

Due Part 1: Monday, February 8, 2016 by the end of the lab

Part 2: Wednesday, February 10, 2016 by the end of the lab (or demo Part 2 Mon, Feb)

Reading Chapter 4.1 – 4.5, pages 160-181

Terminology

attribute / state	UML class diagram	calling method
behavior	encapsulation	method declaration
class	client	method invocation
method header	visibility (or access) modifier	return statement
class header	accessor method	parameters
instance variable	mutator method	constructor

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


- Create a class with methods that accept parameters and return a value
- Understand the constructor and the toString method of a class
- Create a class with main that uses the class created above
- Create a jGRASP project containing all of the classes for your program
- Generate a UML class diagram and documentation
- Create a jGRASP canvas for your program and run the program in the canvas

Description

In this activity you will create two classes: one called UserInfo and another called Activity4A which contains a main method that uses the first class (i.e., Activity4A creates instances of UserInfo and calls its methods). You will also create a project, UML class diagram, and canvas for the program.

Part 1: Create and test the UserInfo class.

- Create your UserInfo class and add the following comments (there will be no main method in this class):

```
 public class UserInfo {  
    // instance variables  
  
    // constructor  
  
    // methods  
}
```

- **Instance variables (or fields) represent the information that an object of the class needs to store.** Declare the following variables after the comment: `// instance variables`
 - firstName: a **String** for the user's first name
 - lastName: a **String** for the user's last name
 - location: a **String** for the user's location
 - age: an **int** for the user's age
 - status: **int** indicates whether the user is online or offline

Hint: Use the *private* access modifier so that values cannot be directly accessed from outside of the object; for example, declare firstName as follows:

```
private String firstName;
```

- **Constants store values that cannot be changed.** Use constants to represent the two possible statuses of the user.

```
private static final int OFFLINE = 0, ONLINE = 1;
```
- **The constructor is used in conjunction with the *new* operator to create an instance of `UserInfo` and initialize the fields.** It should be placed right below the fields and before the methods in the class; that is, right after the comment `// constructor` in your class. The constructor *does not have a return type*, and always has the *same name as the class*. The `UserInfo` constructor will take two parameters as input representing the first and last name of the user. The names for these parameters can be anything; however, many professional programmers use the convention of adding the suffix “In” to the corresponding field name as shown below.

```
public UserInfo(String firstNameIn, String lastNameIn){
    }
}
```

In the constructor, store the first and last name in the appropriate instance variables:

```
    firstName = firstNameIn;
    lastName = lastNameIn;
```

Because you do not have inputs for age, location, and status, you will need to set those to default values:

```
    location = "Not specified";
    age = 0;
    status = OFFLINE;
```

Compile your program, click the Interactions tab, and create a new `UserInfo` object called *u* and display it by entering the following in Interactions tab.

```
▶ UserInfo u = new UserInfo("Jane", "Lane");
▶ u
  UserInfo@3da3da69
▶
```

Note that the odd looking value for *u* is not very useful. When we print or display *u*, we usually want to see some text that tells us something about the object. In the Workbench tab, find *u* and unfold it to see its fields. We can see that the fields in *u* do have the values that we set with constructor. We need to add a *toString* method to our `UserInfo` class.

```
u --> (obj 420 : UserInfo) UserInfo
├── firstName --> "Jane" (obj 409 : java.lang.String) private java.lang.String
├── lastName --> "Lane" (obj 410 : java.lang.String) private java.lang.String
├── location --> "Not specified" (obj 421 : java.lang.String) private java.lang.
├── age = 0 : private int : declared in UserInfo
├── status = 0 : private int : declared in UserInfo
├── OFFLINE = 0 : private final static int : declared in UserInfo
└── ONLINE = 1 : private final static int : declared in UserInfo
```

- **The `toString` method returns a `String` representation of the object.** Create and return a local `String` variable called `output` that contains information about the `UserInfo` object. Place this method after the methods comment: `// methods`

This method returns a `String`.

```
public String toString() {  
    String output = "Name: " + firstName + " "  
        + lastName + "\n";  
    output += "Location: " + location + "\n";  
    output += "Age: " + age + "\n";  
    output += "Status: " + status;  
    return output;  
}
```

After you have successfully compiled your class, click the Interactions tab, and create a new `UserInfo` object called `u` and display it by entering the following in Interactions tab as you did above. This time, when `u` is evaluated, the `toString` method is implicitly invoked on `u` and the return value is displayed.

```
UserInfo u = new UserInfo("Jane", "Lane");  
u  
Name: Jane Lane  
Location: Not specified  
Age: 0  
Status: 0
```

In the example above, you can also display `u` by entering the print statement in interactions:
`System.out.println(u);`

When the `u` is entered without a semi-colon, it is an expression rather than a statement. You could also enter `u.toString()` as an expression. Remember, in the Interactions tab, expressions are evaluated and the result is immediately displayed. Since `u` is an object, it “evaluates” to the result of calling its `toString` method.

