lab: Java Projects in Eclipse

February 13, 2013

Contents

This lab will walk you through the basic steps of editing, compiling, and running Java programs in *Eclipse*. There are many other IDEs available to you as a java programmer, but I recommend Eclipse as a development environment that can grow with you. It's powerful enough to be a popular choice among professionals, but has enough built-in help features to aid the novice.

1 Getting Access to Eclipse

You have several choices here. Which one you choose may depend on where you are at any given time, what kind of network access you have, and what other software you might need to be working with.

- Eclipse is available on PCs in the CS Dept laboratories and, if you are off-campus, on the CS Dept's Virtual PC Lab machines.
- Eclipse can be run on the CS Dept's Linux ssh servers (the command is "**eclipse &**"). Of course, being a graphical user interface, this requires you to be running X. If you are connecting from off-campus, you will really want to be running NX instead of ordinary X.
- You can install Eclipse for free on your own Windows, Mac OS/X, or Linux PC.
  - First you will need to make sure that you have a copy of the Java Software Development Kit (JDK, also known as the J2SE). Note that Oracle distributes Java in two basic forms: the Java Runtime Environment (JRE) is the basic package needed to run Java applications and web applets that someone else has compiled and provided. You probably already have this, but you want the larger Java Development Kit (JDK) that provides the compiler and associated tools. Follow Oracle's instructions to install this.
  - Then get Eclipse from the Eclipse Foundation. There are lots of variations available. I would suggest getting either the "Classic" package or the CDT package that includes support for working in C++ as well as in Java.

Installation is usually no more involved than simply unpacking the Eclipse package into a folder of your choice.

– After you have worked with Eclipse for a while, you may want to use the built-in "Install New Software..." dialog (under the Help menu) to add the CDT package for C++ support, if you did not get that paackage initially. Eclipse makes a very good development environment for C++ as well as Java - it supports the GNU g++ compiler. On Windows it supports both the MinGW and Cygwin/X ports of g++.

2 Overview of Eclipse Concepts

You typically begin work with Eclipse by creating a *project* or by returning to a project that you have previously created.

By default, Eclipse gathers all of your projects together under a common directory called your *workspace*. Each project occupies its own subdirectory within the workspace. However, if you prefer, you can override this default and keep your projects wherever you like. For example, if you already have some source code files in a directory somewhere, you would probably set up the project where these reside.

You can then edit existing files or create new ones. The Eclipse Java editor is smart. It "understands" Java.

- Eclipse can complete partially-typed names and offer suggestions as to what may legally be placed at the cursor position. Type Ctrl-space to see this help. It also may offer such help if you simply pause for a moment at a suggestive spot (e.g., immediately after a '.').

  You may have seen other editors that offered a similar feature, but the Eclipse suggestions reflect its understanding of the Java language and of the state of your project. For example, If you ask for help in a context where you need to type a function call, Eclipse will suggest only those functions that would actually be legal in that context, including both Java "system" functions and functions you yourself have declared elsewhere in your project.

- The Eclipse Java editor has another feature that speeds development. While you are typing your code, it is

checking continuously for errors. Almost all Java compilation errors will actually be flagged for you as you commit them, instead of waiting until you explicitly request a compilation/build of your project.

- When you are reading unfamiliar code and trying to understand it, Eclipse can help you navigate the project. If you select a variable or function name with the mouse, you can ask Eclipse to take you to the line of code where that name is declared. Or you can select a declaration and ask Eclipse to list all the places in your code where that name is used.

When you are ready, you can attempt a project build, via the menus or the toolbar buttons (hover your mouse over each button to discover what it does. If you have error messages, selecting one will take you to the relevant code location.

Once your errors have been dealt with, you can run the project or launch the debugger. You can do this via the menus, the toolbar buttons, or by right-clicking on the executable file in the Project Explorer and selecting the relevant action.

Eclipse has some truly interesting features, especially for Java. For example, it "knows" about many of the common transformations that programmers make to their code and can apply them in one step. These are called *refactoring*s. A basic example of refactoring is renaming a variable or function - Eclipse not only changes the name where the variable/function is declared, but also all the places where it is used, and it is smart enough to use the Java language scope and overloading rules to distinguish between uses of different functions or variables that happen to have the same name. Another refactoring is to "encapsulate" a data member, making the data member private and creating get/set functions for public access to it, all in one step.

3 Example: Building a Java Program in Eclipse

1. Choose a convenient directory where you want to keep a project. Copy the files `Pie.java`, `PieSlicer.java`, and `PieView.java` into that directory.

