CSCI 455

PA

1

CS 455 Programming Assignment 1

Fall 2019 [Bono]
Due date: Wednesday, Sep. 18, 11:59pm

Introduction

In this assignment you will write a graphics-based program to simulate someone tossing a pair of coins some number of times, and display the results. So for example the user may request 10 trials. For each trial two coins are tossed. The program reports in bar graph form how many times the result was two heads, how many times it was two tails, and how many times it was one of each.

This assignment will give you practice with creating and implementing classes, using loops, using the java library for random number generation, doing console-based IO, and drawing to a graphics window. Also you'll get practice in general program development.

Resources

- Horstmann, Section 2.9, 2.10, 3.8, How-to 3.2 Programs that draw stuff
- Horstmann, Chapter 3, Implementing Classes
- Horstmann, Section 4.3.1, Reading input
- Horstmann, Section 6.9.1, Random numbers and Simulations

The programming environment for this assignment

In the normal Vocareum configuration, you have a Linux terminal, but no way to run a program with a graphical user interface (GUI). For this assignment we are using a different Vocareum configuration that will allow you to open multiple windows, including a separate one to run your GUI program. With this configuration, when you start up Vocareum for this assignment, it will not start up a terminal in the workbench window (i.e., the usual one you use), but you use a virtual Linux desktop instead.

How to start up a virtual Linux Desktop in Vocareum
The way you get to a virtual linux desktop in this assignment is to go to a menu that's on the upper right of the workbench window: choose Actions--> Applications --> Desktop.

That will open a linux desktop in another tab in your browser. If it starts with a pop-up dialog, choose "Use default configuration." There are few ways to open a terminal window in this desktop. It may show an icon at the bottom of the screen that you can click to start up a terminal window. If that's not an option, you can right click anywhere on the desktop, and choose "Open Terminal Here".

Warning: depending on how you started up terminal, it might not start out in your home directory (i.e., "work"), but rather starts in the root directory ("/") or somewhere else. So the first thing you should do is

```
cd
```

to get into your home directory. (One way to check if you are in your home directory is you will see the ~ (tilde) right before the $ in the shell prompt.)

Your home directory will be populated with the starter files we are providing you. Part of what we provided is source code for a complete sample Java GUI program there, so you can try out compiling and running such a program in this environment before you write code for your own program. Compile and run this program:

```
javac CarViewer.java
java CarViewer
```

More about this car example (from Section 3.8 of the textbook) later.

You can switch between these two tabs in your browser to switch between editing (normal Vocareum window), and compiling and running (Linux desktop). To make it easier to see your compile errors at the same time as you view your source code, you can put the Vocareum tab in a different browser window altogether.

Another option with the desktop is to use one of the other editors available within the desktop itself. I saw `emacs` and `vim` (Rt-click on desktop-->Applications-->Accessories). I'm not sure how fast these work on this platform, so if you end up using one of these, let me know how it goes. (I only opened emacs there briefly once; it started up pretty fast, so that's a good sign.) Both emacs and vim are a little different than other editors you are used to, so you probably would want to take a look at an online tutorial on the web before using them.

One caveat: this Linux desktop is a newer feature of Vocareum, and it's a bit of a moving target. If it gets wedged you might have to restart the desktop. You can disconnect from it by closing the tab, or in the main Vocareum window (upper right) do: Actions-->Applications-->Stop App. Then you can restart the same way you did earlier.

Using another IDE for this assignment

If you don't want to use Vocareum and its Linux desktop as your development environment, you can use another IDE running locally on your own machine.

If you choose go this other route, you would do the following:

1. Install and try out another IDE. There are a few tutorials in the Java section of the Documentation page to help you get started with the Eclipse IDE.
2. Download the starter files for the assignment into a folder on your laptop (e.g., call it `pa1`). The easiest way to do this is, from the Vocareum PA1 assignment workbench do: Actions-->Download starter code (menu on the upper right).
3. Get your program working, tested, and completely documented on your laptop IDE, and when you are ready to turn it in . . .
4. Upload your complete program into Vocareum. In the Vocareum workbench, click Upload (button in the upper-left area), and you will get a file browser that will allow you to select multiple files to upload. You will only need the source code files (and README), no `.class` or project files. *Do not wait until the final due date/time is imminent before uploading and testing it on Vocareum.*
5. Compile and completely test your program again on Vocareum. You want to make sure what you wrote will run in the Vocareum environment, which is the grading environment. This also helps you to make sure that you are going to be submitting the correct version of the assignment.
6. Submit your assignment on Vocareum. Please read the details about submitting this in the section on it at the end of this document.