- **The methods of your class describe what your object can do (the behaviors of an object).** For a `UserInfo` object, we want to be able to set the `location` and `age` of the user and also to change the status. First, create a set method for location:

Method does not return a value.

```
public void setLocation(String locationIn) {  
    location = locationIn;  
}
```

After compiling your class, test it in the interactions pane:

```
UserInfo u = new UserInfo("Jane", "Lane");  
u.setLocation("Auburn");  
u  
Name: Jane Lane  
Location: Auburn  
Age: 0  
Status: 0
```

Add a method to set the value of age. This will only change age if the age is greater than 0, and it will return a boolean value (true or false) indicating whether the age was set:

```
public boolean setAge(int ageIn) {
    boolean isSet = false;
    if (ageIn > 0) {
        age = ageIn;
        isSet = true;
    }
    return isSet;
}
```

Method returns either true or false.

Add a method to return the age. Notice this method takes no parameters, but returns an int value (instead of void).

```
public int getAge() {
    return ____;
}
```

Method returns an int value.

Finally, add a method to return the location of the user.

```
public ____ getLocation() {
    return ____;
}
```

Add the following two methods to allow the user to log off and on:

```
public void logOff() {
    status = OFFLINE;
}

public void logOn() {
    status = ONLINE;
}
```

The actual values (0 and 1) for offline status and online status are not used in the code because they are declared as constants. This makes code easier to read and modify later. You should also hide the values when printing the class information. Modify the toString method as follows to print "Offline" rather than 0 and "Online" rather than 1:

```
public String toString() {
    String output = "Name: " + firstName + " "
        + lastName + "\n";
    output += "Location: " + location + "\n";
    output += "Age: " + age + "\n";
    output += "Status: ";
    if (status == OFFLINE) {
        output += "Offline";
    }
    else {
        output += "Online";
    }
    return output;
}
```

If statement uses the String "Offline" or "Online" rather than 0 or 1 for status in the return value for the toString method.

After compiling your class, test each of your methods in the interactions pane. Your output should be as follows:


```
▶ UserInfo u = new UserInfo("Jane", "Lane");
▶ u
  Name: Jane Lane
  Location: Not specified
  Age: 0
  Status: Offline
▶ u.setAge(23);
▶ u.setLocation("Auburn");
▶ u.logOn();
▶ u
  Name: Jane Lane
  Location: Auburn
  Age: 23
  Status: Online
▶
```

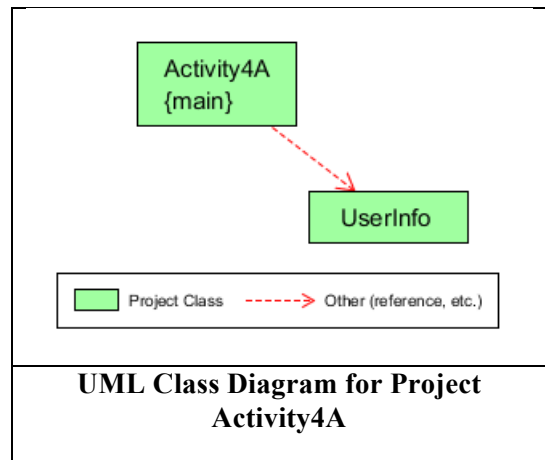
Part 2: Create Activity4A class with main method; create a project file and generate the UML class diagram for the program; create canvas for Activity4A and run the program in the canvas.

