Lab 1: Introduction to Eclipse and Simple Data Definitions

Goals: The goals of this lab are to get familiar with our work environment: the Eclipse IDE, the handin-server submission process, the basics of running a program in Java, and program testing framework.

The second part of the lab will focus on practicing data definitions and examples in Java.

Related files:
tester.jar javalib.jar Shapes.java

1.1 Khoury Accounts

You will use the Handin Server to work on your homework sets, to keep track of revisions, and to submit your homework. You will need a Khoury account to use the Handin Server so please go here to apply for an account if you don't already have one.

Important: if you have had a Khoury (or CCS) account in the past, but are not a Khoury student and your account has been deactivated due to inactivity, *do not* apply for a new account. Instead, send an email to `systems at ccs dot neu dot edu` explaining that you're registered for CS2510, that your account has been disabled, and provide your Khoury username and your NUID in the email. You will not likely be able to log in until the Systems team has a chance to process your account, so be patient.

1.2 Introduction

We start with designing data - designing classes of data that are connected to each other in a systematic way, showing that the *Design recipe for Data Definitions* can be used virually without change in a completely different language than we have used in the first part.

The programs we provide give you examples of the progressively more complex data (class) definitions, and illustrate the design of methods for these class hierarchies.

1.3 Eclipse IDE

Eclipse is an integrated (program) development environment used by many professional Java programmers (as well as programmers using other programming languages). It is an *Open Source* product, which means anyone can use it freely and anyone can contribute to its development.

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 steps to getting started:

- Learn to set up your workspace and launch an Eclipse project.
- Learn to manage your files and save your work.
- Learn how to edit your Java programs and run them, using our tester library.

Learn to set up your workspace.

IF YOU ARE USING YOUR OWN MACHINE:

- First, install the Java Development Kit (JDK): download Java 11 from https://www.oracle.com/java/technologies/javase/jdk11-archive-downloads.html appropriate for your machine, and install it. Make sure you choose the JDK option, and not merely the JRE: the latter only enables you to *run* Java programs, but the development kit lets you *develop* them.

  Do not use any version higher than 11. The submission server does not support higher versions, so you may have some problems when you submit.

- Next, download Eclipse from https://www.eclipse.org/downloads/ – the standard Eclipse IDE release is fine, and the Eclipse site should detect your computer's operating system appropriately. When installing it, select Eclipse IDE for Java Developers, for this is what you are becoming!
- Create a folder where you will keep all of your fundies2 files. We will refer to this folder as *cs2510*.
- Inside of the *cs2510* folder, create a workspace folder where you will keep all of your Java files. We will refer to this folder as *EclipseWorkspace*.
- Next, set up a second folder within *cs2510*, where you will keep all of your Java library files.

  We will refer to this folder as *EclipseJars*. Make sure the two folders *EclipseWorkspace* and *EclipseJars* are subfolders of the *cs2510* folder.

- Start the Eclipse application.

  DO NOT check the box that asks if you want to make this the **default workspace** for **Eclipse** if you are working on a lab computer. If you are working at home or using your laptop, you may want to make the selected workspace to be your default. Select your previously created *EclipseWorkspace* folder as your workspace.

The First Project

- **Right click the following two libraries** and select "Save link as..." Save them to your *EclipseJars* folder.

  The libraries you will need are:

  Related files:
  tester.jar javalib.jar

- **Create a project.**

  Back in Eclipse, in the *File* menu select *New* then *Java Project*. In the window that appears in the *Project layout* section select *Create separate folders for sources and class files*. Click Next. **If you see the option to make a module file, please uncheck it**. Select *Finish*, as the other defaults should be fine. Name the project *Lab1*. Avoid using any spaces or special characters in your project and file names in Eclipse. If you see a tab with "Welcome" in the title that has a lot of options, exit out of the tab with the white x.

- **Make the libraries available to the new project.**

  Highlight your *Project* in the *Package Explorer* pane, then right-click it and select "Properties". In the list on the left, select the *Java Build Path*, then select the *Libraries* tab. **NOTE:** If you are using Java 9 or higher, you should see two options in the main area of the form: "Modulepath" and "Classpath". Make sure that you have clicked on "Classpath" to select it. If you do not see these two options, just ignore this step.

  Note: If the pane is not visible, go to *Window* menu, select *Show View...* then *Package Explorer*. You should also select *Show View... Outline*.

  On the right click on *Add External JARs...*

  The *file chooser window* will be shown. Navigate to your *EclipseJars* folder and select all *jar* files you have downloaded.

  Hit Apply and Close.

- **Add the *Shapes.java* file to your project.**

  Related files:
  Shapes.java

  - Download the file `Shapes.java` to a temporary directory or the desktop.
  - *In Eclipse* highlight the *src* box under the *Lab1* in the *Package Explorer* pane.
  - In the *File* menu select *Import...*.

    Choose the *General* tab, within that *File System* and click on *Next*.

- Browse to the temporary directory that contains your *Shapes.java* file and click "Open".

  Click on the directory on the left, then select the *Shapes.java* file in the right pane and once you have checked the "Into folder" is *Lab1/src*, hit *Finish*.