The assignment files

The files in **bold** below are ones you create and/or modify and submit. The ones *not* in bold are files you will use, but that you should not modify. The ones with a * to the left are starter files we provided.

- * `CarViewer.java`, `CarComponent.java`, and `Car.java`. The code for the example in Section 3.8 of the textbook. For more about why these are in the starter files, see the section on class design.
- * **`CoinTossSimulator.java`** Your `CoinTossSimulator` class. The public interface is provided. You will be completing the implementation and a test driver for it.
- * **`Bar.java`** A `Bar` class. The public interface is provided. You will be completing the implementation of it.
- **`CoinSimViewer.java`** Your `CoinSimViewer` class. You create this file and class.
- **`CoinSimComponent.java`** Your `CoinSimComponent` class. You create this file and class.
- **`CoinTossSimulatorTester.java`** Your unit test program (a.k.a., test driver) for your `CoinTossSimulator` class. You create this file and class.
- * `CoinSimViewer.list` A list of the .java files for compiling the `CoinSimViewer` program. For more information about this, see the subsection on compiling multi-file programs.
- * **`README`** for more about what goes in this file, see the section on README file. Before you start the assignment please read the following statement which you will be "signing" in the README:

> "I certify that the work submitted for this assignment does not violate USC's student conduct code. In particular, the work is my own, not a collaboration, and does not involve code created by other people, with the exception of the resources explicitly mentioned in the CS 455 Course Syllabus. And I did not share my solution or parts of it with other students in the course."

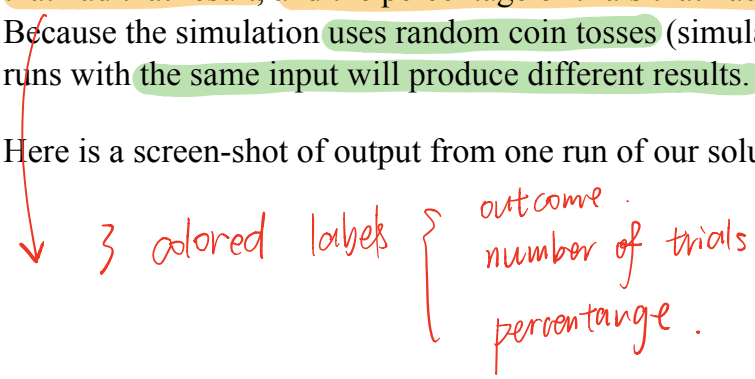
For more information about the classes mentioned above see the section on class design.