- **Activity4A class with main** – In addition to writing the UserInfo class above, you are to create another class, Activity4A, which should be saved in the same folder as UserInfo. Activity4A should have a main method that creates two or more instances of UserInfo and invokes methods on these instances. For example in the code below, two instances of UserInfo are created and assigned to variables user1 and user2 respectively. These instances are printed after they are created. Then after several methods are called on instances, the instances are printed again. Enter the code below incrementally, compiling and running it at strategic points to ensure that the program is correct down to that point and that you understand what each statement does.

```
public class Activity4A {
    public static void main(String[] args) {
        UserInfo user1 = new UserInfo("Pat", "Doe");
        System.out.println("\n" + user1);
        user1.setLocation("Auburn");
        user1.setAge(19);
        user1.logOn();
        System.out.println("\n" + user1);





        UserInfo user2 = new UserInfo("Sam", "Jones");
        System.out.println("\n" + user2);
        user2.setLocation("Atlanta");
        user2.setAge(21);
        user2.logOn();
        System.out.println("\n" + user2);
    }
}
```

- jGRASP Project** – Create a jGRASP project named Activity4A then add Activity4A.java and UserInfo.java to the project. To do this, with Activity4A in the edit window, click **Project > New** then in the Create Project dialog enter Activity4A as the Project Name; make sure the folder name is set to the folder of the source files, and click Next. You can add the two files via the dialog, or alternatively, after the project has been created, you can drag Activity4A.java and UserInfo.java from the Browse tab file list to project Activity4A in the Open Projects list.
- UML class diagram** – After you have created the project file and added Activity4A.java and UserInfo.java, you should generate the UML class diagram for the project by clicking  (or **Project > Generate/Update UML Class Diagram**). After the diagram is generated, you should see green rectangles representing the two classes. The red dashed arrow from Activity4A to UserInfo indicates that Activity4A depends on UserInfo. You can reposition a class in the diagram by selecting it and dragging. In your UML diagram, position the classes as they appear in the figure at right.


















Right-click on UserInfo and select “Show Class Info” to see the class member information in the UML Info tab. See **Members for UserInfo** in the figure at right below the UML class diagram.

Right-click the red dashed dependency arrow and select “Show Dependencies Info” to see the dependencies of between the classes in the UML Info tab; i.e., Activity4A is using the UserInfo constructor, and methods logOn, setAge, and setLocation in the UserInfo class. See figure below.

Activity4A -> UserInfo	
FIELDS:	
CONSTRUCTORS:	
	UserInfo(): UserInfo(java.lang.Strin
METHODS:	
	logOn(): void logOn()
	setAge(): boolean setAge(int)
	setLocation(): void setLocation(jav

Dependencies in UML Info tab

UserInfo	
FIELDS:	
	OFFLINE: private static final int OF
	ONLINE: private static final int ONI
	age: private int age
	firstName: private java.lang.String
	lastName: private java.lang.String l
	location: private java.lang.String lo
	status: private int status
CONSTRUCTORS:	
	UserInfo(): public UserInfo(java.lar
METHODS:	
	getAge(): public int getAge()
	getLocation(): public java.lang.Stri
	logOff(): public void logOff()
	logOn(): public void logOn()
	setAge(): public boolean setAge(in
	setLocation(): public void setLocat
	toString(): public java.lang.String t






Members for UserInfo in UML Info tab

You can also create instances of the class by right-clicking and selecting “Create New Instance”. This places an instance on the workbench.


To invoke a method for an object on the workbench, you can right-click on the object (not the class) and select “Invoke Method”.