- **View the file `Shapes.java`.**
  - Click on the *src* block under *Lab1* in the *Package Explorer* pane. It will reveal *default* package block.
  - Click on the *default* package block. It will reveal *Shapes.java*.
  - Double click on *Shapes.java*. The file should open in the main pane of *Eclipse*.

    You can now edit it in the usual way. Notice that the *Outline* pane lists all classes defined in this file as well as all fields and methods. It is almost as if someone was building our templates for us.

  - **Adjust the *Eclipse* settings.**

    The Code style section explains how to configure Eclipse to only use spaces instead of tabs, and how to set the editor to show you the line numbers for all lines in the code. If you have trouble, ask a TA for help.

Set up the run configuration and run the program.

- Highlight *Lab1* in the *Package Explorer* pane.
- In the *Run* menu select *Run Configurations...*.
- In the list on the left, make sure "Java Application" is selected. Then, in the top left corner of the inner pane click on the leftmost item. (When you mouse over it should show *New launch configuration*.)
- Select the name for this configuration - usually the same as the name of your project.
- In the *Main class:* click on *Search...*.
- Among *Matching items* select *Main - tester* and hit *OK*.
- Click on the tab *(x)= Arguments*. In the *Program arguments* text field enter `ExamplesShapes`.

  Later, when you define your own program, you will use the class name of your `Examples...` class instead of `ExamplesShapes`.

- At the bottom of the *Run Configurations* select *Apply* then *Run*.

  Next time you want to run the same project, make sure *Shapes.java* is shown in the main pane (and is currently selected), then hit the green circle with the white triangle on the top left side of the main menu.

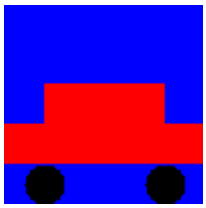
Interface? Implements?

As we fondly remember from CS2500, one of our favorite types of data is union data: enumerations, itemizations, etc. This is covered in Lecture 2, so you can skip to **Problem 1** below if you have not read Lecture 2 yet.

To define union data in Java, we define that union as an *interface* and say each branch of the data *implements* the interface. Read the code in your shapes file: you'll notice the *IShape* interface is defined at the top, and that each class implements it. In the examples section, the examples are given the type of the interface, and are created with the constructor of the specific class. Follow suit when defining your own examples.

Making examples

Add examples of shapes found in the following image (ignore the colors, and note you do not actually have to draw the shapes):



Run your examples class with the tester library to make sure it works.

Simple Data Definitions

Problem 1

Here is a data definition in DrRacket:

```
;; to represent a person
;; A Person is (make-person String Number String)
(define-struct person [name age gender])

(define tim (make-person "Tim" 23 "Male"))
(define kate (make-person "Kate" 22 "Female"))
(define rebecca (make-person "Rebecca" 31 "Non-binary"))
```

Draw the class diagram that represents this data.

Define the class `Person` that implements this data definition and the class `ExamplesPerson` that contains the examples defined above. You should do this in a new *Person.java* file. Right click the default package under *Lab1* and select New > File. Name it *Person.java*.

Run your program to make sure it works.

Data Definitions with Containment

Problem 2

Modify your data definitions so that for each person we also record the person's address. For each person's address we only record the city and the state; each of these should be its own field. Create an `Address` class to contain the address information, then modify the `Person` data definition to include an `Address`.

- Draw the class diagram for this data definition
- Define Java classes that represent this data definition.
- Tim lives in Boston, MA, Kate lives in Warwick, RI, and Rebecca lives in Nashua, NH.

  Make examples of these data and add two more people.

Problem 3 We want to define events which have a title, date, location and host. What types of data should you use for each of these fields? A date should have a day, month and year. A host should be a `Person`.

- Draw the class diagrams for an event data definition.
- Define the class `Event` that implements this data definition.
- Define the class `ExamplesEvent` and create some examples of events.

You should do this problem in a new *Event.java* file. Right click the default package under *Lab1* and select New > File. Name it *Event.java*.

Data Definitions for Unions of Data

Problems 4 and 5 concern material found in Lecture 2. You can attempt them if you like.

Problem 4

A deli menu includes soups, salads, and sandwiches. Every item has a name and a price (in cents - so we have whole numbers only).

For each soup and salad we note whether it is vegetarian or not.

Salads also specify the name of any dressing being used.

For a sandwich we note the kind of bread, and two fillings (e.g peanut butter and jelly; or ham and cheese). Assume that every sandwich will have two fillings, and ignore extras (mayo, mustard, tomatoes, lettuce, etc.)

Define classes to represent each of these kinds of menu items. Think carefully about what type each field of each class should be. Do you need to define any interfaces? Construct at least two examples each of soups, salads, and sandwiches.

Self-Referential Data

Problem 5

The *HtDP* book includes the data definition for *Ancestor Trees*:

```
;; An Ancestor Tree (AT) is one of
;; -- 'unknown
;; -- (make-tree Person AT AT)

;; A Person is defined as above
```

Convert this data definition into Java classes and interfaces. What options do you have for how to translate this into Java? Make examples of ancestor trees that in at least one branch cover at least three generations.