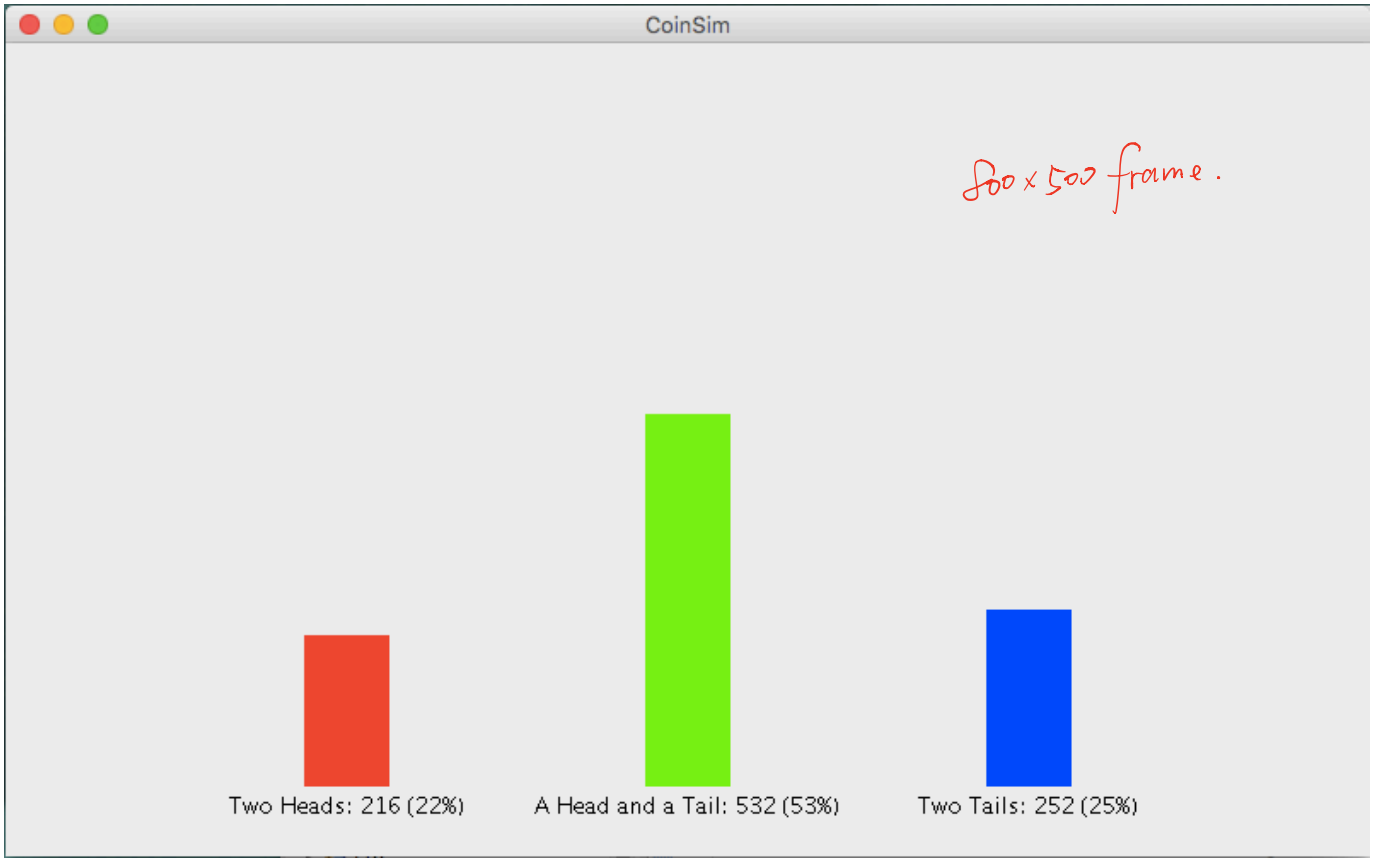
The assignment

Initially your program will prompt for the number of trials to simulate (a trial is two coin tosses) on the console (i.e., terminal window), error checking that a positive value is entered. (More details about error checking here.) This part of the program will be console-based, to keep things simpler.

Then it will run the simulation and display a 500 tall by 800 wide pixel window with the results of that simulation. The results will consist of three labeled bars, each a different color, to show how many trials had the specified outcome. The label will show what the outcome was (e.g., Two Heads), the number of trials that had that result, and the percentage of trials that had that result (rounded to the nearest one percent). Because the simulation uses random coin tosses (simulated using a random-number generator) subsequent runs with the same input will produce different results.

Here is a screen-shot of output from one run of our solution to this assignment, where we do 1000 trials:

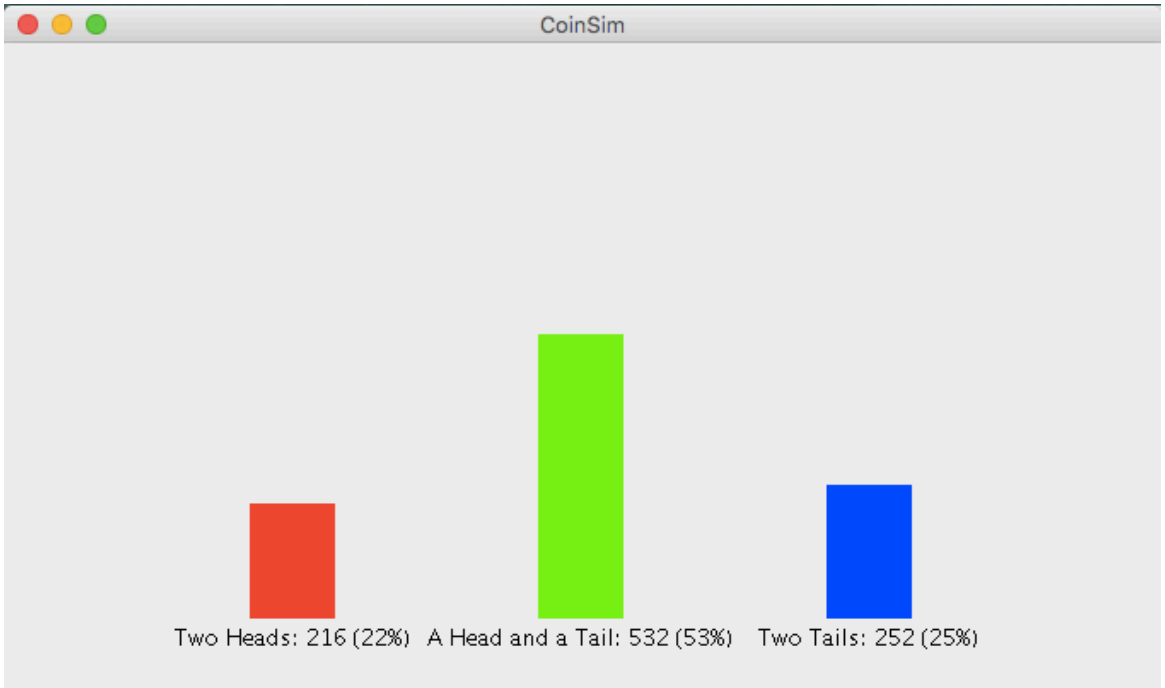




Remember, your output will not be identical to this because of the random nature of the results.

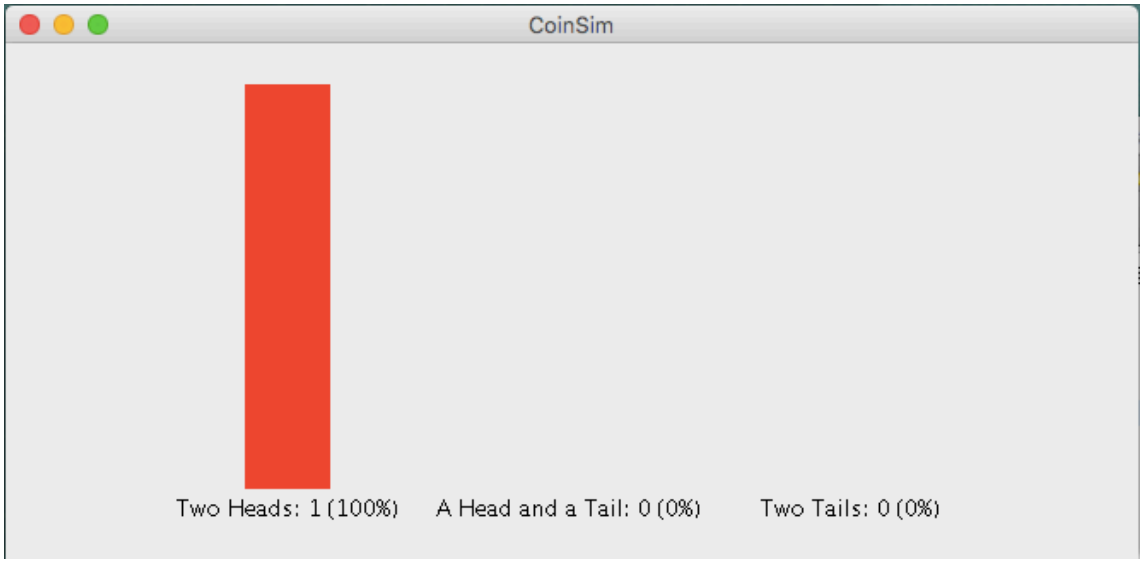
Note the placement of each of the bars evenly across the window. In addition, the height of each bar is given so that 100% would fill up most of the height of the window (but not run into the top of it). Thus the 48% of trials that resulted in a head and a tail in the example above fills up roughly half of the height of the window.

Also, your bar graph should get resized appropriately if the window gets resized. As mentioned in the textbook, every time a window gets resized or iconified and de-iconified `paintComponent` gets called again. Here's a later screen-shot created during same run shown above, but after the window had been resized:



Note that resizing the window does not change the results of the simulation.

Here's an example illustrating what the display looks like when all of the trials have the same result (i.e., the bar doesn't run off the top of the screen), forced here by only doing one trial (screen-shot shows the window after it was resized smaller):



More about the graphics library methods necessary to get these results in the section on Graphics programming.

There are a few other requirements for the assignment discussed in the following sections. To summarize here, the other requirements are:

- you must create and use the classes mentioned in the section on Class design.
- you must create a working unit test program for your `CoinTossSimulator` class, describe further in the section on Testing the `CoinTossSimulator` class.
- you must edit and submit README file discussed in the section about that. Do not wait until you are about to submit the program to answer the README questions, because they involve further tests of your program, and you may discover a bug during that testing.
- your program will also be evaluated on style and documentation. More about this in the section on grading criteria.