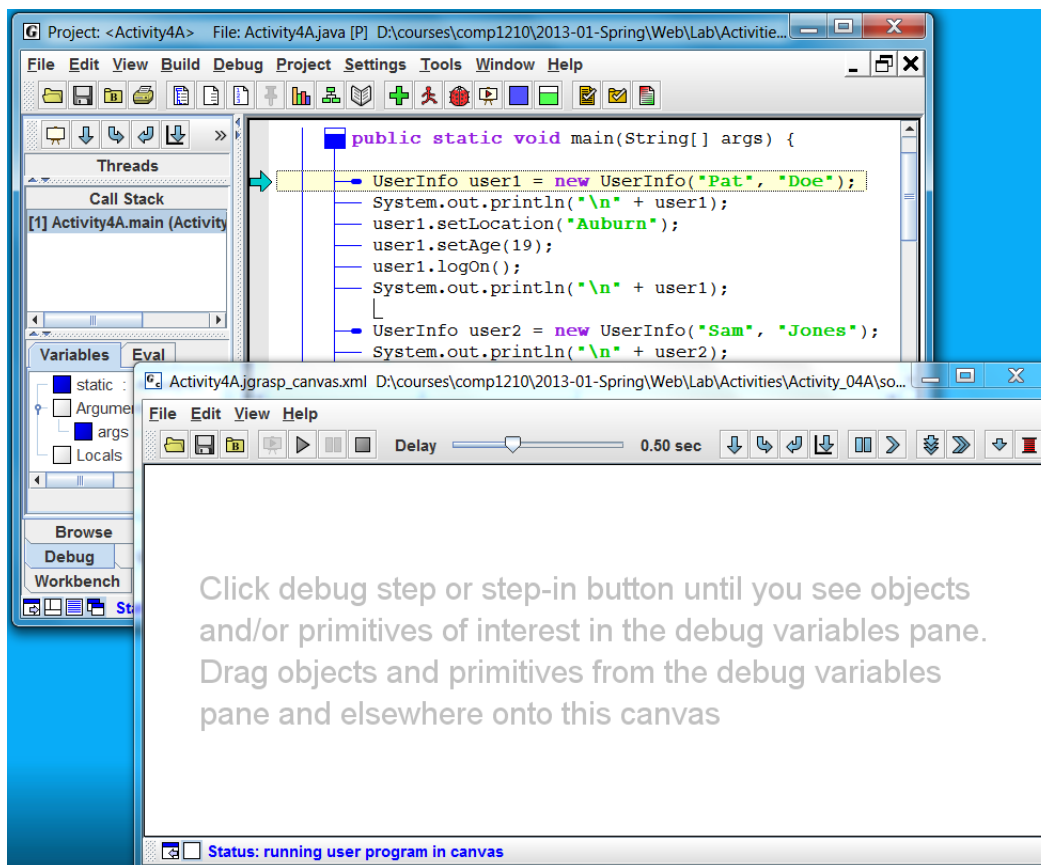
- **jGRASP Canvas** – In this part of the activity, you are to create a viewer canvas for your program. This will allow you to observe the objects created in the program as method are invoked on them. The canvas works in conjunction with the debugger as well as interactions.




After you successfully Compile  your program, you have three ways to run your program in jGRASP: Run , Debug , and Run in Viewer Canvas . In this part of the activity, we focus on Run in Viewer Canvas , which opens a canvas window on a new or existing canvas file. When any primitive, object, or field of an object in the Debug or Workbench tabs is dragged onto the canvas, a viewer is opened using one of several viewers associated with the variable type. Below are the basic steps for creating a canvas for your program.

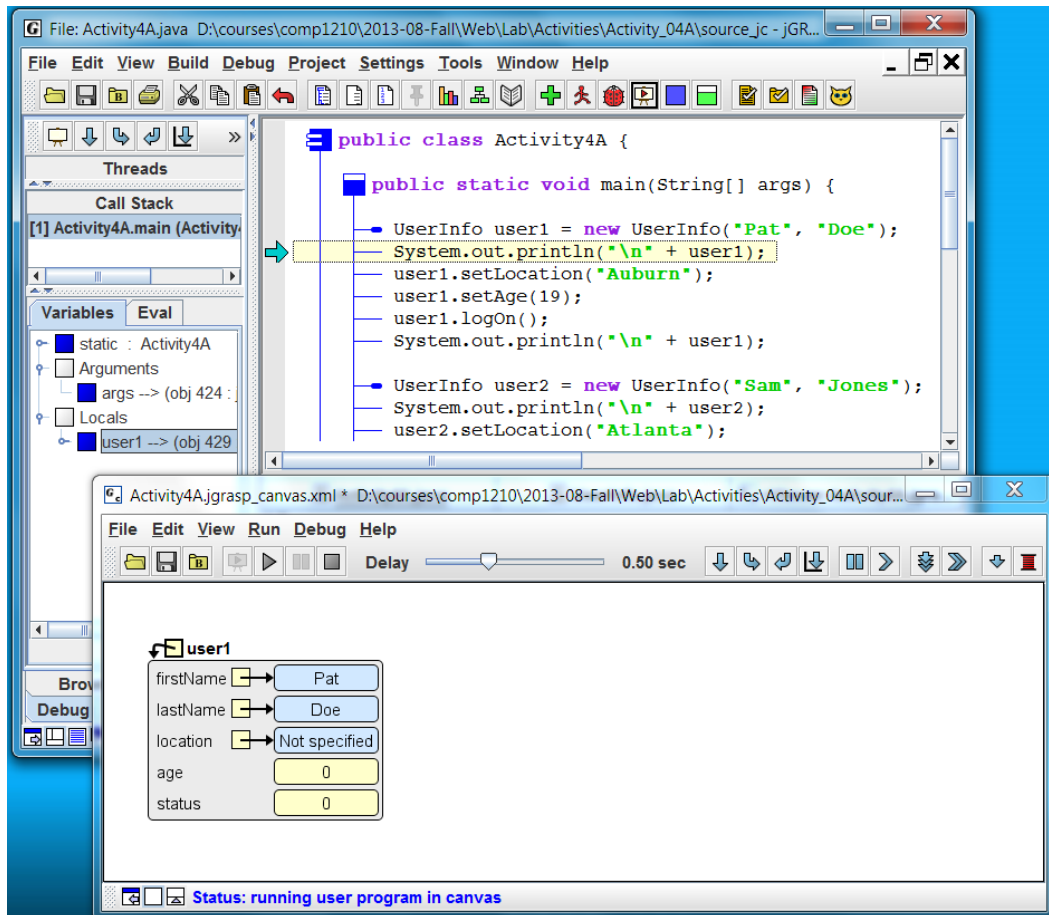
Make sure that Activity4A.java is in the edit window and that you have compiled it. Now you are ready to follow the steps below to create a viewer canvas for Activity4A.

