6 Starting in Eclipse; Understanding Constructors

6.1 Eclipse IDE and the tester library

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 standard Java programming language. The environment provides an editor, allows you to organize your work into several files that together comprise a project, and has a compiler so you can run your programs. Several projects form a workspace. You can probably keep all the work till the end of the semester in one workspace, with one project for each programming problem or a lab problem.

There are several step in the transition from *ProfessorJ*:

1. Learn how to convert *ProfessorJ* programs to programs that run in Java, using the *tester* library.
2. Learn to set up your workspace and launch an Eclipse project.
3. Learn to manage your files and save your work.
4. Learn the basics of the use of visibility modifiers in Java.
5. Learn the basics of writing test cases using the *tester* library.

ProfessorJ vs Java.

The programs you have written so far follow the specification for the full *Java* language, with two exceptions:

- The test cases in ProfessorJ use the special form `check --- expect` that is not available in *Java*. Instead, we provide the *tester* library that allows you to write the tests in a similar way. The `tester` library reports on the failed test cases and provides a display of all data defined in our `Examples` class.
- When a class implements an interface which includes method declarations, every method definition in the class that implements a method declared in the interface must be annotated with the `public` visibility modifier.

We provide a simple program (written in Java) that reads your *ProfessorJ* program and produces a new version with the test cases converted to using the `tester` library in *Java*.

Our goal is to take a *ProfessorJ* program, convert it to *Java* and run it in the *Eclipse* IDE. We will start with a fairly complex program you already know. Download the program *bookstore-methods4.java* from the lecture notes and save it in some temporary directory. In the instructions on the web page use this program every time it refers to *books-authors.java*.

Learn to set up your workspace.

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:\\...\EclipseWorkspace
```

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

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

```
z:\\...\EclipseJARs
```

Start the Eclipse application.

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

On Your Own

The libraries you will need as well as the converter program are available at a public web site at:

http://www.ccs.neu.edu/javalib/

Follow the link to *Conversions*, and to *Eclipse*.

The instructions guide you through building a *Converter* project, the conversion of the *bookstore-methods.java ProfessorJ* program to a *Java* program, and in learning how to run the new *Project*.

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.

You see that the output is very similar to what we saw in *ProfessorJ*.

Learn to edit and save your work.

First, modify your file *bookstore-methods4-eclipse.java* adding two more examples of books to the *Examples* class. Run your program.

You can 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.

Learn to edit the program and design the test cases.

In the class `Book` design the method `cheaperThan` that determines whether the sale price of the book is less than the given amount.

Add tests for the method to the `Examples` class, following the technique already illustrated there.

Designing tests using the Tester test harness

In the *Conversions* page of the *javalib* web site click on *Overview*. Use it as a guide for how to design test using the *tester* library. For a full coverage of all possible types of test and the use of the *tester* library, follow the *Tester* link from the main *javalib* site.

Include in your program a couple of test that you know will fail and observe how the errors are reported.

6.2 Understanding Constructors: Data Integrity; Signaling Errors

Goals

In this part of this lab you will practice the use of constructors in assuring data integrity and providing a better interface for the user.

Designing constructors to assure integrity of data.

We start with the `Date` class we may use to check for overdue books.

```
// to represent a calendar date
class Date {
  int year;
  int month;
  int day;

  Date(int year, int month, int day){
    this.year = year;
    this.month = month;
    this.day = day;
  }
}
```

and a simple set of examples:

```
class Examples {
 Examples() {}

  // good dates
  Date d20060928 = new Date(2006, 9, 28);     // Sept 28, 2006
  Date d20071012 = new Date(2007, 10, 12);    // Oct 12, 2007

  // bad dates
  Date b34453323 = new Date(3445, 33, 23);
  }
```

- Create a project `Date` in the Eclipse and add a new file named *Examples.java*. Copy into this file the definition of the class `Date` and the class `Examples`.
- Import the `tester` library and add the `tester.jar` to the project as external JAR. Now run the project.
- Look at the third example of a date.

  Of course, the third example is pure nonsense. Only the year is possibly valid - still not really an expected value. To validate the date completely (taking into account all the special cases for different months, as well as leap years, and the change of the calendar at several times in the history) is a project on its own. For the purposes of learning about the use of constructors, we will only make sure that the month is between 1 and 12, the day is between 1 and 30, and the year is between 1000 and 2200.

- Did you notice the repetition in the description of the valid parts of the date? This suggests, we start with the following methods:

  - method `validNumber` that consumes a number and the low and high bound and returns true if the number is within the bounds (inclusive).
  - methods `validDay`, `validMonth`, and `validYear` designed in a similar manner.

  Design at least one of these methods - you can finish the others at home.

- Once you have done so, change the constructor for the class `Date` as follows:

```
Date(int year, int month, int day){
  if (this.validYear(year))
    this.year = year;
  else
    throw new IllegalArgumentException("Invalid year in Date.");

  if (this.validMonth(month))
    this.month = month;
  else
    throw new IllegalArgumentException("Invalid month in Date.");

  if (this.validDay(day))
    this.day = day;
  else
    throw new IllegalArgumentException("Invalid day in Date.");
}
```

  This example show you how you can signal errors in Java. The class `IllegalArgumentException` is a subclass of the `RuntimeException`. Including the clause

```
throws new ...Exception("message");
```

in the code causes the program to terminate and print the specified error message. Later we will learn how we can customize the error reporting and also how to respond to errors without terminating the program execution.

- Make additional examples with invalid day, invalid month, and invalid year. Run the program, then comment out one invalid example at a time, to see that all checks work correctly.

Overloading constructors to provide flexibility for the user: providing defaults.

When entering dates in the current year it is tedious to always have to enter `2008`. We can make avoid the need to type in the year by providing an additional constructor that requires the user to give only the day and month and assumes that the year is the current year (`2008` in our case).

Remembering the *single point of control* rule, we make sure that the new **overloaded** constructor defers all of the work to the primary **full** constructor:

```
Date(int month, int day){
  this(2008, month, day);
}
```

Add examples that use only the month and day to see that the constructor works properly. Include examples with invalid month or year as well. (Of course, you will have to comment them out.)

Overloading constructors to provide flexibility for the user: expanding the options.

The user may want to enter the date in the form ”Oct 20 2006”. To make this possible, we can add another constructor:

```
Date(String month, int day){
  this(1, day);        // make an instance with a wrong month
  if (month.equals("Jan"))
    this.month = 1;
  else if ...

  else
    throw new IllegalArgumentException("Invalid month in Date.");
}
```

To check that it works, allow the user to enter only the first three months (”Jan”, ”Feb”, and ”Mar”). The rest is tedious, and in a real program would be designed differently.

Finish the work at home and save it as a part of your portfolio.

6.3 Converting a larger program to an Eclipse project

When the program gets larger, we no longer want to keep all class definitions in one file. Typically, in Java every class or interface is defined in its own file, though at times we may group together related classes and interfaces.

- Create a new project *Bookstore2*.
- For each class or interface in the file *bookstore-methods4.java* create a new file with the name of the class followed by `.java`. Make each class and its constructor `public`. Make the interface `ILoB` `public`.
- Do the necessary corrections until all errors disappear.
- Now create a new file `Examples.java` and copy into if the definition of the `Examples` class.
- Add the statement

  ```
  import tester.*;
  ```

  at the beginning and add the `tester.jar` library to the project.
- You can now run your project. Ask for help if *Eclipse* does not recognize the project you are trying to run.