More details of the error-checking

As mentioned in the earlier section, when your program prompts for the number of trials, you will error check that a positive value is entered. More specifically, we mean that on an invalid number of trials the program will print out a informative error message and then prompt and read again until the user enters a valid value. Example (user input shown in italics):

```
Enter number of trials: -5
ERROR: Number entered must be greater than 0.
Enter number of trials: 0
ERROR: Number entered must be greater than 0.
Enter number of trials: 100
```

Your program does not have to handle non-numeric input. (We will not test it on that case.)

Class design

To help you make your program object-oriented, we are giving you the general class design for this program.

The Car Example
Our program follows the conventions of graphical classes used in the textbook (see Resources, near the beginning of this document, for relevant textbook readings). In particular, this general design follows the car example in Section 3.8 of the textbook that has a viewer, a component, and a graphical object that can get instantiated multiple times and drawn in different locations on the screen (in that one the object class is a `Car`, here it's the `Bar`).

In the starter files we provided you with the source code for that example. In addition to examining the general structure of the code, you can use it to test out running a GUI program in the Vocareum virtual Linux Desktop before developing your own code. When you run it there, you can also see how the display changes when you change the size of the window in which the CarViewer application is running and the corresponding code that gets that to happen (our display for our program will also change when the window is resized). Furthermore, you should use the code in CarViewer as a starting point for your CoinSimViewer class for this assignment.

You are required to use the following classes (ones in bold are ones you will be creating yourself or implementing):

- **`CoinSimViewer`**. Contains the `main` method. Prompts for the number of trials, and creates the `JFrame` containing the `CoinSimComponent`. Besides `CoinSimComponent`, this class does not depend on any of the other classes mentioned here (e.g., if one of those other classes changed, `CoinSimViewer` would not have to change.) The later section on communicating information between objects will be useful when developing this and the next class listed.

- **`CoinSimComponent`**. Extends `JComponent`. Constructor initializes any necessary data and runs the simulation. Overrides `paintComponent` to draw the bar graph, using `Bar` objects for each bar in the graph. This class uses the `CoinTossSimulator` and `Bar` class.

- **`CoinTossSimulator`**. This is the class responsible for doing the simulation. It has no graphical output, and in fact, does no I/O to the console either. It has a `run` method to run a simulation of tossing a pair of coins for some number of trials. It has accessors to get the results of the simulation. Subsequent calls to `run` add trials to the current simulation. To reset the `CoinTossSimulator` to start a new simulation, call the `reset` method. For examples of how it operates, see the sample output of the test program from the section on testing this class.

  We are giving you the exact interface to use for this class. By *interface*, we mean what clients need to know about the class to use it, i.e., the class comment, the method headers and associated method comments. Do not change the interface when you incorporate it into your own program. For all assignments in this class, when we say that, we mean no changing the provided method headers, no adding public methods, no removing public methods. We will be using our own test programs with such classes, and if you change the public interface your code might not even compile with our test programs.

`CoinTossSimulator` does not depend on any of the other of our classes or the graphics library. It does use `Random` (described further below). The skeleton code for `CoinTossSimulator` is in `CoinTossSimulator.java`.

- **`Bar`**. For drawing a bar in a bar graph. You specify the location, dimensions and color of the bar in the constructor. The only other method is `draw` which draws the bar given. We are giving you the exact interface for this class. (See the previous bullet point for more details about what we mean by that.) `Bar` does not depend on any of the other of our classes, but does use the graphics library. The skeleton code for `Bar` is in `Bar.java`.

- **`CoinTossSimulatorTester`**. A program to test your `CoinTossSimulator` class independently from its use in the `CoinSimViewer` program. It will have its own `main` method. This class is described in more detail in the section on testing the `CoinTossSimulator` class.

- `java.util.Random`. The java random number generator. See section 6.9 of the textbook for examples of its use. One thing to note about `Random`: it's a class that through a sequence of method calls generates a sequence of values that depend on the internal state of the object (in this way it is similar to `Scanner`). Students often want to create a new `Random` object every time they want a new random number. Don't do that. Instead, normally you create one `Random` object in your program, and then whenever you want a new random number you make another call to `nextInt` on that same object. (If you create a `Random` object every time you are not generating a pseudo-random sequence, although it will appear that you are.)

  To reflect what would be going on in the real-world version of these trials, your program must generate one random number to simulate a single coin toss.

Note: this list doesn't include all the java library classes that will be used in the program; for example `CoinSimComponent` will need `java.awt.Graphics`.

Incremental development

Any program of non-trivial size will be developed faster, with fewer bugs, using the technique of incremental development, which means developing, and testing, pieces of the program incrementally. The incremental aspect is that your program may gradually grow until it includes the complete functionality. (Other people use different names for the same thing. Sometimes it's called building subsets.)

