CSCI 2600 Principles of Software

Homework 0: Setup and Java Introduction

Due: Friday, Jun. 5 @ 11:59 pm

The purpose of this assignment is to help you set up your development environment, get acquainted with Java, and introduce you to tools we will be using throughout the rest of this class.

If you would like to get more practice with Java, then we recommend trying Oracle's Java tutorials we mention under Problem 4. Try to complete Homework 0 first, so you can use the tools we describe here when doing the examples in Oracle's tutorial.

If you are having trouble with this assignment, get in touch with us immediately so we can get you back on track.

Getting Started

In what follows, we assume that you have Eclipse and Git properly installed, and you have checked out project csci2600-hw0. If you encountered problems, please contact the TAs or the instructors immediately.

Throughout the course you will receive starter code and submit your assignments through Git. Git is a version control system that allows software engineers to backup, manage, and collaborate on large software projects.

The instructions in the Setup handout outline basic Git commands, Eclipse, and JUnit. Throughout this homework we explain Git commands, Eclipse, and JUnit in context.

Problem 1: Your first Java class — RandomHello

Create your 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`. (Assuming your repository is checked into `csci-2600/hw0`, the file name would be *`csci-2600/hw0`*`/src/main/java/hw0/RandomHello.java`.) To create a new Java class file, go to Package Explorer and select package `hw0` under the `src/main/java`, then select **File -> New -> Class**. Specify the enclosing package `hw0` and class name `RandomHello`.

Java requires every runnable class to contain a `main` method whose signature is `public static void main(String[] args)`. 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. Add a public method called getGreeting() as shown below.

`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*, 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 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, le 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 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. To run select `RandomHello.java` in Package Explorer and then choose **Run -> Run** from the main menu, or right-click on `RandomHello.java` in Package Explorer, then select **Run As -> Java Application**.

Next, add your new file to version control, commit it into your local repository, and push to the repository on the server. Follow the Setup handout for the relevant Git commands.

For Problems 2-4, **DO NOT** edit files with test cases: FibonacciTest.java, BallTest.java, BallContainerTest.java, and BoxTest.java. If you do, you may make all your test cases run successfully on your local machine. However, when you submit your assignment to Submitty, we will be using our own original copies of all test cases for autograding, not the ones you might have committed to your repository. It might lead to your code failing some or all the test cases on Submitty despite the fact that the entire test suite ran successfully on your local machine.

Problem 2: 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. (Later homework assignments will look more deeply into unit testing.)

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

Now run `FibonacciTest`. Right-click on `FibonacciTest.java`, then select **Run As -> JUnit test**. If you don't see **Run As -> JUnit test** when you right-click, right-click on the project name in the Package Explorer and select **Properties**. Select **Java Build Path**, and click the **Add Library** button. Select **Junit** and click **Next**. Select **Junit 4** (not Junit 5) and click **Finish**. Click the **Apply and Close** button.

A window or panel with a menacing red bar will appear, indicating that some of the tests in `FibonacciTest` did not complete successfully. 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.

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 an `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.

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`.

For Problem 3 and Problem 4 submit your answers as .PDF files named `hw0_problem3.pdf` and `hw0_problem4.pdf` in the `answers/` directory of your repository.
You MUST type up your answers. Handwritten solutions will not be accepted or graded, even if they are scanned into a PDF file.
We recommend using LaTeX. If you have never used LaTeX, take a look at this tutorial.

Problem 3: Answering Questions About the Code

In a newly created file (e.g., if you are using LaTeX it would be named `hw0_problem3.tex`) answer some questions about the `Fibonacci` class. Make sure that this file is placed in the `answers/` directory of your project. This will ensure that the PDF file which you generate (e.g., `answers/hw0_problem3.pdf`) is also placed in the `answers/` directory. Most programming homework assignments that you will be given will require you to submit some sort of a response or write-up in addition to your code. Write your answers to the following questions:

1. Why did Fibonacci fail the **testThrowsIllegalArgumentException** test? What did you have to do to fix it?
2. Why did Fibonacci fail the **testBaseCase** test? What (if anything) did you have to do to fix it?
3. Why did Fibonacci fail the **testInductiveCase** test? What (if anything) did you have to do to fix it?

Generate a PDF file from the file with your answers. E.g., if you are using LaTeX, run `pdflatex` command or use your LaTeX editor to create `answers/hw0_problem3.pdf`. Make sure to add this PDF file to Git index as described in Problem 5. **You MUST ensure that your repository contains required PDF files. We will not be able to grade your source answer files (like `.tex` or `.docx`).** PDF files that you submit must be real PDF documents. Merely renaming your `.tex` or `.docx` files so that they have a PDF extension will not make them actual PDF documents, and we will be unable to grade them.

Problem 4: 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 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 can be somewhat challenging. 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 `hw0_problem4.pdf` in the project's `answers` folder.

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

   Take a look at `src/main/java/hw0/Ball.java`. A `Ball` is a simple object that has the volume and the color.

   - What is wrong with `Ball.java`? Please fix the problems with `Ball.java` and document your work in `hw0_problem4.pdf`.