2. Start Eclipse. You will be asked where you want to keep your workspace (the default area for all of your project settings). Accept the suggestion or browse to a more convenient directory.

   If this is your first time running Eclipse, you will be taken to a Welcome page. Close that or click on the curved arrow on the right to enter your workbench.

   If you have just installed Eclipse on this machine, then edit the Preferences (under the Window menu) on your first run. Under "Java", look at "Installed JREs" and make sure that it shows your JDK and that the JDK is checked to make it the default.

3. Now let's create a project. Select "New->Project..." from the File menu or by right-clicking in the Package explorer pane on the left. Select "Java Project" and click Next.

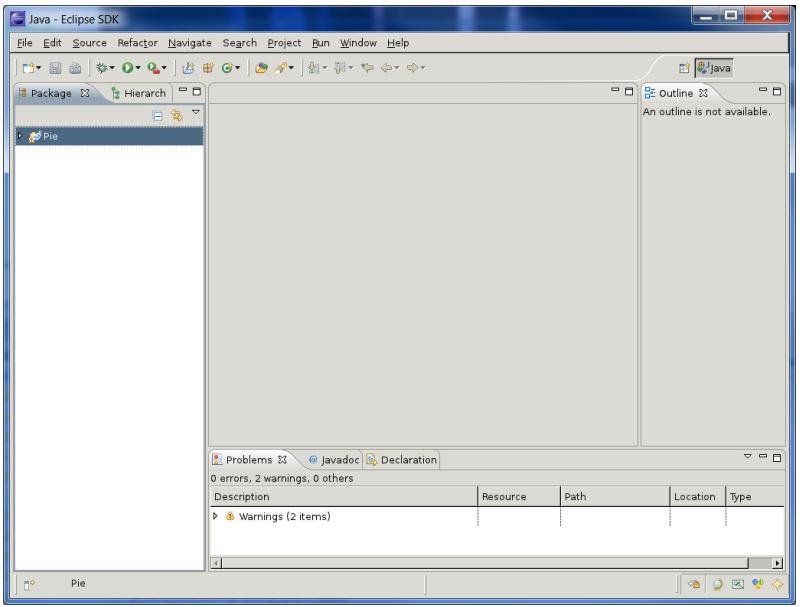
   Give your project the name "Pie". Select the option to "Create project from existing source" and then browse to the directory where you placed those Java files. Accept the rest of the defaults, clicking Next until you come to Finish, then click that.

   A box may come up asking if you want to open the Java perspective. Click in the "Remember my decision box" and then click "Yes". (You probably want to do this whenever you are asked about opening a new perspective).

4. At this point, you should be looking at something like the picture below.

If you expand the listing, in your Package pane, you will see an entry for the "default package" and one for the system library. Expand the default package and you will see the three Java files that you had placed in the directory. Remember that, in Java, classes are organized into packages. If you don't name a package to serve as the container, then the class is assumed to belong to the "default" package - hence it's listing here in Eclipse, which organizes your code according to your package structure.

By this time, Eclipse has probably already compiled all three files. You may see a couple of warnings down in the Problems area. You can expand the Warnings item in the Problems pane to see the resulting messages. These should be some technical warnings about the lack of a serialVersionUID, which is a technicality that we really don't need to worry about.



5. Double-click on `Pie.java` to open it in the editor. Let's introduce a simple error into the code. Look for a line that mentions `Color.red` and change "red" to "maroon". A small red marker will appear to the left of this statement to indicate that an error has been detected. To the right, another red marker will appear to show roughly how far down in the file this error is located. Hover your mouse over the red marker on the left. A tip should pop up stating that "Color.maroon cannot be resolved". In other words, the class *Color* does not contain a constant named "maroon".

Of course, you might not know what color names are provided by that class. But Eclipse can tell you. Delete the "maroon", leaving your cursor just after the '.' in that line. Hit Ctrl-space to request help, and you will be given a list of all constants and functions provided by the *Color* class that have the appropriate data type to be used here. You can scroll through the list and pick one (Don't pick green, though.) The red error markers should disappear once you have done this.

Save your changes via the File menu or via the floppy disk icon (how dated!) in the toolbar.

6. When you save, the code is actually compiled. This compilation may find errors not detected by the editor's on-the-fly checking. If so, you will see red error markers appearing both in the editor window and in the Package browser on the left. To see this effect, move down to the `draw` function and put a "2" after the "w" to change the name to "draw2". This is a perfectly legal change to make, so there is no initial complaint from Eclipse. Now save the file again.

