CSE 331 Software Design and Implementation

Homework 0: ***Environment Setup and Java Introduction***
Due: Friday April 5 @ 11pm

Contents:

The purpose of this assignment is to help you set up your development environment, get reacquainted with Java, and familiarize yourself with tools you will use for the rest of the course. Although the assignment description is *long*, we expect that our step-by-step instructions will make doing the assignment less overwhelming than reading it may be.

You are strongly encouraged to complete this assignment first on the instructional workstations (IWS) in one of the Allen Center software labs using exactly the tools we describe here.

If you would like to get more practice with Java, then we recommend walking through Oracle's Java tutorials described under Problem 7. Try to complete Homework 0 first so you can use the tools we describe here when doing the examples in Oracle's tutorial.

You are free to get help from anyone, on any portion of this problem set; that is, the standard collaboration policy does not apply to this homework. You will need to know this material for the remainder of the quarter, however, so make sure you understand the tools and the concepts.

If you are having trouble with finishing this assignment on time, get in touch with the staff immediately so we can try to get you back on track. Here are some steps to take if you get stuck on something in this (or any) homework:

a. check the FAQs and corrections for the homework
b. check the Catalyst forum to see if someone else has posted a similar question
c. *and then,* ask the staff for help.

Getting Started

Before you being this assignment, make sure to run the student-setup script, as explained in the Tools Overview.

Problem 1: Choosing and Setting-Up a Development Environment

You are free to choose your own Java development environment. We strongly recommend using Eclipse for CSE 331 homeworks; we also provide instructions for working from the Linux command line. We focus on the use of the CSE instructional computers. If you plan to work on your own laptop or home machine, we don't promise help for individual configuration issues. Of course, you can use your own machines to access the instructional computers remotely via SSH or other methods.

Once you have chosen a development environment, you should set it up for CSE 331 use. See Starting Eclipse for instructions on how to do this. (Note: if you switch development environments later on in the quarter, make sure to run the appropriate setup instructions again.)

Important: Eclipse is a widely-used Java development environment with a number of excellent features including integration with SVN, Ant, and JUnit - tools that we use in CSE 331. However, instead of using the official Java compiler ("javac") from Oracle/Sun, Eclipse uses an Eclipse-specific Java compiler. Bugs in either compiler are rare, but they do manifest on rare occasions, causing the same code to have different behavior with different compilers. Course staff will compile and grade your code using the `javac` compiler, so you'll need to make sure that your code behaves correctly when compiled with `javac`. One way to verify this is to use the Ant build file we provide for each homework. See the Compiling Java Source Files document for details.

Refer to the Eclipse quick reference to learn about some of the handy features Eclipse provides.

Problem 2: Obtaining files from SVN

Throughout this quarter you will receive starter code and submit your assignments through Subversion (SVN). SVN is a version control system that allows software engineers to backup, manage, and collaborate on large software projects (you will learn more about SVN in section).

For this problem, you should follow the setup instructions in the Verision Control handout, which describe how to "checkout" your CSE 331 SVN repository.

The Verision Control handout also contains instructions for updating, adding, and committing to your repository. You should familiarize yourself with these commands as you will need to use them throughout this assignment and for the duration of CSE 331.

Problem 3: Warm-Up Exercise-HolaWorld

For this problem, you will fix some buggy code we provide.

Editing and Compiling Source Files

See Editing and Compiling Source Files. This part will help you become familiar with how to perform the following basic tasks in your environment of choice: adding new files to your directory structure, compiling Java code, and reading the Java compiler's output (which may indicate errors).

Also see the CSE 331 Java Style Guide. That will familiarize you with standards of Java style, which we will expect you to follow in this class.

Fixing HolaWorld

Try to compile the provided code in `HolaWorld.java`. You should be notified of compilation errors in the file. (And possibly in `hw0/test/RandomHelloTest.java` as well; if so, ignore these for now since we will fix them in the next part.) In particular, the lines:

```
        System.out.println(world.);
```

and:

```
        return SPANISH_GREE;
```

are problematic. (If you're using the the Ant builder, you may only see the second error after you've fixed the first one.) If you are using Eclipse, these errors will be marked with red squiggly lines in `HolaWorld.java`, and `HolaWorld.java` itself will be marked with a red crossmark in the **Package Explorer**.

