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 Windows 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 four folders in it: *Book*, *Tester*, *BlobWorld*, and *UFO*.

  The first one contains two simple classes *Book.java* and *BookTests.java* designed to get you started.

  The second one *Tester* ads the file that provides the test harness code and an example of its use.

  The folder *BlobWorld* has three files, *Blob.java* and *TimerTests.java* that illustrate the use of the *world* library that differs from the *world* library in DrScheme/ProfessorJ only in the fact that it requires that the method names (and the fields) be declared as *public*. We will get back to this soon.

  The fourth folder *WorldLibrary* has the *World* library that works with the full Java in the Eclipse project..

  You will now 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. 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. Until you learn how to design the test suite yourself, you must declare as public every field that is involved in determining equality of two different instances of this class.

Declare the fields *name* and *age* in the class *Author* to be public. Make example of data in the *BookTests* class and modify the examples of the *Book* class correspondingly.

Design the method *younger* that determines whether the author of one book is younger than the author as another book.

Add tests for the method to the *BookTests* class, following the technique already illustrated there.

6.4 Learn the basics of writing test cases in Java.

Without the tools we built you would now be on our own - with no help from ProfessorJ to show you nicely the information represented by our objects, or to provide an environment to run our test suite.

We will learn how to do this - gradually. Until then, the *tester* package lets us both design tests quite easily, and print the results formatted for human consumption.

Viewing the data definitions

The class *BookTests* shows the simplest way to display information about out data. It is sufficient for simple data, but becomes quite complex when our data represents a list of five items, each with three fields, where one of them is an abject with two fields — not an uncommon situation in our programs.

Designing tests using the Tester test harness

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.

Download and unzip the **Testing.zip** file. Add to your project the file *Examples.java* from the **Testing.zip** file. Look at its contents.

It starts with import tester.*; Read the file and identify its parts:

- The class *Examples* must implement *Testable*:

  ```
  //-----------------------------------------------------
  // Examples class for the Book class
  public class Examples implements Testable{
  ```

- Next we define the data to be used in tests:

  ```
    public Book book1 = new Book("DVC", 2002);
    public Book book2 = new Book("Beach", 1999);
  ```

- Next we define the tests in a method that implements the *Testable* interface:

  ```
    // combine all tests
    public void tests(Tester t){
      t.test("Book before a", book1.before(2000), false);
      t.test("Book before b", book2.before(2000), true);
    }
  ```

  Each test case has a name - that way we can see which of the tests failed. Each test is an invocation of the method *test* in the class *Tester*, using the given instance of *Tester*.

  The method requires three arguments, the name of the test, the actual value, and the expected value. It compares the given values and records the result of the test for the final report.

- Next we define the method that manages the test run. It first creates an instance of the *Tester* class. Then it invokes the method *runTests* that in turn invokes the *tests* method in this class, using the new instance of the *Tester* class as its argument. So the results of all tests are recorded by our instance of the *Tester* class.

  ```
    // ---- generate the test reports ----------------
    // add at the end of the Examples class
    public void go(){
      // create a new Tester to record all test results
      Tester tester = new Tester();

      // run all tests
      tester.runTests(this);
  ```

- We can now report the tesrt results, either the short version that reports only the failures:

```
// print a short report - of only the failed tests
System.out.println("Invoking t.testReport: \n");
tester.testReport();
```

- Or the full report that shows the actual and expected values in all tests:

```
// print full report - all expected/actual values
System.out.println("---------------------------");
System.out.println("Invoking t.fullTestReport:\n");
tester.fullTestReport(); }
```

- Finally, we need the *main* method with the header *public static void main*(*String*[]*argv*) that creates a new instance of the *Examples* class, prints all fields defined in this class and then invokes the *go* method defined above:

```
public static void main(String[] argv){
  System.out.println("In the Examples class:");

  Examples e = new Examples();

  System.out.println(
    "Show all data defined in the Examples class:");
  System.out.print(Inspector.makeString(e));

  System.out.println("\n\n----------------------");
  System.out.println("Invoke tester.runTests(this):");
  e.go(); } }
```

6.4.1 Managing the Libraries

The `import tester.*;` statement indicates that someone somewhere has defined a library that contains the code for the *tester* classes that are combined into the *tester* **package**. 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.

- 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 file *tester.jar*. *Note* Later we will also add the libraries for the *World*: *draw.jar*, *colors.jar*, and *geometry.jar* to this folder.
- 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 *tester* as *Name* and click on *File...* to select the *tester.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 files *draw.jar* to the *Variables*, *colors.jar* to the *Variables*, and to add the file *geometry.jar* to the *Variables*, once we get to working with the *World* library.
- 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

In our project *BookTests* after we added the *Examples.java* file, the *Examples* file is marked with a number of errors. It needs the *tester* library.

To work with the libraries you need to add the *tester Variable* 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 *tester* entry in the list of available *Variables* and hit *OK*.
- When you are done, hit *OK* to get back to you project environment.

You can now run the code by selecting *Run as Java Application* button when the *Examples* class is in the main window.

Add more tests and examples of data to the *Examples* class and make sure your program runs 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.

Follow the same steps as before to define three classpath variables, one for each of the three packages: *draw.jar*, *colors.jar*, and to add the file *geometry.jar*.

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.

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.

Spend the rest of the lab adding new features to the *BlobWorld* game.