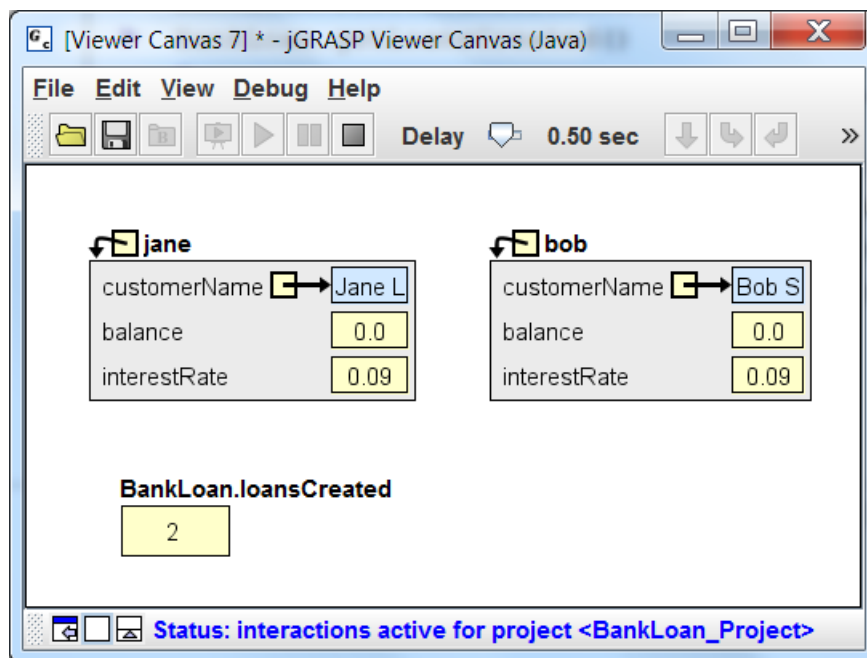
Figure 1. Canvas after *jane* and *bob* are created
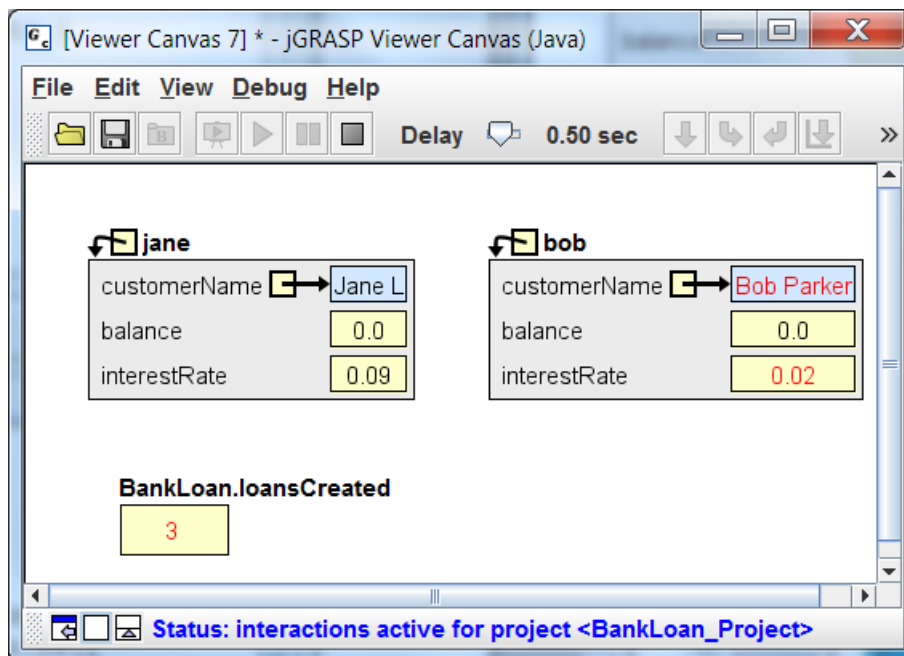
Figure 2. Canvas after *bob* is reassigned to a *new* BankLoan

**Part 3. JUnit Test Files** – **NOTE: In order to use JUnit on you own computer, you must download and unzip JUnit (as you did Checkstyle), and then JUnit must be configured in jGRASP (Tools > JUnit > Configure). Note that the following example will work with the unmodified BankLoan.java file.**

Now let's create a JUnit test file to allow us to test (and retest) the method in a class. Suppose we want to test the chargeInterest() method. We'll need to create a test method that essentially includes the statements that we would enter as interactions to informally test the chargeInterest() method.