Fix these errors and run `HolaWorld`.

After you've fixed the errors and run the code, it would be a good time to commit your changes to SVN.

Problem 4: Your first Java class-RandomHello

Create your first Java class with a `main` method that will randomly choose, and then print to the console, one of five possible greetings that you define.

Create the file `RandomHello.java`, which will define a Java class named `RandomHello` that will reside in the Java package `hw0`; that is, its file name is *YourWorkspaceDirectory*`/cse331/src/hw0/RandomHello.java`.

Java requires every runnable class to contain a `main` method whose signature is `public static void main(String[] args)` (for example, the `main` methods in `HelloWorld` and in `HolaWorld`).

A code skeleton for the `RandomHello` class is shown below. Eclipse will generate some of this skeleton for you when you create the new `RandomHello` class.

`RandomHello.java`:

```
package hw0;
/**
```

```
 * RandomHello selects a random greeting to display to the user.
 */
public class RandomHello {

    /**
     * Uses a RandomHello object to print
     * a random greeting to the console.
     */
    public static void main(String[] argv) {
        RandomHello randomHello = new RandomHello();
        System.out.println(randomHello.getGreeting());
    }

    /**
     * @return a random greeting from a list of five different greetings.
     */
    public String getGreeting() {
        // YOUR CODE GOES HERE
    }
}
```

This skeleton is meant only to serve as a starting point; you are free to organize it as you see fit.

No Need to Reinvent the Wheel

Don't write your own random number generator to decide which greeting to select. Instead, take advantage of Java's **`Random`** class. (This is a good example of the adage "Know and Use the Libraries" as described in Chapter 7 of Joshua Bloch's *Effective Java*. Learning the libraries will take some time, but it's worth it!)

Type the following into the body of your `getGreeting()` method:

```
Random randomGenerator = new Random();
```

This line creates a random number generator. (Not a *random number*, which comes later, but a Java object that can *generate* random numbers._ In Eclipse, your code may be marked as an error by a red underline. This is because the `Random` class is defined in a package that has not yet been imported (`java.lang` and `hw0` are the only packages that are implicitly imported). Java libraries are organized as packages and you can only access Java classes in packages that are imported. To import `java.util.Random`, add the following line under the line `package hw0;` at the top of your file (after the `package hw0;` declaration):

```
import java.util.Random;
```

This will import the class **`Random`** into your file. To automatically add all necessary imports and remove unused imports, Eclipse lets you type **CTRL-SHIFT-O** to ***O**rganize* your imports. Because there is only one class named `Random`, Eclipse will figure out that you mean to import `java.util.Random` and will add the above line of code automatically. (If the name of the class that needs to be imported is ambiguous - for example, there is a **`java.util.List`** as well as a **`java.awt.List`** - then Eclipse will prompt you to choose the one to import.)

Using java.util.Random

Read the documentation for `Random`'s `nextInt(int n)` method by going to the Java API and selecting `Random` from the list of classes in the left-hand frame. Many classes also allow you to pull up documentation directly in Eclipse. Just hover over the class or method name and press SHIFT+F2.

Use the `nextInt(int n)` method to choose your greeting. You don't have to understand all the details of its behavior specification, only that it returns a random number from 0 to n-1.

One way to choose a random greeting is using an array. This approach might look

something like:

