Designing Classes with FunJava

1 Data Definitions and Setup

In this lab we will focus on the use of SVN, Eclipse, and the creation of data definitions and examples in a class based object-oriented language (like Java, but not quite).

We will work in a "professional" *Integrated Development Environment (IDE)* (**Eclipse**) using a language library called *FunJava*.

1.1 Homework Submission and `SVN`

You should have logged in using your CCIS credentials. If not, then shame on your Lab leader!

Open your web browser to the following web page for information on how to access your Unix/Linux directories. (likely not necessary, if you trust us)

```
http://howto.ccs.neu.edu/howto/accounts-homedirs/home-
   directory-access-on-linux-and-windows/
```

Navigate your way to the `Z:` drive. If you can't find "`Computer`" on the desktop, then double-click the `Recycle Bin` and click on `Computer` in the left pane, then double-click the network drive `myhome ... (Z:)`.

You should see the various directories and files that make up your Unix home directory. Right-click on the `classes` directory and select `SVN Checkout...` from the pop-up menu.

Visit this page for a quick discussion of HW Handin:

```
http://www.ccs.neu.edu/course/cs2510/Assignments/
         Handin/
```

From there you should be able to locate your HW partern's information, and the number of your pair. Back to the checkout... for the *URL of the repository*, enter:

```
https://trac.ccs.neu.edu/svn/cs2510s11/pairXXX
```

where `XXX` is your pair number from the HW submission page. For the *Checkout directory*, enter:

```
Z:\classes\CS2510-Workspace
```

Or name it something you like better. Click OK. It should ask you for your CCIS username and password. Once you enter those a and it's finished, click OK, then navigate into the `classes` directory. You should be able to see your new SVN workspace directory!

Create another directory in `classes` where you will keep all library (JAR) files, call it *EclipseJars*. For the rest of the Lab/term we will refer to these two directories as *EclipseWorkspace* and *EclipseJars*. Keep the file/explorer window open, we'll be using it later in the Lab.

1.2 Eclipse IDE and `FunJava`

Eclipse includes an editor and allows you to organize your work into many files that together make up a *project*. It has an "*incremental*" compiler that so you can edit and run your programs while getting relatively fast error feedback. Your Eclipse *workspace* can contain many projects, so you should be able to keep all your work in one workspace, with one project for each assignment or lab.

Setting up your workspace

Start Eclipse. It should ask you where you want your workspace to be... so enter (or click `Browse` and navigate to) the workspace directory you created (saw that coming). Feel free to check the "*Use this as the default...*" box. Click OK.

Once Eclipse starts, close the annoying *Welcome* screen if it comes up.

Zeroth things First

There's a few settings we need to get out of the way before you start. If/when you work from your own computer you'll have to adjust these settings too, otherwise the graders might be angry... and you wouldn't like them when they're angry.

Select `Preferences` under the `Window` menu and change the following settings. You should also be able to follow along with your friendly Lab

Leader.

1. Type "`tab`" in the search box at the upper-left to minimize the available selections.
2. Select "`Text Editors`" on the left. Make sure the "`Insert spaces for tabs`" check-box is checked.
3. Select "`Formatter`" on the left. Click the "`Edit...`" button, then choose the "`Indentation`" tab at the top. Change the "`Tab policy`" to "`Spaces only`".
4. When you say OK it will force you to create a name for the profile... you can just say "`Mine`" or some other cool name.

Great... now let's get on to Java-like programming!

Create a new project, and Check it into SVN

- Select `New > Java Project` from the `File` menu. Enter the name "`last-Lab-02`" but replace "*last*" with your last name. Under `Project layout` make sure the *Create separate folders for sources and class files* radio button is selected, then click *Finish*.
- Your project should show up in the *Package Explorer* on the left (feel free to cheer at this point).

We will be submitting HWs via SVN through the `Trac` repository workspace that you just setup.

Go back to your file/explorer window, and enter your workspace folder Your project directory should be visible... right-click on it and select `TortoiseSVN > Add...`. If you want you can uncheck some of the weird entries, then click OK, then click OK again, when the action completes.

Right-click the project directory again and select `SVN Commit...`. Enter a message like "`First checkin`", then click OK. In general the msage should be meaningful, so that your partner can tell what you were doing when you changed things. After the action completes you now have all you need to submit assignments!

See the *HW Handin* link off the course webpage for more information and details on HW submission and organization.

A Java-like Experience

1. Now that you're an Eclipse and SVN expert, grab the library (JAR) we need for this Lab from the lab's index page.

   `FunJava-1.jar`

   Save it in your *EclipseJars* directory . The JAR provides output and testing functionality you saw in lecture and a tool for restricting the language a bit, for now. The "`-1`" in the name is there because we will be updating (hopefully improving) the library throughout the semester, so the number will increase as we release changes.

   Right-click on your project directory in the *Package Explorer* pane and select `Build Path > Add External Archives...`. Navigate to your `EclipseJars` directory and select your `FunJava.jar`.

2. Add the `Shapes.java` file to your project

   - Download the file `Shapes.java` form the lab index page and save it into the `src` directory within your project directory, in your new workspace. Then *right click* on your project in the *Package Explorer* and select `Refresh`. You can also select the project and hit `F5`.
   - Alternatively, you can save it to a temporary directory and import it using
     `File > Import...`. Select the `General` tab, `File System`, and click `Next`. Browse to your temporary directory, select it, and find your file. Make sure you put it into your project `src` directory.

