6 Starting in Eclipse

Goals

In the first part of this lab you will learn how to work in a commercial level integrated development environment IDE Eclipse, using the Java 1.5 programming language. There are several step in the transition from ProfessorJ:

1. Learn to set up your workspace and launch an Eclipse project.
2. Learn to manage your files and save your work.
3. Learn the basics of the use of visibility modifiers in Java.
4. Learn the basics of writing test cases in Java.

6.1 Learn to set up your workspace and launch an Eclipse project.

Start working on two adjacent computers, so that you can use one for looking at the documentation and the other one to do the work. Find the web page on the *documentation* computer:

http://www.ccs.neu.edu/howto/howto-windows-n-unix-homedirs.html

and follow the instructions to log into your Windows/Unix account on the *work* computer.

Next, set up a workspace folder in your home directory where you will keep all your Java files. This should be in

```
z:\\eclipse\workspace
```

Note that `z:` is the drive that Windos binds your UNIX home directory.

Start the Eclipse application.
DO NOT check the box that asks if you want to make this the default workspace for eclipse

Starting a new Project

- In the **File** menu select **New** select **Project**.
- In the pane that opens, under **Java** wizard select **Java Project**.
- Name the project *Project1*
  You can select a different name, but here we will refer to this project as *Project1*.
- In the bottom part select **Create separate source and output folders** and click on **Next**.
- In the next pane just hit **Finish**.
- Now in the **Package Explorer** pane there should be *Project1*. Click on the triangle or the plus sign on the side to open up the sub-parts, and do so again next to **src** line.
- Download the file EclipseLab.zip to the desktop and un-zip it. Ask for help if you do not know how. You should now have a folder named *EclipseLab* with three folders in it: *Book*, *BlobWorld*, and *UFO*.

  The first one contains two simple classes *Book.java* and *BookTests.java* designed to get you started. The second one *BlobWorld* has three files, *Blob.java* and *TimerTests.java* that illustrate the use of the *world.jar* library as well as the *world.jar* library itself. The third folder contains nearly identical files to the ones you worked with in your last lab.

  You will start working with the first folder.
- Highlight the **src** in the **Package Explorer** pane and select **Import**.
- Under **Select an import source** choose **File System** and click on **Next**.
- Next to **From directory** click on **Browse** and select the folder *Book*.
- Highlight the *Book* in the left pane, then select both files in the right pane.
- Leave all other selections unchanged and click on **Finish**.
- You should be back in the main Eclipse view. In the **Package Explorer** pane under the **src** in your *Project1* there should be a **default package** with the two files in it. Open both files.
- Right-click on *BookTests.java* and select **Run as Java Application**.

- The program should run and produce output in the **Console** window on the bottom. However, the window is very small. If you double-click on any window tab in the Eclipse workspace, it will get resized to cover the whole Eclipse pane. Double-clicking on its tab again restores it back to the original view. Try it with the source files as well.

6.2 Learn to manage your files and save your work.

You noticed that instead of using one file to keep all of our work we now have two different files. Java requires that each (*public*) class or interface is saved in a separate file and the name of that file must be the same as the name of the class or interface, with the extension *.java*. That means, you will always need several files for each problem you are working on.

First, modify the files you were given by adding two more examples of books to the *BookTests* class and showing the data in the main test driver. Run your program.

Now save all your files as an archive. Go to the *workspace* subdirectory of your *eclipse* directory and find the directory *Project1*. Make a .zip archive of the files in the *src* subdirectory and save the archive in a folder where you keep your work.

You can also create an archive of your project by highlighting the project, then choose **Export** then select **Zip archive**. Eclipse will ask you for a folder where to place the zip file and will let you choose the name for the zip file.

Your project will remain in the Eclipse workspace, but now you have saved a copy that will not change as you keep working.

6.3 Learn the basics of the use of visibility modifiers in Java.

Add a class *Author* that contains the information about author's name and age and modify the class *Book* to refer to an object in the *Author* class. Of course, you need to define a new file with the name *Author.java*.

Notice that all declarations in the project files start with the word *public*. These keywords represent the *visibility modifiers* that inform the Java compiler about the restrictions on what other programs may refer to the particular classes, fields, or methods.

Declare the fields *name* and *age* in the class *Author* to be private. Now design a method *sameAuthor* to the class *Book* that consumes a name of the author and determines whether the book was written by an author with

the given name. Write your examples as comments for now. We will turn them into tests in the next part.

You should fail in making this method work. Run it. You will see the message **Error in a required project. Continue launch?**. At times the compiler is smart enough to fix small errors and hitting **OK** works just fine. In this case, hit **Cancel**. The program launch stops and it looks like nothing happened. Go to the tab **Problems** in the bottom pane and see what the problems are. You should see the message *The field author.name is not visible* (or something similar). The error was probably signalled in your code already. Clicking on the red cross mark to the left or the erroneous statement pops up message indicating what is wrong, and even offers suggestions for fixing the problem, whenever possible.