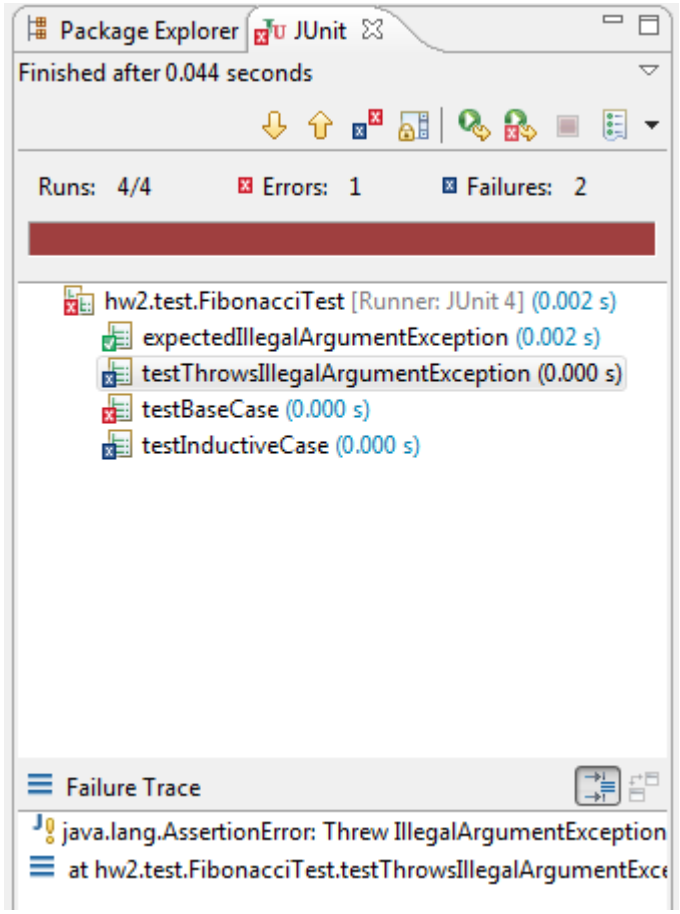
```
String[] greetings = new String[5];
greetings[0] = "Hello World";
greetings[1] = "Hola Mundo";
greetings[2] = "Bonjour Monde";
greetings[3] = "Hallo Welt";
greetings[4] = "Ciao Mondo";
```

The `main` method in the skeleton code above prints the value returned by `getGreeting`. So if you insert code in `getGreeting` to select a greeting randomly, when the class is run it will will print that greeting.

When you are finished writing your code and it compiles, run it several times to ensure that all five greetings can be displayed.

Again, now it would be a good idea to add your new class to version control and commit your code.

Problem 5: Testing Java Code with JUnit



Part of your job as a software engineer is to verify that the software you produce works according to its specification. One form of verification is testing. JUnit is a framework for creating *unit* tests in Java. A unit test is a test for verifying that a given method in a class conforms to its specification. In this problem, we will provide you with a quick overview and simple example of how JUnit works. (The next homework will look more deeply into unit testing.)

Open both hw0/Fibonacci.java and hw0/test/FibonacciTest.java. From the comments, you can see that **`FibonacciTest`** is a test of the **`Fibonacci`** class.

Now run the JUnit test `hw0.test.FibonacciTest`.

A window or panel with a menacing red bar will appear, indicating that some of the tests in `FibonacciTest` did not complete successfully (see screenshot at right). The top pane displays the list of tests that failed, while the bottom pane shows the Failure Trace for the highlighted test. The first line in the Failure Trace should display an error message that explains why the test failed (it is the responsibility of the author of the test code to produce this error message).

As shown in the figure above, if you click on the failure `testThrowsIllegalArgumentException()`, the bottom pane will switch to the appropriate error message. In this example, the first line of the failure trace shows that `Fibonacci.java` improperly threw a `IllegalArgumentException` when tested with zero (0) as its argument. (You may have to scroll the pane to the right to see this). If you double-click on the name of a test in the top

pane, Eclipse will jump to the line where the failure occurred in the editor pane. Figure out the problem in `Fibonacci.java`, fix it, and rerun the JUnit test. Eclipse will automatically rebuild when you make changes, but if you are running Junit from the command line, you must manually rebuild (compile) `Fibonacci.java` before you rerun JUnit. This can be done by rerunning the `ant` command (which also compiles the files) or by following the compiling instructions and then clicking the "Run" button.

Use the information in the Failure Trace box to help you continue debugging `Fibonacci`. Keep a record of what you did to debug `Fibonacci` as you will have to answer questions about your debugging experience in the next problem. After you have fixed all the problems in `Fibonacci`, you should see a bright green bar instead of a red one when you run `FibonacciTest`.

Now look at the JUnit tests that we wrote for `HolaWorld` and `RandomHello` in Problem 5 earlier. They are called `HolaWorldTest` and `RandomHelloTest`, respectively. Ensure that your modified code passes these tests before you turn in your homework.

