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 Java 1.5 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 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 using the *tester* library.

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 files EclipseLab1.zip to the desktop and un-zip it. Ask for help if you do not know how. You should now have a folder named *EclipseLab1* with three files in it: *Examples.bjava* — a simple program that runs in *ProfessorJ Beginner* language, *Examples.java* — your first program to run in the Eclipse IDE, and a test library *tester.jar*.

  The first one contains two files *Examples.bjava* and *Examples.java* designed to get you started.

  The second one *Tester* ads the file that provides the test harness code.
- Open the file *Examples.bjava* in *ProfessorJ Beginner* language in DrScheme. Read the code quickly (you have seen this kind of program a number of times before) and run it once.

The second file, *Examples.java* is an equivalent program that runs in Eclipse with the *tester* library replacing the testing library we used in *ProfessorJ*.

Start working with the *Examples.java* file.

- 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 *Eclipse-Lab1*.
- Highlight the *EclipseLab1* in the left pane, then select the *Examples.java* file 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 file *Examples.java* in it. Open the file.

  Now you can see how the program differs from the one you ran in *ProfessorJ*.

- In our next step we select the libraries that our program needs. For now, we need only one library, saved in the *tester.jar* file.

  – In the **Project** menu on the top select **Properties**.
  – On the left hand side select **Java Build Path**.
  – On the right hand side select **Add External JARs...**
  – Browse to locate your tester.jar file. (You should keep this file in an easily accessible directory that will not change over time.)
  – Select **OK** on the bottom right.

  You should notice that the red marks that highlighted errors in your *Examples.java* file have disappeared. Your project is now ready to run.

- Right-click on *Examples.java* and select **Run as Java Application**. If it asks for main, select *Main*.

- 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 *Examples.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.

Visibility modifiers in Java.

Notice that the *Examples* class definition starts with the word *public*. Java requires that exactly one class or interface in each file is declared as *public*, and the name of the file must match the name of this *public* interface or class.

The *public* keyword represents the *visibility modifier* that informs the Java compiler about the restrictions on what other programs may refer to the particular classes, fields, or methods. We will learn about these *visibility modifiers* soon.

Learn to edit the program and design the test cases.

Change the class *Book* so that contains the information about the date when the book is due to be returned to the library. Of course, you need to add the *Date* class.

Design the method *isOverdue* that determines whether the book is overdue on a given day. Assume throughout that each month has 30 days and there are no leap years.

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

Designing tests using the Tester test harness

The *Examples* class is very similar to what we have seen in *ProfessorJ*. let us look at the differences:

- It starts with

  ```
  import tester.*;
  ```

  identifying the library we will need - just as we did with the *draw*, *geometry*, and *colors* libraries.

- The class *Examples* must be declared *public* and must have a *public* constructor:

  ```
  //---------------------------------------------------
  // Examples class for the Book class
  public class Examples implements IExamples{
    public Examples(){}
  ...
  }
  ```

- Finally, class *Examples* must implement *IExamples* interface that consists of a single method *public void tests*(*Tester t*);

  We define the *tests* in a method that implements the *IExamples* interface as follows:

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

  Each test is an invocation of the method *checkExpect* by an instance of the class *Tester*.

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

  The third parameter does not have to be supplied, but it helps us in remembering what was the purpose of that test.

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 used 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:

```
public class Examples {
  public 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*. Delete from the *Examples* class all examples of data and test cases that deal with books and authors - leave only the examples of dates and test cases for any method you have designed for the *Date* class.

- Import the *tester* library and add the *tester.jar* to the project as external *JAR*. Now run the project.

- Add the examples above and run the project again.

  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:

```
public 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:

```
public 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:

```
public 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.

- Download the program *list-of-songs-acc.ijava* that is the solution to the second problem of last week's lab. Create in Eclipse a new project with the name *Songs*.
- We want to *translate* the program into an Eclipse project. The original program consists of four classes: *Song*, *MtLoS*, *ConsLoS*, and *Examples* as well as one interface *ILoS*. We will divide this program into three files: *Song.java*, *Examples.java* and *ILoS.java*.

  Copy the class definition for he class *Song* into a new file named *Song.java*. Make the class and its constructor *public*.
- Copy the classes that represent a list of songs into another new file names *ILoS.java* and make the interface *ILoS public*.
- Look at all the problems Eclipse signals. In Java, every method defined in an interface is considered *public* even if it does not say so explicitly. Therefore, every time you implement a method defined in an interface, you must give it the *public* visibility.

  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 *import tester.∗;* statement at the beginning and add the *tester.jar* library to the project.

- Make the *Examples* class implement the *IExamples* interface and set the visibility for both the class and its constructor to *public*.

We are almost done. The only remaining task is to convert the test cases from *ProfessorJ* to tests managed by the *tester* library.

Here is an example of how it works. The original tests for the method *count* were defined as:

```
// * Count Examples
boolean countTests =
    ((check this.mtlos.count() expect 0) &&
     (check this.list1.count() expect 1) &&
     (check this.list2.count() expect 3));
```

They translate to the *tester* tests as follows:

```
// * Count Examples
void countTests(Tester t){
    t.checkExpect(this.mtlos.count(), 0);
    t.checkExpect(this.list1.count(), 1);
    t.checkExpect(this.list2.count(), 3);
}
```

- Convert the remaining test cases in a similar way.
- We still need to implement the interface *IExamples* that is defined as follows:

  ```
  interface IExamples{
    public void tests(Tester t);
  }
  ```

  The method *tests* will group together all the test methods we have designed as follows:

  ```
  // Run all tests:
   public void tests(Tester t){
      countTests(t);
      totalSizeTests(t);
      ...

   }
  ```

- Finish the design of the *tests* method and run the program.