The problem is, that you no longer can see the field *name* in the class *Author*. The class *Author* does not let you see how the author's name is represented in its class. For all we know, it could be a list of integers that give you the position of each letter in the alphabet, so that an author with the name *Bach* would have his name encoded as a list (2 1 3 7). However, we can let the outside world find out whether this author's name is the same as the given *String*. Design a *public* method *sameName* to the class *Author* that determines whether *this* author has the same name as the given *String*.

Modify the previous method to use this helper method to solve the problem.

6.4 Learn the basics of writing test cases in Java.

We are now on our own - with no help from ProfessorJ to show us nicely the information represented by our objects, or to provide an environment to run our test suite.

Viewing the data definitions

To make it possible to view the values of the fields for the objects we define, we add to each class a method *toString*() that produces a *String* representation of our data. Java allows us to use the + operator to concatenate two *String*s - it is much less messy than using the *concat* method we used earlier.

The simplest way for defining the *toString* method for the class *MyClass* is:

```
public String toString(){
  return "new MyClass(" + this.field1 + ", "
                        + this.field2 + ")";
}
```

Our example for the class *Book* shows a more elaborate version that gives us not only the value of each field, but also its name.

Java provides a *toString*() method for the class Object, but it typically does not give us the information we are interested in, and so we override the Java *toString*() method.

You must override the method *toString*() for every class you design, even if, at times, it may show only a portion of the data represented by the instance of the class.

Designing tests

Our goal when designing tests is to make sure that we can tell easily not only that some tests failed, but also which test failed.

Read the code that tests the method *before*. It prints out a *String* that consists of the names of the tests and the results of the tests. Convert your examples for the tests for the methods *sameAuthor* and *sameName* into similar tests and run your code again.

Save your results as a .zip file.

6.5 The World

Our projects that extended the *World* contained three *import* statements, indicating that we need to use classes defined in three different libraries written by someone else. Before we start a project that uses these libraries, we need to make sure that the libraries are saved in a known location and that the projects that need them will be able to find them.

Managing the Libraries

- First, create a folder *EclipseJars* in the same folder where you have the *Eclipse* workspace. (This is our convention, not an *Eclipse* requirement.)
- Copy into this folder the three library files *draw.jar*, *colors.jar*, and *geometry.jar*.

- In the *Project* menu select *Properties*.
- In the left pane select *Java Build Path*
- In the top menu line select *Libraries*
- On the right select *Add Variable* .... A pane with title *New Variable Classpath Entry* will open.
- Click on *Configure Variables...*
- Click on *New* to get the *New Variable Entry* pane
- Enter *draw* as *Name* and click on *File...* to select the *draw.jar* file in your *EclipseJars* directory.
- Hit *OK*. A new entry should be visible under the *Classpath Variables*.
- Click again on *Configure Variables...* and follow the same steps to add the file *colors.jar* to the *Variables*, and to add the file *geometry.jar* to the *Variables*.
- Hit *Cancel* to get back to the main *Eclipse* environment.

  From now on all your projects will be able to use these libraries.

Configuring a Project with the World Library

Start a new project *BlobWorld*. Import the .java files from the *BlobWorld* folder. Notice that the files are marked with a number of errors. You need the *World* library.

To work with the libraries you need to add the three *Variable*s you defined earlier to this project. The process is similar to what you did earlier:

- In the *Project* menu select *Properties*.
- In the left pane select *java Build Path*
- In the top menu line select *Libraries*
- On the right select *Add Variable* .... A pane with title *New Variable Classpath Entry* will open.
- Click on *draw* entry in the list of available *Variables* and hit *OK*.

- You are back in the pane where you started adding a variable, but now, the entry for *draw* is available.

  Repeat the last two steps for the *colors* and *geometry Variables*.

- When you are done, hit *OK* to get back to you project environment.

You can now run your *BlobWorld* project. The key controls the movement of the ball, but the timer also moves the ball randomly on each tick. The user interface is nearly the same as we have seen in ProfessorJ.

Make sure you can run the project and see how it is designed.

6.6 Quiz

6.7 Pong Game

Create a new project named PongGame and import all files from the PongGame you designed in Lab 4. Add the libraries (variables) to the project and run it. You may need to make small changes in your code:

- Rename the file *Ball.java*. All Java file names must have the same name as the first class defined in the file.

- Add to the imports on the top the following: *import java.awt.*;*

  so we can use Java colors.

- Add the *public* visibility modifier to the four methods declered in the abstract class *World* that we override in the class *PaddleWorld*: *draw*, *erase*, *onKeyEvent*, and *onTick*.

- Modify the *draw* methods in the classes *Ball* and *Paddle* so that the argument they take is *World w* and the method body starts with *return w.theCanvas.draw*(. . . .

- Modify the *draw* method body in the class *PaddleWorld* to be similar to the *draw* method in the class *Blob*:

```
boolean draw(World w){
  return w.theCanvas.drawRectangle(...);
}
```

i.e., supply *this* world as argument to the *draw* methods for the *World* components and then draw the shapes on the instance of *theCanvas* in the given *World*.

Finally, add the *Examples* class to the project, then right click on the *Examples* class to run the program.

Spend the rest of the lab adding new features to the Pong game.