# CS 273 Laboratory 9: Debugger

This lab gives you additional experience using the debugger in BlueJ. Its focus is debugging when a program has multiple methods and classes.

## Preliminaries

1. Flag Application

In this lab, you will be using the BlueJ debugger to control the execution of a program that draws an American flag on a window. This flag is built from:

* A Flag object contains:
  + A ColorRect array, which represents the stripes on the flag. A ColorRect is a rectangle that knows its color.
  + A StarRect object, which represents the square with stars on it. A StarRect contains:
    - A (blue) ColorRect object.
    - A StarRow array. Each StarRow object represents a horizontal row of stars. It contains a set of:
      * Star objects. A Star is a star-shaped Polygon that has a color. Because it is a Polygon, it implicitly contains an x- and y-position.

In other words, each star and rectangle that you see is an individual object that knows how to paint itself.

Each of the above classes implements:

* A constructor, that allows an object of that class to be created. Most of the above object types simply create all of the sub-objects.
* A paint method, that paints the object on the screen. Most of the above object-types simply invoke paint on all their sub-objects.

*Note:* there are many instance variables in this lab that are presently declared as public, that would eventually be changed to private. Leaving them public allows us to have access to objects' internal structure while debugging. We have declared these instance variables as public//private , as seen in the ColorRect class:

// a polygon that defines our rectangle

public//private

Polygon p;

(Thus, the private is commented out.) This would allow you to easily find and change them to private at some point in the future.

### 2. Running the Flag Application

In order to run this flag application you must perform the following steps:

* Right click on the Flag file.
* Select "new Flag()".
* Accept the default object name by pressing "OK". (You can give it different name, if you'd like.) A red icon at the bottom of the main BlueJ window which represents the flag object you’ve just created will appear.
* Right click on the newly created object and select "void view()"

### 3. Restarting BlueJ (should it hang)

BlueJ often refuses to let you recompile because it thinks that something is still running. If you get into that situation, and "terminate" does not work, you might try the following in order to "cleanly" restart after having been at a breakpoint:

* Clear all your breakpoints
* Click "Continue" to allow the flag window to be completely drawn
* Close the flag window by pressing its "close" icon
* At this point, it still may not let you recompile. If this is the case, perform these steps:
  + Delete the flag object (red icon in main BlueJ window) by right clicking on it and selecting "Remove" (e.g., "flag\_1")

At this point, it still may not let you recompile. If this is the case, follow these steps:

* + Create a new flag object by right-clicking on "Flag" and selecting "new Flag()".
  + At this point, it often lets you recompile. If it doesn’t, then you will have to kill BlueJ and restart it.

### 4. Advice for this lab

**Advice #1:**

* Have your BlueJ windows organized on your screen for this lab.
* Open up the classes in BlueJ in this order: FlagViewer, FlagCanvas, Flag, ColorRect, StarRect, StarRow, Star

**Advice #2:**

“Step” executes one line of current method

“Step into” goes into body of method if the line has a method call

Note: they do the same thing if the line of code does not contain any method calls

**Be careful: you can “step into” Java library code!!** That’s not a major issue, but it can be confusing or overwhelming.

**Advice #3:**

“First” means the 0th item in a row or column

**Advice #4:**

Ctrl-b (in Windows; Cmd-b in MacOS) sets and removes breakpoints. This only works if your code is compiled. It also doesn’t work on blank lines or comments.

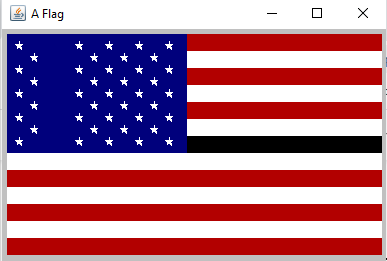
## Laboratory

### Part 1: Examine the Java files, and become familiar with the program by making some simple changes

1. Study the constructors in Flag, StarRect, and StarRow to see how they create their sub-objects. Examine the paint methods to see how they paint their sub-objects.
2. To help you become familiar with the objects, open the project and run it according to the instructions in “2. Running the Flag Application” on page 2.
3. Then, make changes to the source code so that the following happens. *Take care to make your changes so that they are easy to undo*.