Problem 6: Answering Questions About the Code

In a newly created text file, answer some questions about the `Fibonacci` class. Most design projects that you will be assigned will require you to submit some sort of response or write-up in addition to your code. Follow these instructions to create a text file, named `hw0/answers/problem6.txt`, with answers to the following questions:

a. Why did Fibonacci fail the **testThrowsIllegalArgumentException** test? What did you have to do to fix it?

b. Why did Fibonacci fail the **testBaseCase** test? What (if anything) did you have to do to fix it?

c. Why did Fibonacci fail the **testInductiveCase** test? What (if anything) did you have to do to fix it?

Problem 7: Getting a Real Taste of Java-Balls and Boxes

Until now, we have only been introducing tools. In this problem, we will delve into a real programming exercise. If you are not familiar with Java, we recommend working through the Optional Problems and Oracle's Learning the Java Language tutorial. (Skip the section on generics for now.) Fragments of Oracle's other tutorials may also be useful, specifically "Getting Started", "Essential Java Classes", and "Collections".

This problem is intended to give you a better sense of what Java programming entails. This problem will likely be somewhat challenging for most of you. Don't be discouraged, we're here to help. And we expect that time spent now will pay off significantly during the rest of the course.

As you work on this problem, record your answers to the various questions in `problem7.txt` in the project's `hw0/answers/` directory.

a. **Warm-Up: Creating a Ball**:

Take a look at Ball.java. A `Ball` is a simple object that has a volume.

- What is wrong with `Ball.java`? Please fix the problems with `Ball.java` and document your work in `problem7.txt`.

We have included a JUnit test called **`BallTest.java`** to help you out. If you are using Eclipse, one of its warnings should help you find at least one of the bugs without referring to the JUnit results.

b. **Using Pre-Defined Data Structures**:

Next, we want to create an class called **`BallContainer`**. As before, skeleton code is provided (see BallContainer.java). A BallContainer is a container for Balls. BallContainer must support the following methods and your task is to fill in the code that will implement all these methods correctly:

1. add(Ball)
2. remove(Ball)
3. getVolume()
4. size()
5. clear()
6. contains(Ball)

The specifications for these methods are found in the javadoc file for **`BallContainer`**.

In BallContainer, we use a **`java.util.Set`** to keep track of the balls. This is a great example of using a predefined Java data-structure to save yourself significant work.

Before implementing each method, read the documentation for `Set`. Some of your methods will be as simple as calling the appropriate predefined methods for **`Set`**. To help you out, we have included a JUnit test called **`BallContainerTest.java`**.

Before you start coding, please take time to think about the following question (which you need to answer in the text file):

There are two obvious approaches for implementing getVolume():

1. Every time `getVolume()` is called, go through all the Balls in the `Set` and add up the volumes. (**Hint:** one solution might use a for-each loop to extract `Ball`s from the `Set`.)
2. Keep track of the total volume of the Balls in `BallContainer` whenever Balls are added and removed. This eliminates the need to perform any computations when `getVolume` is called.

Which approach do you think is the better one? Why?

c. **Implementing a Box**:

In this problem, you will do a little more design and thinking and a little less coding. You will implement the **`Box`** class. A Box is also a container for Balls. The key difference between a `Box` and a `BallContainer` is that a Box has only finite volume. Once a box is full, we cannot put in more Balls. The size (volume) of a Box is defined when the constructor is called:

```
public Box(double volume);
```

`Box` `BallContainer`

Since a `Box` is in many ways similar to a `BallContainer`, we internally keep track of many things in the `Box` with a `BallContainer`, allowing us to reuse code. Many of the methods in `Box` can simply "delegate" to the equivalent in `BallContainer`; for example, removing from a `Box` cannot cause it to exceed its volume limit. This design of having one class contain an object of another class and reusing many of the latter class's methods is called **composition**.

(**Optional Note**: If you are familiar with Java, you may wonder why we did not simply make `Box` extend `BallContainer` via "inheritance"; that is, why did we not make `Box` a subclass of `BallContainer`. We will discuss this much more deeply later in the course, but the key idea is that `Box` is not what we call a true subtype of `BoxContainer` because it is in fact more limited than `BallContainer` (a `Box` can only hold a limited amount); hence, a user who uses a `BallContainer` in his code can not simply substitute that `BallContainer` with a `Box` and assume the same behavior in his program. (The code may cause the Box to fill up, but he did not have this concern when using a `BallContainer`). For this reason, it is not a good idea to make `Box` extend `BallContainer`.)