- (1) On the desktop toolbar, click the Run in Canvas button . This launches the program in the debugger, stops at the first executable statement, and opens an empty canvas window (since a canvas has not yet been created for program) as shown below.





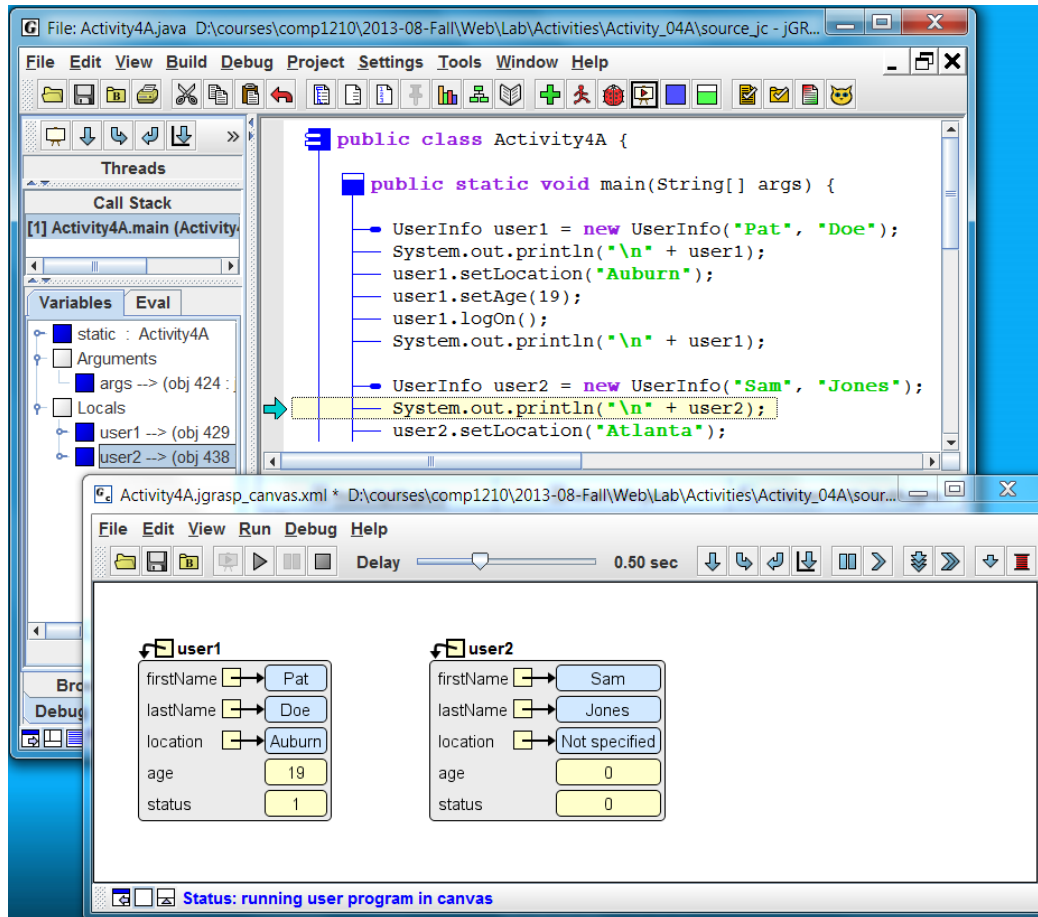
Running Activity4A in the canvas and stopped at the first executable statement