   Notice that a red mark appears in the Package browser next to the `PieView.java`. There is also now an Errors section in the Problems tab at the bottom of the page. Expand the error listing and double-click on the error message there. The editor immediately takes you to the location of that error, in `PieView.java`. Unsurprisingly, perhaps, the error occurs at a call to the "draw" function, which, after our change, no longer exists.

   Move your mouse over the error marker to the left of the draw() call. Hovering over that location causes the error message to pop up. We've seen that before. But now left-click on that error marker. Eclipse actually offers suggestions on how to fix the error, And they are actually good suggestions! (How cool is that?) One suggestion is to change the call to "draw2". The other is to add a new function named "draw" to the Pie class. Either of these is a perfectly reasonable suggestion. Choose the option to change to "draw2". Depending on your operating system, you might select this either by using the arrow keys to highlight it and then hitting Enter, or by simply clicking on it. Eclipse makes the suggested change immediately.

   The error markers disappear from the editor, although one remains by the file name in the Package browser. Save the changes made to this file, and that one goes away as well. At that time, the listing in the Problems area also disappears.

7. Try using the same technique to get rid of those two nagging warnings. Double-click on each warning message in the Problems area to go to the code location with the warning. Left-click on the yellow warning marker on the left, and select one of the actions marked with the green plus sign to correct it. Save your changes, and savor the delight of a completely empty Problems listing!

8. Now let's run the program. The `main` function for this program is in `PieSlicer.java`. Right-click on this in the Package browser and select "Run As..." "Java Application". You should see a small window pop up with a circle and a couple of buttons. Move your mouse around inside the circle a bit, and try the buttons. It's nothing spectacular - just a bit of fluff for us to play with in this lab. Close the window to kill the program.

   You can now run the program again from the run button (usually a white triangle within a green circle) in the toolbar. Actually, because this is a simple project, you could probably have used that the first time. But what this button normally does is to remember the last program that you ran within a project. So if you have multiple "main" functions in a project, you want to use the right-click menu the first time you want to run one.

9. Now let's add a new class to the project. In the Package browser, right-click on the default package and select New...Class. A dialog box will pop up offering you a number of options for your new class. Put "Filling" in the Name box, select Generate Comments, then click Finish.

   A basic class template will be set up in the editor.

   Click inside the comment, at the end of the line just above the @author tag. Hit Enter and add the description "What's the stuff inside the pie?".

   Then click on the line between the "{}" and type

```
public String filling = "rhubarb";
public Pie myPie;
```

   With the cursor positioned anywhere inside that line, you can correct the indentation with Ctrl-I (or choose "Correct Indentation" from the `Source` menu.

It's worth taking a moment to look at the `Source` menu. There's lots of options for controlling formatting and appearance of your code. You can do a quick cleanup after you have typed a lot by using Ctrl-A to select all the code in your file, then choosing Format from the `Source` menu to reformat the whole thing.

You can also find some help in generating code. For example, one of the entries in the Source menu is Generate toString(). Later, you'll see that I believe that almost every class should have this function, as it is extremely useful in debugging. So try selecting it, check the "Generate method comments" box, and click OK.

That's kind of nice, isn't it? In fact, go ahead and save your changes to that file. Then return the `Pie.java` file, click anywhere inside the *Pie* class, and generate a toString function for this class as well. Save the result.

10. Let's do something similar for the *Pie* class. Edit `Pie.java`. Use the Source menu to generate a toString() function for this class. Save your file, and run the program again. Notice that the display at the top has changed. It has actually been showing the results of `Pie.toString()` all along. However, the default `toString()` function isn't all that useful. Now we have replaced it with one that,although not particularly well formatted for this application, still shows a great deal more useful information.

11. While still running the program, use the buttons to change to colors of the pie segments. Note that the `toString()` display changes to reflect the colors that you have selected.

12. Exit Eclipse. It will save your project information so that, when you return and run Eclipse again, as long as you select the same workspace you will find your project waiting for you. (Note that, although we chose in this example to store the code for this project outside of the workspace, the workspace is still responsible for remembering that this project exists and where we keep it.)

    We'll return to this program in future labs.

4 Generating a Jar File in Eclipse

In the Lab, you saw that we could use a single .jar file as a transport mechanism for all of the .class files that make up an entire program. In , you saw how to create Jar files from the command line. We can also use Eclipse to create Jar files.

1. Restart Eclipse.
2. Right-click on your Eclipse Pie project in the package Explorer and select "Export..." Under "Java", select "JAR file"
3. You'll be presented with a display in which you can select the files you ant to include in the JAR. In the "Select the resources..." area, select your Pie project. You probably want to check "Export generated class files.." and, optionally, "Export Java source files...".