In addition to the constructor described above, you will need to implement the following new methods in `Box`:

1. add(Ball)
2. getBallsFromSmallest()

The specifications for these methods can be found in the javadoc file for **`Box`**.

A few things to consider before you start writing code:

- You shouldn't need to implement your own sorting algorithm. Instead, take advantage of the Java API (remember: "Know and Use the Libraries").
- Also, you shouldn't need to change your implementation of `BallContainer` or `Ball` for this problem. In particular, you should not implement the `Comparable` interface. If you are tempted to do so, consider using `Comparator` instead. `Comparator` is a companion interface to `Comparable` and is used throughout the Java libraries: check out the sort methods in `java.util.Collections` as an example.
- If you do make any changes to `BallContainer` or `Ball` for this problem, then explicitly document what changes you made and why in `problem7.txt`.
- Be cautious if you plan on using Java's `TreeSet`; remember that `TreeSet` does not store duplicates, and if you provide a `TreeSet` with a `Comparator`, it will use that `Comparator` to determine duplication. See the **`TreeSet`** API documentation for more details.
- Before you start working on `getBallsFromSmallest()`, we recommend strongly that you consider using **`Iterator`**.
- The JUnit test **`BoxTest.java`** should help you out. However, we do not guarantee that the tests we provide will catch all bugs in your program.
- Oh yeah, don't forget to commit your code more than occasionally.

Also, take some time to answer the following questions in your text file:

1. There are many ways to implement `getBallsFromSmallest()`. Briefly describe at least two different ways. Your answers should differ in the implementation of `Box`, not in lower-level implementation (for example, using an insertion sort instead of a selection sort is a lower-level implementation because it does not affect how `Box` is implemented). Hint: think about different places in the `Box` class where you

could add code to achieve the desired functionality.

2. Which of the above ways do you think is the best? Why?

There is no one **correct** answer. Our intent is to help you fight that urge to code up the first thing that comes to mind. Remember: **More thinking, less coding**.

Problem 8: Turning In Your Homework

You will turn in the solutions for the problem sets, including this one, by checking them into your SVN repository. All the relevant files should be included, for example:

- Java source files we provide that you change.
- Java source code you write.
- Text files, drawings, or documentation of your code.

Each homework will indicate exactly what you need to turn in a Section entitled "What to Turn In".

If you add a new file, you must explicitly add it to your SVN repository. This means that you need to tell SVN to "track this file" as part of the repository. In addition, you need to **commit** the file to actually put the contents in the repository. Make sure that, when you're done, your working copy is committed to the repository so that we get and grade your most up-to-date version.

Since files like `RandomHello.java`, `problem6.txt` and `problem7.txt` are not in your SVN repository, you must add them. You also want to commit the changes you made to `HolaWorld.java`, `Fibonacci.java`,`Ball.java`, `BallContainer.java`, and `Box.java`.

After completing the above steps, all your code and materials to turn in should be in your SVN repository. The final step to turn in is to *validate* your solutions. The validate script performs general sanity checks on your solutions. In particular, it checks out a fresh copy of the module from your SVN repository into a temporary directory, checks that it has the required files, compiles the Java files, and tests your code against our suite of public tests (see here for more information about this testing framework, and about the test classes `hw0.test.SpecificationTests` and `hw0.test.ImplementationTests`, which you do NOT have to modify for this homework).

To validate, execute the **validate** target in your Ant buildfile (`build.xml`). This does not work well with Eclipse's integrated Ant support, so even if you are using Eclipse as your development environment, you should validate on the command-line, by running the following on `attu`:

```
cd ~/workspace331/cse331/src/hw0/
ant validate
```

You can do this via SSH from any machine. (Even if you are working on an Allen Center Linux machine you still need to SSH into attu.) If you are working at home you will first need to check out a copy of your code on `attu` using the command line. Note that if you check out your working copy in the location suggested there, the path to your project (as listed in the directions above and in the ouput below) will not include the `workspace331/` directory.