- (2) Click the Step button  on the canvas window or debug tab. This creates an instance of `UserInfo` and assigns it to the variable `user1`. You should see `user1` in the Variables tab of the Debug pane.
- (3) Drag `user1` from the Debug tab onto the canvas; a default viewer should open for `user1` as shown below.







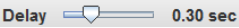
After `user1` has been dragged onto the canvas


- (4) Click the Step button  on the canvas window or debug tab to execute next statement in `Activity4A`. When a method is invoked on `user1`, you should see the respective field updated on the canvas.
- (5) Keep stepping  until you have created the second instance of `UserInfo` and assigned it to the variable `user2`. Now drag `user2` onto the canvas and position it to the right of `user1`. Your canvas should look similar to the one below.



Canvas with user1 and user2

- (6) Save the canvas by clicking the Save button  on the canvas window.
- (7) Now click the Step button until the program ends. After the last step in the program, you should see a status message at the bottom the canvas indicating that the run in canvas has ended. If you step one more time, the objects will be grayed out. At this time, the viewers on the canvas should be grayed out to indicate that the objects no longer exist. Now close the canvas window by clicking the close button on the upper left corner.
- (8) On the desktop toolbar, click the Run in Canvas button . This launches the program in the debugger, stops at the first executable statement, and opens the canvas window that you created above. Now step  through the program as you did above and observe the changes in the objects and the output of program. After the last step in the program, you should see a status message at the bottom the canvas indicating that the run in canvas has ended. If you step one more time, the objects will be grayed out as before.
- (9) Now you are going to “play” your program. On the desktop toolbar, click the Run in Canvas button . This launches the program in the debugger, stops at the first executable statement, and opens the canvas window that you created above.

On the canvas, click the Play button ► (auto step-in) on the canvas to start the visualization. Use the Pause button ■■ and Stop button ■ as needed. To regulate the speed of the program, decrease or increase the delay between steps using the Delay slider. 

When you play ► your program, you will be stepping into each of your methods. If it is going too fast for you to see and understand what is happening at each step, then increase the delay between steps by using the Delay slider.  You can also click the Pause button ■■.

After you pause ■■ your program, you can step ⏴ as you did above. Or if you want to step into a method at a statement that calls the method then you can click the Step-in button ⏴.

You can also set one or more **breakpoints** (right-click on the line and select *Toggle Breakpoint*) in any of your methods and then Resume ► to the next breakpoint. The debugger will stop at the breakpoint, and you can examine the variables in the Debug tab or on the canvas. You can then step ⏴, step-in ⏴, or play ► your program as needed.

You can always stop ■ (or end) the program, and then start it again by clicking the Run in Canvas button 🏠 followed by the Play button ►.

It is important that you understand what your program is doing at each step. Although you can observe the behavior of your program using the debugger alone, the canvas works with the debugger to provide a more conceptual visualization of your program. When you are studying the example programs provided with the class notes, you are encouraged to run these in canvas mode. After you have added the variables of interest to your canvas and saved the canvas, you should be able to play or step through the program to understand the details of its behavior. Until are able to understand the example programs, it will be difficult for you to write your own programs for the project assignments.

For the Activity4A program above, you wrote the entire program before you created the canvas for it. However, you could have created the canvas as soon as you were able to compile and run part of the program. As you write you own programs for the project assignments, you can create a canvas as soon as there is any observable behavior. This will help you ensure that the program is correct at each increment during development. The canvas will be particularly useful when you are attempting to discover and correct errors in your program.

Note that you can also use the canvas via the Debug or Workbench tabs by clicking the Open New Viewer Canvas 🏠 button on the debug or workbench toolbar and then dragging one or more variables onto the canvas. Changes to these variables resulting from statements executing in the Interactions tab, from stepping the debugger, and/or from executing methods via the Invoke Method dialog will be reflected on the canvas.

Usually, you will need only one canvas for your program. However, if you need more than one, you can open a new canvas window via the Debug tab drag variables onto it and save it as described above.