3. Set up a FunJava run configuration

   - Highlight your project in the in the *Package Explorer* pane.
   - In the *Run* menu select *Run Configurations...*. In the left pane select `Java Application`.
   - In the upper left corner click on the leftmost item (the icon with the plus in the corner). When you mouse over it should show *New launch configuration*.
   - Make up a name for this configuration - usually the same as the name of your project, lab, or assignment.

- The *Project* field should be your current project (e.g., `chadwick-Lab-02`), and in the *Main class* field enter `FunJava`.
- Click on the tab `(x)= Arguments`. In the *Program arguments* text field enter "src/Shapes.java". Make sure to enclose it in quotes. Later, when you define your own program, you will use a different (but just as meaningful) file name that corresponds to the file/assignment you're working on.
- At the bottom of the *Run Configurations* select *Apply* then *Run*.

You should see a few messages in the `Console`, and a display of the `Examples` class with the example instance(s).

If the program does not run and reports that it cannot find `java` or `javac`, open your *Run Configuration* and click on the tab `Environment` then select `New` and enter the following information:

- Name: PATH
- Value: `C:\Program Files\Java\jdk1.6.0_23\bin`, which is the location of the Java Compiler on the Lab machines, though at home yours will likely be different.

Then click OK and try running again.

4. View, Edit, and rerun `Shapes.java`

   - Unfold the `default package` under the `src` directory under your project in the *Package Explorer*.
   - You should see `Shapes.java`; double-click it, and the file should open into an editing tab in the main pane.
   - Add new examples of `Circle` and `Rect` to the `Examples` class. Make sure there are no errors or warnings.
   - Now run the file again. make sure `Shapes.java` is shown in the main pane, then hit the green circle with the white triangle in the top toolbar. You can also click and select your configuration for the drop-down menu to the right of the Run button.

2 Practice with FunJava

Remember all the Rockets we launched (and landed) last semester? Here's a Racket data definition for a simple Rocket:

```
;; A Rocket is:  (make-rocket String Number Boolean)
(define-struct rocket (name height landing?))

;; Example Rockets:
(define r1 (make-rocket "Apollo 11" 150 false))
(define r2 (make-rocket "Gemini 3" 25 true))
```

1. Draw the class diagram for this data definition
2. Create a new file in your Eclipse Lab project named `Rocket.java` and convert the data definition/class diagram into a `FunJava` class definition.
3. Create an `Examples` class, and include instances of `Rockets`. See `Shapes.java` for the necessary format(s).
4. Create a new `Run Configuration` and run the examples.

2.1 Classes containing classes

Rather than just keeping track of the height of a `Rocket`, we might also need it's horizontal coordinate. Let's use a `Posn` to keep track of both. Of course `posn` is built into DrRacket, so the following is mostly just for illustration.

```
;; A Posn is:  (make-posn Number Number)
(define-struct posn (x y))

;; Example Posns:
(define p1 (make-posn -30 150))
(define p2 (make-posn 50 25))

;; A Rocket is:  (make-rocket String Posn Boolean)
(define-struct rocket (name loc landing?))

;; Example Rockets:
(define r1 (make-rocket "Apollo 11" p1 false))
(define r2 (make-rocket "Gemini 3" p2 true))
```

1. Draw the class diagram for these data definitions
2. Add the `Posn` class to your file and modify the `Rocket` class (and constructor) to match the Racket definitions.

3. Create examples of `Posn`s and adjust your `Rocket` Examples accordingly.

4. Run the examples to make sure everything is correct.

2.2 Unions of Classes (of data)

The class of data that represents Rockets is not as interesting as it could be. For instance, the rocket that lifts the space shuttle is *very* different than the one that took Buzz to the moon. And a *toy* rocket might have a price associated, whereas you probably don't want to know how much the space shuttle rockets cost (especially if you pay taxes).

Create an **`interface`** to represent various kinds of rockets... call it `IRocket` (and so do you). The data class of `IRocket` will be made up of three different FunJava **`class`**es... feel free to embellish as you see fit.

- **SelfFlying** which includes the location and the version number of the auto-pilot software.
- **DualEngine** which includes the location, the power of each engine, and the name of the pilot.
- **Toy** which includes only the max-height and price of the rocket.

Notice that the names are distinct from your other classes... you can use the same `java` file, just include new examples in your `Examples` class

1. Draw a class diagram for the class hierarchy that represents these three types of Rockets

2. Design data definitions (i.e., **`class`**es) for each of the definitions (including the interface)

3. Add examples of each kind of `IRocket` to your `Examples` class. Be sure to use `IRocket` as the type of the example fields.

2.3 Self-Referential Class Hierarchies

Remember the Russian-Dolls example from last semester? Brings back fond memories of recursion doesn't it? Let's re-live a little... here's something like the definitions we had last semester:

```
;; An IDoll is one of:
;; - 'solid
;; - (make-doll IDoll)
(define-struct doll (inner))

;; Example IDolls:
(define d1 (make-doll 'solid))
(define d2 (make-doll (make-doll (make-doll 'solid))))
```

- Draw the class diagram for `IDoll`s and the related structures. **Hint**: rather than using a `symbol`, use a class without any fields.
- Transform the diagram into class definitions
- Make examples of `IDoll` and add them to your `Examples` class. Use references to earlier fields (e.g., **`this.`**`doll1`) to build your dolls.

3 Wrapping up, checking in

Last, but not least, save your file(s), and add them to SVN. Right click on the file and choose `TortoiseSVN > Add...`. When you're done select Commit for the entire workspace directory.