8 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 1.4 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.

8.1 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.

Now go to the web page:

http://www.ccs.neu.edu/howto/eclipse/howto-windows-eclipse.html

on the *documentation* computer and set up your Eclipse working environment. Ask for help if anything is not clear.

Start your Eclipse and follow the **Welcome Tutorial** to **Build a simple Java application**.

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 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 sould now have a folder named *EclipseLab* with three files in it: *ISame.java*, *Book.java* and *Booktests.java*.

- Highlight the **src** in the **Package Explorer** pane and in the **File** menu 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 *EclipseLab*.

- Highlight the *EclipseLab* in the left pane, then select all three 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 three files in it. Open all three files.

- Highlight *Project1* and select **Run** in the **Run** menu.

- In the **Create, manage, and run configurations** select **New**. Under **Project** you should see *Project1*. If you do not, browse and find it. Give a name to this configuration. We choose the name *Project1*.

- Click on the **Search** button next to **Main class**. It should prompt you for the class *Booktests*. Hit **OK**.

- Back in the **Create, manage, and run configurations** click on the **Run** button.
- 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.

8.2 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 three 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.

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

8.3 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 problem 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 ba 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 slove the problem.

8.4 Learn the basics of writing test cases in Java.

We are now on our own - with no help from ProfessorJ to shows 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 Strings - 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 define our own.

You must define 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.

8.5 Books and Authors

Start a new project. You may import the files that represent books, authors, and your **BookTests.java** file as a starting point.

We now want to represent the following information. In the bookstore we have a list of all books, each book has a list of authors, because we know some books have more than one author. The authors are represented by a class as well, as there is quite a lot we would like to know about each author. Specifically, we would like to know the list of books this author wrote.

1. Design the class hierarchy that represents the list of books and the list of authors as described above. Drawing the class diagram is the best - as we will run into problems when defining the constructors.

2. Write down examples of books and authors, as well as lists of books and authors - in English. Make sure to include an example of an author who wrote more than one book, as well as an example of a book that has more than one author.

3. Define the classes that represent this hierarchy, but make the constructor for the class *Author* initialize the list of books the author wrote to the empty list. This constructor consumes only two arguments, the author's name and the year of birth.

4. Add *toString*() methods to all concrete classes. However, in the class *Author* display only the titles of the books the author wrote. You may need a helper method in the classes that represent the list of books.

5. Make examples of the data that can be represented in this class hierarchy, leaving authors with empty lists of books they wrote. Add the statements that will display the values of your examples as Strings.

6. Design the method *addBook* in the class *Author* that adds the given book to the list of books this author wrote. This is the first method that does not produce a value (*void* is used to indicate the return type for such methods). The work of the method is accomplished through side-effects, in this case through assigning a new value to the field *books* for **this** *Author* object.

   Think of how you would test such method.

7. Now extend your earlier examples by adding to your authors the books they wrote. See that the method works by displaying the contents of the *Author* objects using the *toString*() method.

8. In class we have modified the constructor for the class *Book* so that it would automatically add **this** book to the instances of all authors in the list of this book's authors. Make this change and repeat your 'visual tests'.

9. Design several methods that asks typical questions about the contents of the list of books and the list of authors, such as

   - How many authors wrote this book?
   - How many books did this author write?
   - Did this author write a book with another (given) author?