   In the "JAR file:" area, designate the name and location you want for the JAR file.

   Click on Next.
4. On the next screen, select "Save the description..." and choose a location. This will allow you to quickly rebuild the JAR if you alter make changes to the code in your project.

   Click Next.
5. Select "Generate the manifest file". In projects like this one, where we have a single class that provides a main() function, we ca indicate that we want the Jar file to be "runnable", meaning that it will use that class by default (e.g., if someone double-clicks on the Jar file in Windows). In the "Main class:" area select the PieSlicer class.

   Click FInish.
6. Check and you should find that a new jar file has been created.
7. Exit Eclipse,

8. Try transferring the newly created Jar file to another machine. Run it by double-clicking on it (if you are in Windows or a similar operating system) or via the command line with the command

   ```
   java –jar jarFileName
   ```

   You should see the now-familiar pie display pop up shortly.

5 Debugging

This section will walk you through the basic steps of using the Eclipse debugger.

An automated debugger should allow us to set breakpoints, run the program until hitting a breakpoint or a run-time error, examine the call stack and data whenever the program is stopped, and to step through the code in steps of varying sizes. You've probably worked with such tools before, so we're only going to do a quick tour here.

1. Start Eclipse and return to the workspace that you used in the earlier lab. Open your Pie project (if necessary) by right-clicking ion it and selecting "Open".

2. Open `Pie.java` in the editor and set a breakpoint in the `setSliceAngle` function by double-clicking in the margin to the left of the assignment statement. A small blue circle should appear to indicate that a breakpoint has been set.

3. In the Package browser, right-click on PieSlicer.java and select "Debug as...Java Application". After the first time you have run a program, you can repeat this via the bug button in the toolbar. You may be asked whether you want to open the debugging perspective. Answer yes.

   The program will execute and will stop at the first call to `setSliceAngle`.

4. In the upper left, you can see the stack of active calls (*activations*) that led you to this point. `setSliceAngle` was apparently called from within the *Pie* constructor, which was called from the *PieView* constructor, which was called ... and so on down to `PieSlicer.main()`. Clicking any of those lines shows you the point of the call.

5. In the upper right, you can see a list of the parameters and local variables in the function call currently being displayed. Click on the various function calls on the left and observe how the list of parameters and variables changes.

6. Ranged above the list of function activations, you can see the controls for the debugger. Hover your mouse over these to see what they do.

   Click the Resume button to move forward to the next hit on this breakpoint. The window displaying our application will probably appear. In fact, if it appears far enough away from your mouse, the program won't actually pause because the setSliceAngle function has not been called again. Move your mouse inside the pie so that the program pauses.

7. This time, setSliceAngle will be called from PieView.TrackMouse. Click on that activation and look at the data. x and y and the various numeric variables that follow it are easy enough to understand. Clicking on one of those variables causes its value to appear in the box at the bottom of the data area. For the numeric variables, this does not tell us anything new.

   Click on the activation of `setSliceAngle`. You can see the value of the numeric input parameter easily. The "this" above that refers to the particular pie that `setSliceAngle` was called upon. Click on the triangle to the left of "this" to expand it. You can see each of the data members. Many of these can be expanded to show their internal data members. In theory, we could examine the value of anything by expanding it enough times. That can be rather tedious with complicated types, however.

8. As a quicker way of viewing the value, click on "this" to select it. In the box below the listing, you will see something like "Pie [baseColor=...]", which actually gives the value of each data member of our Pie class.

   Now, you might think that the debugger is generating that listing for us. But, in the editor pane, look at the toString function that we created earlier. You can see the opening string "Pie [baseColor=" followed by a lot of code to build a string out of the entire list of Pie data members.

   In fact, the debugger displays variables for us by invoking the variables' toString() function.

Try replacing the body of the Pie toString function by

```
return "Yum!";
```

Save your change and click on "this" again. Notice that the displayed value does indeed change to this less-than-informative string.

Try removing the toString function entirely from the Pie class. You'll probably be asked to restart the running program. Go ahead and let the debugger restart it. When the new execution halts, click on "this" again. You're now looking at the default output that the debugger uses for classes that do not provide a toString() function. Not very useful, is it? You can see why I say that almost every class should provide that function, and why it's so convenient that Eclipse can generate a useful version for us.

9. You can continue playing with this code a bit. Try out the various stepping controls for moving incrementally through the code. They should be, by and large, familiar to you from other debuggers that you have worked with.

10. When you are done, exit Eclipse.