- For example, we could test this method using interactions by entering the following:

  ```
  BankLoan loan1 = new BankLoan("Jane", .10);
  loan1.borrowFromBank(1000.00);
  loan1.chargeInterest()
  loan1.getBalance()
  1100.0
  ```

- Create a project file for the BankLoan class. After you have created the project and added BankLoan.java to it, you should see the following in the Open Projects section of the Browse tab.
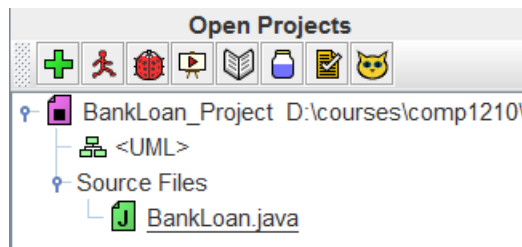
- With BankLoan.java open in the CSD Window, on the Desktop menu click on the Create JUnit Test File button 📄.
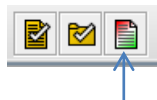
- This creates a JUnit test file name BankLoanTest.java that is listed in the "Test File category in the Project tab.
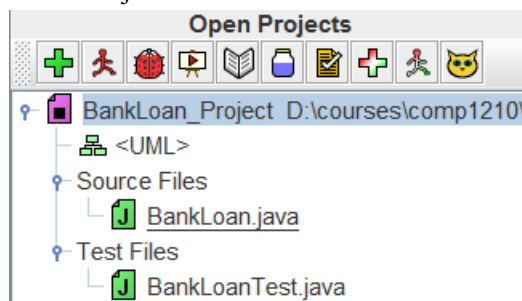
- BankLoanTest.java contains a class with the same name. The class contains a setup() method and test method named defaultTest() that asserts that 0 equals 1 which will always <u>fail</u>. This is an example test method that you should comment out, modify, or delete.

```java
/** A test that always fails. **/
@Test public void defaultTest() {
    Assert.assertEquals("Default test added by jGRASP. Delete "
                + "this test once you have added your own.", 0, 1);
}
```

- Either modify the method above or create a new test method as follows. <u>If you create a new method, be sure to delete or comment out the default test method</u>. Note that we won't be using the setup() method, so it can be deleted or left as is since it has an empty body.
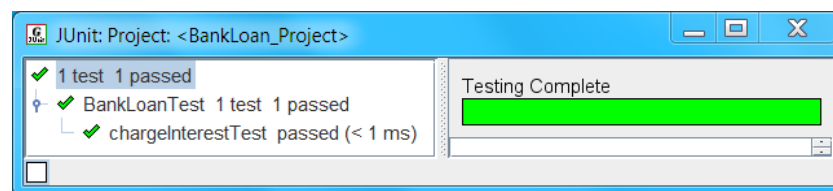
```java
@Test public void chargeInterestTest() {
    BankLoan loan1 = new BankLoan("Jane", .10);
    loan1.borrowFromBank(1000.00);
    loan1.chargeInterest();
    Assert.assertEquals(" ", 1100, loan1.getBalance(), .000001);
}
```

Note that the third parameter is the "delta" required when two doubles values are compared for equality (i.e, how close do the values have to be to be considered equal; in this case, we have chosen 0.000001).

- After you have entered your test method, you can compile ✛ the test file or you can compile and run ⚘ the test file using JUnit buttons on the Project tab or on desktop menu. You should make sure BankLoanTest compiles successfully before you run it.

- A successful test will have output such as the following output in the Run I/O tab and in the JUnit results window

```
Runing 1 JUnit test.

Completed 1 tests  1 passed
```


- To test other methods in the BankLoan class, you simply create additional @Test methods in BankLoanTest.java using the procedure above. Remember creating a test method is only slightly more effort than manually testing your method in interactions the first time. In fact, many of the same statements used in interactions are used in the test method. These are then followed by an appropriate JUnit assert statement. As you create new methods in a class, you can also create appropriate test methods in the corresponding test class. This helps ensures each new method is correct before you proceed to the next method. Being able re-run all test files with a single click is a huge time saver as you build your program incrementally and make changes during development.