If validation was successful, you should see output that looks something like:

```
Buildfile: /homes/iws/username/workspace331/cse331/src/hw0/build.xml

validate:

     [echo] Validate checks out a fresh copy of the hw, checks for the
     [echo]       presence of required files, and runs all your tests to make sure
     [echo]       they pass.  This target is meant to be called from attu; don't
     [echo]       trust it to make sure everything works unless you are on attu.
     [echo]       Note: the test reports will be generated under the scratch
     [echo]       directory the validate target creates.
     [echo]
   [delete] Deleting directory /homes/iws/username/workspace331/cse331/scratch
    [mkdir] Created dir: /homes/iws/username/workspace331/cse331/scratch
     [echo] /projects/instr/10sp/cse331/username/REPOS
     [exec] A    cse331
     [exec] A    cse331/.classpath
     [exec] A    cse331/.project

     ...

     [exec] Buildfile: /homes/iws/username/workspace331/cse331/scratch/cse331/src/hw0/build.xml
     [exec]
     [exec] cleancopy:
     [exec]      [echo] hw directory: /homes/iws/username/workspace331/cse331/scratch/cse331/src/hw0
     [exec]

     ...

     [exec] BUILD SUCCESSFUL
     [exec] Total time: 2 seconds
     [exec] Buildfile: /homes/iws/username/workspace331/cse331/scratch/cse331/src/hw0/build.xml
     [exec]
     [exec] build:
     [exec]
     [exec] single.build:
     [exec]      [echo] This hw is independent; building only this one.
     [exec]     [javac] Compiling 23 source files to /homes/iws/username/workspace331/cse331/scratch/cse331/bin
     [exec]
     [exec] multi.build:
     [exec]
     [exec] test.impl:
     [exec]      [mkdir] Created dir: /homes/iws/username/workspace331/cse331/scratch/cse331/src/hw0/test/reports
     [exec]      [junit] Running hw0.test.ImplementationTests
     [exec]      [junit] Tests run: 0, Failures: 0, Errors: 0, Time elapsed: 0.004 sec
     [exec]
     [exec] test.spec:
     [exec]      [junit] Running hw0.test.SpecificationTests
     [exec]      [junit] Tests run: 24, Failures: 0, Errors: 0, Time elapsed: 0.181 sec
     [exec]
     [exec] test:
     [exec]      [echo] Records of this testing can be found in
/homes/iws/username/workspace331/cse331/scratch/cse331/src/hw0/test/reports/
     [exec]
     [exec] test.strict:
     [exec]
     [exec] BUILD SUCCESSFUL
     [exec] Total time: 4 seconds
     [delete] Deleting directory /homes/iws/username/workspace331/cse331/scratch

BUILD SUCCESSFUL
Total time: 9 seconds
```

If there is an error, the validate script should provide some information about what is wrong:

```
Buildfile: /homes/iws/username/workspace331/cse331/src/hw0/build.xml

validate:
     [echo] Validate checks out a fresh copy of the hw, checks for the
     [echo]       presence of required files, and runs all your tests to make sure
     [echo]       they pass.  This target is meant to be called from attu; don't
     [echo]       trust it to make sure everything works unless you are on attu.
     [echo]       Note: the test reports will be generated under the scratch
     [echo]       directory the validate target creates.
     [echo]
...
     [exec] cleancopy.check:
     [exec]      [echo] Found required file BallContainer.java
     [exec]
     [exec] cleancopy.check:
     [exec]      [echo] Found required file Box.java
     [exec]
     [exec] cleancopy.check:
     [exec]      [echo] Found required file answers/problem6.txt
     [exec]
     [exec] cleancopy.check:
     [exec]
     [exec] BUILD FAILED
     [exec] /homes/iws/username/workspace331/cse331/scratch/cse331/src/common.xml:81: The following error occurred while
executing this line:
     [exec]
     /homes/iws/username/workspace331/cse331/scratch/cse331/src/common.xml:96:
     Could not find required file: answers/problem7.txt
     [exec]
     [exec] Total time: 2 seconds

BUILD FAILED
/homes/iws/username/workspace331/cse331/src/common.xml:129: exec returned: 1
```

This error would indicate that a required file, `answers/problem7.txt` is missing. Make sure you've committed this file to SVN.

If validate failed because the test suite failed, you can view a summary of the JUnit

failures in your `YourWorkspaceDirectory/cse331/scratch/src/hw0/test/reports/` directory.