A desirable feature of individual classes is that they are as independent as possible from a program that uses them. Some classes, such as `String`, or `ArrayList` (which we will see soon) are general-purpose and can be used in many different programs. Other classes are more special-purpose, such as `CoinTossSimulator`, but still are modules that can be separated from a particular program that uses them. We can test such a module using a unit-test, which is a program specially designed to test the module.

We often unit-test one (or more) classes, and then once we are convinced that unit is working correctly, we can integrate that class with other code that uses it. If this larger code base is now buggy, we can feel fairly certain that the bug is in the new code we added, since we already tested the first class. So any time we find bugs, it's in a small program: much easier than locating bugs in large programs.

Similarly, if we make *later* changes or enhancements to our non-tester client program our code will be more

robust in the face of these changes because, in our unit-test, we tested the module in ways not specific to how it was used in the original client program. (As you have experienced as a user, software is always getting changed over time, e.g., the latest version of Windows is in the double-digits.) For example, in this assignment, there are methods and method functionality of `CoinTossSimulator` that are not used by the program that draws the bar graph, but you would never be sure whether they worked if you didn't test them.

For this assignment, the final product will not be a very large program, but we want to get you in the practice of using incremental development, so you will still be successful when you are trying to develop and debug much larger programs. Even in this program there are at least two distinct issues to deal with: (1) figuring out how to use the random-number generator to do a coin-toss simulation and (2) figuring out how to do the graphics to draw the results of the simulation. It will be much easier you we can deal with these issues one at a time, so you can isolate bugs related to each one more easily. For this assignment you are required to write a console-based Tester class to test our CoinTossSimulator class. This test program is described in more detail in the next section.

Similarly, you could test your Bar class apart from its use in this particular bar graph by creating several bars with hard-coded data or data from the keyboard using a Scanner. We won't require you to submit such a BarTester program for this assignment, however.

Testing the `CoinTossSimulator` class

You are actually going to submit two programs for this assignment, both of which use your `CoinTossSimulator` class. One is `CoinSimViewer`, described earlier, that has a graphical display. The other is a console-based program, `CoinTossSimulatorTester`, expressly written to thoroughly test your `CoinTossSimulator` class, without including the drawing functionality of the `CoinSimViewer` program. The rationale for unit tests was discussed in the previous section. First, here's more information about compiling Java code:

How to compile and run multi-file Java programs on the command line Often you can just list the file that contains `main` in the compile command and `javac` figures out what other classes are used in that program and compiles those as well. However, sometimes the Java compiler gets confused when you only have modified some of the source files since the original compile. For *running* a program that uses multiple class files, the only class name you give as the argument to the `java` virtual machine is the one containing `main`.

When you are compiling and running your test program you should be able to do it as follows:

```
javac CoinTossSimulator*.java
java CoinTossSimulatorTester
```

compile all other classes that will use CoinToss Simulator.

The wild-card in the compile command will match the two files `CoinTossSimulatorTester.java` and `CoinTossSimulator.java`.

*: wild-card

#: pound sign

For the larger program we are doing for this assignment (for that one `main` is in `CoinSimViewer.java`), you can either list all of the files it uses on the command line; but the following is a convenient shorthand:

```
javac @CoinSimViewer.list
java CoinSimViewer
```

The `CoinSimViewer.list` file just consists of the list of files to compile for the program. The `@` on the command line tells java to look in the file that follows it find out what java files to compile. An alternate is to use `*.java` instead.

compile all java documents under the current directory.

As mentioned in the previous section a test program like `CoinTossSimulatorTester` is called a unit test; we have discussed such unit tests in lecture, and they are also discussed in Section 3.4 of the textbook. One goal of this test program is for you to test the full functionality of the CoinTossSimulator class. Its use in the CoinSimViewer program does not test the full functionality. In particular, with the CoinTossSimulator you can do multiple calls to the `run` method to add more trials to the ones already made. Only if you call `reset` does it reset the simulation back to its beginning state (i.e., no trials done yet).

So, what should you put in your `CoinTossSimulatorTester`? This will be a console-based program -- i.e., no GUI. It will be a *non-interactive program* (i.e., fixed data, nothing read in from the user), that tests every method multiple times, printing informative output to the console with the results of each operation. Make sure you also test creating multiple instances of the class.

?