* The second star in each row does not get painted. (Remember, array indexing starts from 0, so the index of the second star is 1.)
* The color of the **middle** red stripe is black.
  + Hint: you should only need to edit or add one to two lines of code in the appropriate classes to make the changes

This is what the flag should look like with the second star missing in every row and the middle red strip as black.



**checkpoint 1 (15 points): Show your lab instructor or assistant the executing application and the code you changed.**

### Part 2: Arrange your windows so that they will not overlap the debugger

The purpose of this part is to get the debugging window "out of the way" of the window. During the debugging of a paint method, if a window covers and uncovers the window, the system may send additional paint commands to the window, causing it to hang. When you're debugging the application, it is often stopped, and therefore is unable to repaint itself. Follow these steps:

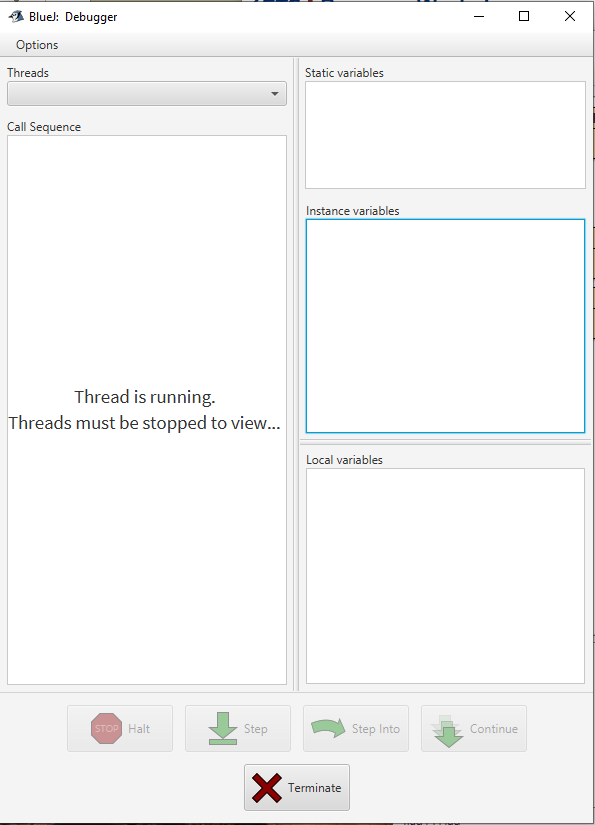
1. *Undo* the changes that you made in part 1. You should now have a real American flag again.
2. Run the application.
3. Open editor windows for all the .java files except for FlagViewer.java. Move all of the editor windows so that none of them cover any portion of the running application—the top-left part of the screen--but are still as visible as possible.  You should move them all to the same part of the screen. During this lab, you should avoid moving these windows over the space where the flag will appear.

In later steps, whenever we want to debug the application's paint method (or something it calls), we will

* 1. Clear any breakpoints that might be set.
  2. Set one or more breakpoints. Note, you can only set a breakpoint if the code has been compiled first. There is no need to recompile after you set a breakpoint (because compiling is necessary only when you change code).
  3. Run the application in the debugger.

1. Perform the steps a, b and c above. When you get to step b, set a breakpoint at the first statement of the paint method in Star.java. After you do step c, it should stop at that breakpoint; the flag should be painted, but without stars (because they are not painted before the breakpoint stops the application). (If the debugger covers the flag-window when it comes up, move the debug-window, terminate the application using the debugger’s Terminate button; then rerun the application.)
2. Then hit Continue in the debugger window. Each time you press Continue, a single star should appear on the blue rectangle.

\*\*\* If you don’t see anything in the debugger window, like this, follow the steps below to fix that.



* + - 1. Go to the Options menu in the debugger window and uncheck Hide System Threads.
      2. Then in the Threads dropdown, select AWT-EventQueue-0 (at breakpoint)

**checkpoint 2 (15 points): When you reach this point, show your lab instructor or assistant the application and your Star.java file with execution stopped at the breakpoint. Then do several Continue operations. For each one, a single star should appear on the blue rectangle.**

### Part 3: Use the debugger to watch the flag get drawn object-by-object

Two of the most important debugger-commands are:

* Step[[1]](#footnote-1): This executes one line in the current method. When the method finishes, it returns to the method who called it.
* Step Into: If the current line contains a method call, it goes into the body of that method so you can watch what is happening inside the method. If the current line has no method call, it behaves like Step.