- If a test method fails, set a breakpoint at or above the assert that failed, then run the test file using the test file debugger 🌐 button. When the program stops at the breakpoint, drag variables onto the canvas and examine the values. Hopefully, you'll be able to see the problem and correct the error. Remember that if a test method fails, either the method you are testing has an error or the test method itself has an error. <u>You may need to *step in* at the statement that calls the method you are testing. For example, *stepping in* at `loan1.chargeInterest()` will take you into the `chargeInterest` method where you can step through it looking for a problem.</u> Figure 3 shows the jGRASP desktop after stopping at the breakpoint and then stepping one time to create *loan1*. Figure 4A shows the canvas after loan1 has been dragged onto it but before any methods have been called on *loan1*. Figure 4B shows the canvas after calling the borrowFromBank and chargeInterest methods on *loan1*.
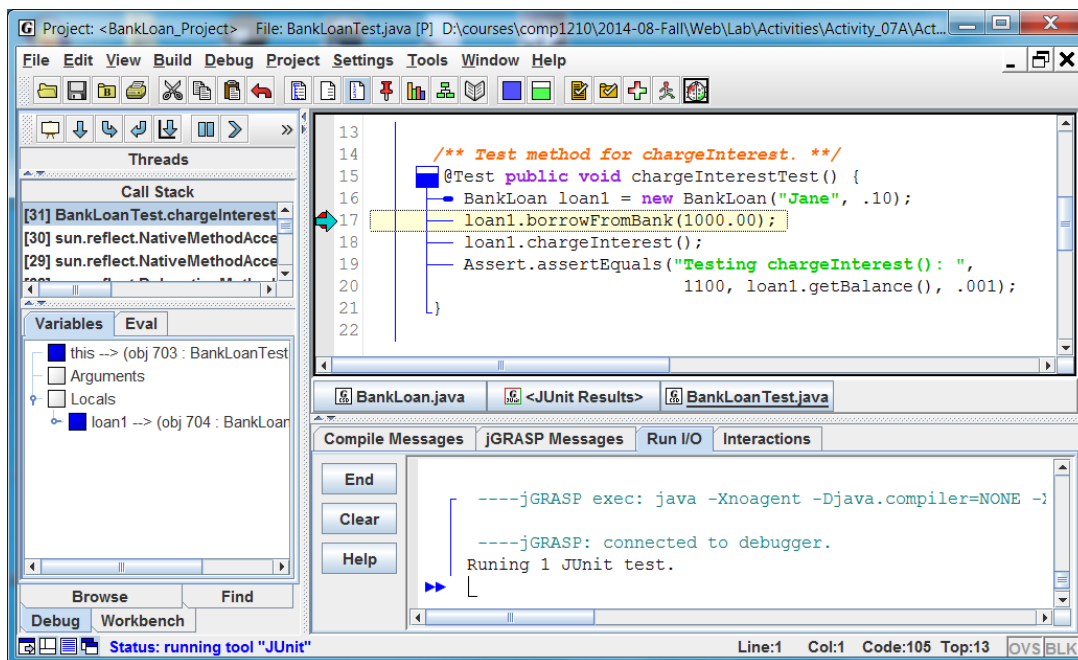

Figure 3. Running BankLoanTest in the Debugger, after stopping at the breakpoint and then stepping one time to create *loan1*.
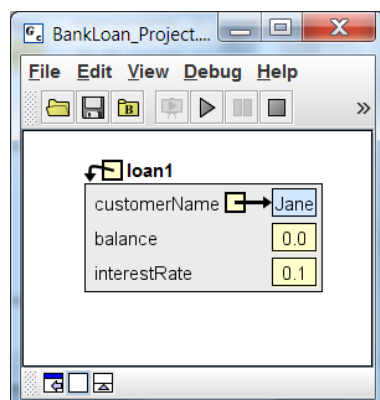

Figure 4A. Canvas with loan1 after it was created but before calling borrowFromBank.
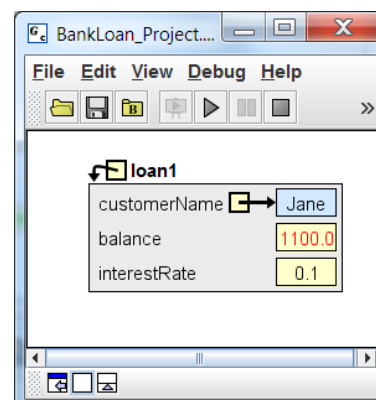

Figure 4B. Canvas after calling the borrowFromBank and chargeInterest methods.

Finally, submit your BankLoan.java and BankLoanTest.java files to the grading system.