Unlike the unit-test programs in the textbook and lecture, we can't predict the exact results of calls to `run`, because of the random nature of the program. Instead, write code to test that the sum of the number of two-head tosses, two-tail tosses, and head-tail tosses adds up to the total number of trials. (Hint: In the sample output below, we display `true` or `false` for this result by just printing out the result of a boolean expression. So if we ran it on a buggy CoinTossSimulator, it might result in `false`. NOTE: we will be doing such a test on your program.)

Your output should look like the following. This shows only *part* of a sample run of our tester program. A few explanatory notes first:

- `exp` below means expected results. These are meant to be computed a different way than the actual results (i.e., you would not call `getNumTrials()` for the value).
- the part in italics is not the literal output you should produce, but describes what we left out here. The parts in **red** are parts we would like you to pay particular attention to here -- your output will not appear in red.)

```
After constructor:
Number of trials [exp:0]: 0
Two-head tosses: 0
Two-tail tosses: 0
One-head one-tail tosses: 0
Tosses add up correctly? true

After run(1):
Number of trials [exp:1]: 1
Two-head tosses: 0
Two-tail tosses: 1
One-head one-tail tosses: 0
Tosses add up correctly? true

After run(10):
Number of trials [exp:11]: 11
Number of trials:
Two-head tosses: 2
Two-tail tosses: 3
One-head one-tail tosses: 6
Tosses add up correctly? true
```

the testor hasn't been reseted, so the whole trails will be added up.

```
After run(100):
Number of trials [exp:111]: 111
Two-head tosses: 28
Two-tail tosses: 30
One-head one-tail tosses: 53
Tosses add up correctly? true

[ . . . output for tests with different number of trials were here . . .]

After reset:
Number of trials [exp:0]: 0
Two-head tosses: 0
Two-tail tosses: 0
One-head one-tail tosses: 0
Tosses add up correctly? true

After run(1000):
Number of trials [exp:1000]: 1000
Two-head tosses: 265
Two-tail tosses: 229
One-head one-tail tosses: 506
Tosses add up correctly? true

[ . . . output for other tests were here . . .]
```

need to print this!

Remember you won't get these exact numbers because of the random nature of the simulation.

Note: When you test a method such as `run` which has a restriction on its parameter (in this case the restriction is that the value must be greater or equal to one) it means that the behavior of the method is undefined if that precondition is not met. That means that your code for `run` does not have to handle that case, and your tester program should not test that case.

Hints on graphics programming

Most of the graphics primitives you will need for this program are covered in the graphics sections at the end of Chapters 2 and 3 of the textbook, except for a few things we will discuss here. So, you will not need to go hunting through the online documentation or random web sites to figure out how to do the necessary drawing. More specifically: how to draw a filled rectangle is illustrated in the alien face example in textbook section 2.10.4; and the start of the section of this assignment on class design discusses another example from the textbook that has a similar object-oriented design to this one.

Your program may use a fixed size for the width of each bar, and for the buffer-space between the tallest possible bar plus its label and the top and bottom of the window (the solution whose results we showed earlier also does this). Any such constants in your program need to be named constants (see section on grading criteria below, for more information). For the purposes of this assignment you do not have to worry about the fact that if we resize the window small enough horizontally, the labels centered under each bar, and eventually the bars themselves will start running into each other.

(Note: named constants would also be helpful to map the bar colors to what they are used for, e.g., constant `HEAD_TAIL_COLOR`.)

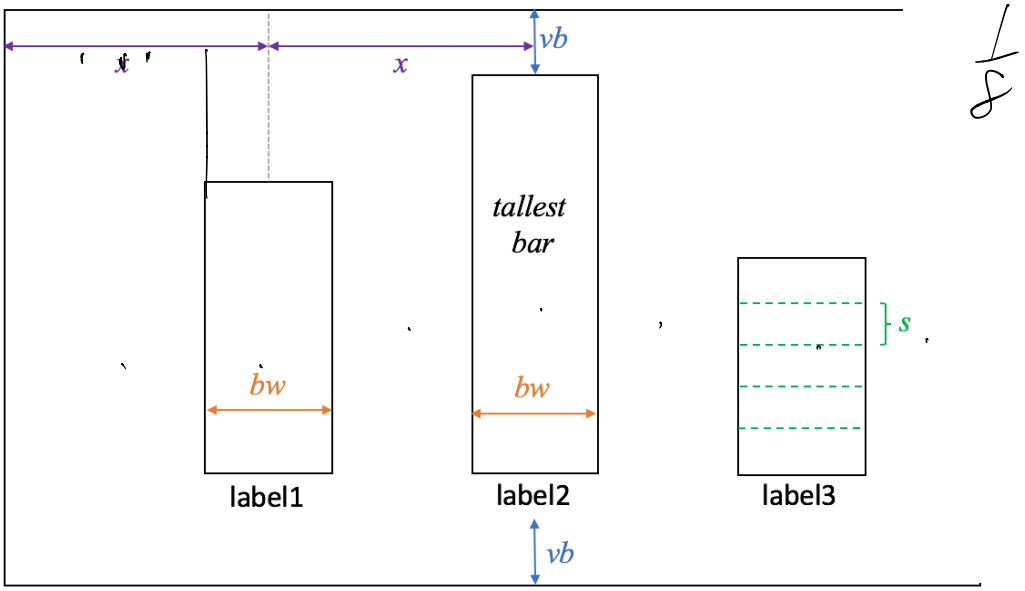
The `JComponent` methods, `getWidth()` and `getHeight()`, which get the width and height of the component, will come in handy here. Since `CoinSimComponent` is a subclass of `JComponent` you can directly call those methods from your component object. For an example of such calls, see the CarComponent class included in