   We have included a JUnit test called `src/test/java/hw0/BallTest.java` to help you out. In Eclipse, one of its warnings should help you find at least one of the bugs without referring to the JUnit results. Warnings are indicated by a small yellow marker to the left of the line number. Moving the mouse over the marker will show you the warning. Clicking on the marker will give you hints about possible ways to modify the code to resolve the warning.

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

   Next, we want to create a 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:

   - `add(Ball)`
   - `remove(Ball)`

- `getVolume()`
- `size()`
- `differentColors()`
- `areSameColor()`
- `clear()`
- `contains(Ball)`

The specifications for these methods are found in the code of `BallContainer.java`.

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 the Set. To help you out, we included a JUnit test called `src/test/java/hw0/BallContainerTest.java`.

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

There are two obvious approaches to implementing `getVolume()`:

- Every time `getVolume()` is called, go through all the Balls in the `Set` and add up the volumes. **Hint:** one solution might be to use a for-each loop to extract `Ball`s from the `Set`.
- 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? Include your answer in `hw0_problem4.pdf`.

3. **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);
```

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 `BallContainer` 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 their code can not simply substitute a `BallContainer` with a `Box` and assume the same behavior in the program. The code may cause the Box to fill up, but they 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`:

- `add(Ball)`
- `getBallsFromSmallest()`

The specifications for these methods can be found in the code of `Box`.

A few things to consider before you start writing code:

- You should not 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 `hw0_problem4.pdf`.
- Be cautious if you plan on using Java `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 strongly recommend that you consider using Iterator.
- The JUnit test `src/test/java/hw0/BoxTest.java` should help you out. However, we do not guarantee that the tests we provide will catch all bugs in your program.
- Don't forget to commit and push your code more than occasionally.

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

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

There is no single **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 5: Turning In Your Homework

Each homework will indicate exactly what you need to turn in a Section entitled **What to Turn In** (see below).

Make sure that you have added all new files to version control (e.g., `RandomHello.java`, `answers/hw0_problem3.pdf`, and `answers/hw0_problem4.pdf`) by right clicking on the file name in the Eclipse Project Explorer and selecting **Team/Add to Index**. In Eclipse, right-click on project `csci2600-hw0`, then select **Team -> Commit** to commit all

changes. Don't forget to push into the remote repository on the server: right-click on project `csci2600-hw0`, then select **Team -> Push to Upstream**.

After completing these steps, all your code and materials to turn in should be in your Git repository on the server. Proceed to the Submitty to complete the submission of the assignment!

IMPORTANT: Make sure that you have the correct folder structure. If you break the structure, compilation on the Submitty server will fail resulting in a grade of 0. At this point, you must have project `csci2600-hw0` with subfolders `src`, `answers`, and `docs`. Folder `src` must have subfolders `main` and `test` each of which, in turn, must have subfolders `java` and `resources`. Folders named `java` must have subfolders `hw0`. These show as `hw0` subfolders of `src/test/java` and `src/main/java` in Package Explorer. Java classes (e.g., `Ball.java`) must be in `src/main/java/hw0`, your PDF files (`hw0_problem3.pdf` and `hw0_problem4.pdf`) must be in `answers` and all JUnit test classes (e.g., `BallTest.java`) must be in `src/test/java/hw0`.

You must click the Grade My Repository button for you answers to be graded. If you do not, they will not be graded and you will receive a zero for the homework.

For this and all other homework assignments, Submitty is configured to allow 20 grading attempts without penalty. For all submissions starting from the 21st, there will be a small penalty charged for each additional submission.

What to Turn In

We should be able to find the following folders and files in your `csci2600-hw0` folder:

- `src/main/java/hw0/RandomHello.java` that prints out one of five random messages when its `main` method is executed.
- `src/main/java/hw0/Fibonacci.java` that passes all four tests in `src/test/java/hw0/FibonacciTest.java`. Note that you should **NOT** edit `src/test/java/hw0/FibonacciTest.java` to accomplish this task.
- `src/main/java/hw0/Ball.java`, `src/main/java/hw0/BallContainer.java`, and `src/main/java/hw0/Box.java` that pass their respective JUnit tests. Again, you should not modify JUnit tests, though you are most welcome to read the source code to understand what they test for.
- `answers/hw0_problem3.pdf` and `answers/hw0_problem4.pdf` containing answers to the questions in Problems 3 and 4.

Grade Breakdown

This homework is worth 50 points. Submitty server runs the provided JUnit tests plus a few additional tests. Test and debug your code in Eclipse before committing and submitting to Submitty! If your code passes all tests in Eclipse, then chances are it will pass them on the Submitty server, too.

- `RandomHello`: 2 pts
- Compilation: 4 pts (autograded)
- `FibonacciTest` JUnit tests: 4 pts (each test 1 pt, autograded)
- `BallTest`: 3 pts (each test 1 pt, autograded)
- `BallContainerTest`: 10 pts (each test 1 pt, autograded)

- `BoxTest`: 8 pts (each test 1 pt, autograded)
- Instructor `Ball` tests: 3 pts (each test 3 pts, autograded)
- Instructor `BallContainer` tests: 3 pts (each test 3 pts, autograded)
- Instructor `Box` tests: 3 pts (each test 3 pts, autograded)
- Answers to Problem 3 questions: 3 pts
- Answers to Problem 4 questions: 7 pts

Parts of this homework are derived from University of Washington's Software Design and Implementation course.