By using these commands, you can watch the program executed on a "step-by-step" basis.

You can also terminate execution using the debugger’s Terminate button.

Set a breakpoint at the first statement of the paint method in FlagCanvas.java. Run the application in the debugger. After the breakpoint is hit, use the "Step" and "Step into" buttons to get yourself in a position to do each of the checkpoints below. The only breakpoint that should be set during any of these is the one at the first line of the paint method in FlagCanvas.java. You should use the "Step" and "Step into" buttons  to navigate to appropriate portions of the program as it executes.

**checkpoint 3 (30 points): Show your instructor or lab assistant the following items:   
1) The stripes being drawn one at a time each time you hit the "Step" button twice.**

**2) Rows of stars being drawn one row at a time each time you hit the "Step" button twice.**

**3) Stars being drawn one star at a time each time you hit the "Step" button twice.**

### Part 4: Examine the call stack

The *call stack* (which is labeled the “Call Sequence” in the BlueJ debugger) is the list of methods that are currently in the process of running: the first element of the call stack is the method that is currently running; then next is the method that called it; below that is the method that called it; and so on. You can examine the call-stack and find out where each call "came from" using the call-stack window. The *call stack* is located on the left-hand side of the debugging window brought up by BlueJ.

1. Terminate the application using the Terminate button in the debugger.
2. Run the debugger on the application as before, with a breakpoint set at the first line in the paint method of Star.java. When you hit the breakpoint, click on the various methods in the call-stack window. (Ignore "system" methods that you do not recognize, such as dispatchEventImpl or anything that starts with sun.awt or java.awt.) Notice that method will be displayed in the code window, regardless of which class it is in. That allows you to easily jump to the part of the code you care about most when you’re debugging something.

**checkpoint 4 (10 points): Show your lab instructor or assistant the that you can examine the call-stack.**

### Part 5: Use the inspector to examine and modify objects

BlueJ has three variable windows that are shown when you are debugging. They are the *static*, *instance* and *local* variables and are located on the right-hand side of BlueJ's debugging window.

1. Terminate the application using the Terminate button on the debugger.
2. Restart debugging, ensuring that the application window (AKA the flag window) overlaps no others. Set a breakpoint at the first statement of the paint method in FlagCanvas.java. Use "Step into" and "Step" to get to the point in StarRect.java where it is just about to paint star row 4 (that is, the fifth row, because Java starts counting from zero).
3. Before it paints row 4, use the variable window to inspect row 4:

* Double-click on the rows instance variable (not the row local variable).
* Click on the fourth index and click Inspect.
* Click on the stars variable and click Inspect.
* Click on any array position and click Inspect.

You can now see the data inside the star object. What do you suppose each data member means or is used for?

**checkpoint 5 (10 points): Show your lab instructor or assistant that you can examine the objects in this way and explain the contents of a Star object.**

### Part 6: More Object Modification

1. Close all Object Inspector windows and terminate the application using the debugger’s Terminate button.
2. Create a new Flag object, but do not right-click and invoke the view method on the Flag.
3. Change the color of the third star in the fourth row by doing the following:

* Find the object you want by navigating (via inspectors) down from the Flag object (e.g., flag\_1 red icon) to find the object you want. That is:
  + Right-click on the Flag object and select the Inspect button.
  + Inspect the starRect object
  + Inspect the array of StarRow objects
  + Select and inspect the fourth StarRow object
  + Inspect the array of Star objects
  + Select the third Star and click the Get button
  + Enter a name of the object (or just use the default) and select Ok.
  + This Star sub-object will now appear along the bottom row of BlueJ’s main window (next to the Flag object). You may need to close the inspector windows to see it.
* You can right-click on the Star object, and apply a method (in this case, setColor) to it.
  + Java will bring up a dialog box that will prompt you for the argument(s) to the method; type it in. (BlueJ seems to need the java.awt.Color.green rather than just Color.green.)
* You can now perform a view on the Flag object in order to see the effect.