Validate your homework as many times as necessary until there are no errors. You have now successfully turned in your first CSE 331 homework. We encourage you to give the Optional Tutorial Problems a try.

What to Turn In

Your TA should be able to find the following when he or she checks your `src` directory of SVN:

- `hw0/HolaWorld.java` that works as described in Problem 5 with no compilation errors.
- `hw0/RandomHello.java` that prints out one of five random messages when its `main` method is executed.
- `hw0/Fibonacci.java` that passes the three tests in `hw0/test/FibonacciTest.java`. (Note that you should **NOT** edit `hw0/test/FibonacciTest.java` to accomplish this task.)
- `hw0/Ball.java`, `hw0/BallContainer.java` and `hw0/Box.java` that pass their respective JUnit tests. (Again, you should not modify the JUnit tests, though you are most welcome to read the source code to understand what they test for).
- `hw0/answers/problem6.txt` and `hw0/answers/problem7.txt` containing answers to the questions in Problems 6 and 7.

Please include your first and last name in every text file you turn in (i.e., files ending with `.txt`).

Optional Tutorial Problems

If you have not had much experience with Java, we recommend that you do these extra Optional Problems (source code for these problems can be found in the `hw0.optional` package that came with your `hw0` checkout). If you do not know Java, CSE 331 will require you to learn it quite quickly. The optional problems, although not required, will help you become more comfortable with Java.

Optional Debugger Tutorial

In this part, you will learn to Eclipse's built-in debugger. As the name suggests, a debugger helps you debug your program by allowing you to "watch" your program as it executes, one line at a time, and inspect the state of variables along the way. In the past, you may have used `System.out.println()` to probe the state of a variable or find out whether a particular line is reached. In that sense, debuggers are like an ultra-super-powerful `System.out.println()`.

One of the most useful features of a debugger is the ability to set *breakpoints* in your program. When you run your program through the debugger, it will pause when it reaches a line with a breakpoint. You can inspect the current state of variables, then continue running your program normally or step through one line at a time to watch the variables change.

- Open Adder.java. This simple program is supposed to print the sum of two user-provided integers. Try running it a few times (or reading the source code) and you'll

see that it doesn't behave as expected.
- Double-click in the left-hand margin of the class next to the line `return x - y;` A blue circle should appear, indicating a breakpoint. (Double-clicking again removes the breakpoint.)
- Run the program in debug mode by using F11, `Run >> Debug`, or the green bug icon in the toolbar. As before, enter two ints (say, 3 and 4) in the console when prompted. When your program hits the breakpoint, Eclipse asks if you want to open the Debug perspective. Choose yes.
- The debug perspective looks overwhelming at first, but don't worry! In the top-right panel, select the "Variables" tab and you'll see the names and values of all variables in the current context. The top-left panel ("Debug") shows where the program is currently paused, where the current method was called, and so on. (This is called the stack trace). Double-click on a method name to see the corresponding line in the panel below, where you can view and edit your source code. Finally, the bottom-left panel shows the console window.

  What are all the names and values listed in the "Variables" panel? What does the "Debug" panel list as the current method and line number? (Write down the text that was highlighted when the Debug perspective first opened.)

- Immediately above the "Debug" panel are several buttons for running your program. Mouse over each one for a description. "Resume" (green arrow) causes your program to continue executing normally until it finishes or hits another breakpoint. If you want to monitor what happens shortly after the breakpoint, use "Step Into" and "Step Over" (yellow arrows). "Step Over" executes the current line and pauses on the next line. "Step Into" enters any method called on the current line so you can execute that method line-by-line. (To finish the current method and jump back to the caller, use "Step Out.") Hit "Step Over" once to execute the `return` statement and exit `computeSum`. Hit "Step Over" again to progress to the next line.

  What are all the names and values listed in the "Variables" panel after EACH of the two step overs?

- Hit "Resume" to allow the program to finish executing. Return to the default Java perspective by clicking "Java" in the top-right corner or by going to `Windows >> Open Perspective >> Other...` and selecting Java (default).

Hints

We strongly encourage you to use the CSE 331 Online Forum.

Errata

None yet.

Q & A

This section will list clarifications and answers to common questions about homeworks. We'll try to keep it as up-to-date as possible, so this should be the first place to look (after carefully rereading the homework handout and the specifications) when you have a

problem.