This is a permanent change to that Star object. The star will remain green until that star (or the entire flag) is deleted. If, on the other hand, you created a new flag object, it would be colored normally.

1. Use a similar process to change the Flag's first stripe to be yellow and its last stripe to be gray. **Again, do not modify the code; rather modify the object using the debugger.** In order to see the effect, you may need to force a repaint on the flag (e.g., by minimizing and unminimizing the flag window).

**checkpoint 6 (20 points): Show your lab instructor or assistant your running application with the third star in the fourth row as green, top stripe as yellow, and bottom stripe as gray.**

### EXTRA CREDIT

### In the following optional checkpoint, you will walk through a tutorial about a more elegant IDE (integrated development environment) than BlueJ.

### Part 7: Eclipse

### Log in to the lab computer via Virtual Desktop at desktop.up.edu.

### Launch Eclipse. You can search for Eclipse using the magnifying glass icon on the taskbar, or you can click on the Start Menu and then CS, which opens the folder to C:/Courseware/CS, and you’ll see Eclipse listed there in alphabetical order.

### Upon launch, you may see the following message. Click Skip.

### 

### Click on Create a Hello World application.

### 

### On the left, click on Create a Java Project.

### 

### Select the Java Project wizard.

### 

### Enter your choice of project name (i.e. HelloWorld). Leave the rest of the settings as default. Click Next (not Finish).

### 

### Uncheck Create module-info.java file. Then click Finish.

### 

### Click on File > New > Class. Enter the name of the class as HelloWorld. Check the box to create public static void main(String[] args). Click Finish.

### 

### Inside of the main method, type System.out.print. Notice how Eclipse suggests which methods are available that match the letters you’ve typed so far. Furthermore, it provides the method header and an explanation of the method, so you know what it does. You can hit Esc to hide the suggestions and Ctrl + Space to bring them back. You can use the arrow keys or your mouse to scroll through the suggestions. You can hit Enter on a highlighted suggestion or double-click using the mouse.

### 

### Once you’ve entered the appropriate code to print out Hello World, hit Ctrl + F11 to run the program. You can also click on the green circle with a white arrow inside of it . If you’re prompted to Save and Launch, make sure the box next to HelloWorld.java is clicked, as well as the checkbox at the bottom. Hit Ok. You should see the output in the Console window at the bottom of the screen. Note, in Eclipse, you do not need to compile before you run the program because the program is always compiled automatically when you tell it to run.

### 

### 

### Let’s create a for loop that will print out Hello World a random number of times up to 100.

### Start by creating an object of type Random (don’t import anything before doing so). You’ll notice red squiggly lines, indicating an error. There will also be a little red box with a white x inside on the left margin. Click the x to see a list of suggested solutions. The most appropriate solution in this case is to import the library that contains the Random class, so select that option. Notice, the import statement goes at the top of the class (as we would expect) and the error went away!

### 

### Store the result of rand.nextInt(101) into an int variable. We will use this as the upper bound of the loop.

### Write a for loop that prints Hello World a random number of times. Notice that when you type the opening brace and hit Enter that Eclipse automatically creates the closing brace for you. Update the print statement to include the counter so you know exactly how many times Hello World is being printed. Run your program a few times to make sure that Hello World is being printed a random number of times.

### Right-click on the variable used inside the for loop. Select Refactor > Rename… Type the new variable name and hit enter. Notice all uses of that variable got updated. Run your program a few times again to make sure it still works as expected.

### 

### I hope you found it worthwhile to go through this tutorial. Now you know some of the highlights that make Eclipse a nicer IDE to use for Java than BlueJ. Going forward, you are welcome to use either IDE for this course. Check with your CS203 instructor if you want to use an IDE other than BlueJ for that course.

### checkpoint EC1 (10 points): Show your lab instructor or assistant how to refactor a variable name and demonstrate that your program runs as expected with a random number of lines printed.

1. In other debuggers, this is sometimes called “Step Over” [↑](#footnote-ref-1)