the starter code (and discussed further here). (This is similar to how we could call the inherited `get` and `set` methods from `GregorianCalendar` objects in lab 2, even though they were defined in the superclass, `Calendar`.)

To make sure all the necessary information appears on the window and in the right place, you will also need to know the dimensions of the label you will be displaying (here we'll just use the default font size for the given graphics context). This is not covered in the textbook, so here is a code snippet:

```
String label = "Hello, world!";   // suppose this is the label you want to display
Font font = g2.getFont();
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D labelBounds = font.getStringBounds(label, context);
int widthOfLabel = (int) labelBounds.getWidth();
int heightOfLabel = (int) labelBounds.getHeight();
```

The following diagram illustrates some of the specification for how the window should be laid out -- it is not meant to show coin-toss output, per se. (You may want to compare this with the earlier screenshots.) It will also help illustrate the meaning of the parameters to the `Bar` constructor.



- *x* bars are evenly spaced: center of middle bar is centered in window. Depends on the window size.
- *bw* bar width. This is a fixed value in the program.
- *vb* vertical buffer. This is a fixed value in the program. Tallest bar means the tallest possible bar you could draw (i.e., for this program, if all the trials came out the same way).
- label1, label2, etc. labels on a bar. These are centered under each bar.
- *s* (s is for `scale`) one application unit in pixels. So, this bar is five application-units high (`barHeight` is `5`) The scale depends on window size.

To relate the last item to our program: `scale` and `barHeight` are parameters to the `Bar` constructor. For our program the application unit in the bar graph is a coin-toss trial.

How to communicate information between objects

There are several techniques to communicate information between classes and methods of classes, including

via parameters and return values of methods. In particular, here we have the issue of receiving some information in `main` in `CoinSimViewer`, that is, the number of trials, but needing to use that information in the component. To do this, your `CoinSimComponent` class will need to have its own constructor (Note: this is different than the simpler component examples in the book). From `main` you can pass the information to that constructor, and then, if you also need access to it in other methods, you would save it in an instance variable.

Recall that you never will be calling `paintComponent` yourself, nor are you allowed to change the parameters to it.

README file

For this and all other programs you will be required to submit a text file called `README` with your assignment. In it you will initial the certification we mentioned earlier. This is also the place to document known bugs in your program. That means you should describe thoroughly any test cases that fail for the the program you are submitting. (Not your bug history -- just info about the version you are submitting.) You should also document here what subset your solution implements if you weren't able to complete the whole program (more about that in the next section). You can also use the `README` to give the grader any other special information, such as if there is some special way to compile or run your program (this would be unusual for students who complete the assignment).

For this program, also put the answers to the following questions in the `README`:

1. In CS 455 what code from the web are you allowed to use in your assignment solution, assuming you modify that code once you get it? (No clue? Hint: Reread the syllabus.) Note: no need to mention the "starter files" for the assignment in your answer.

2. What is the probability of each of the three outcomes in a trial: two heads, one head and one tail, and two tails? Why is one of them different than the other two?

3. Roughly how many trials do you need to do before you mostly get results within 1% of these probabilities? Do several simulations using your program to determine an answer to this.

4. Roughly how many trials can you do before it crashes with an overflow error? State the largest value you tried where it still worked correctly, and one where it overflowed. You can get an idea of what general values to try by looking at the limits on Java integers (see Section 4.1.1 of the text). Play around with your program to verify that it actually works for very large values that are within the limits. Note: this kind of testing can sometimes turn up bugs, so it's good to do. Warning: For very large values you may have to wait a fair amount of time before you get the results.

Grading criteria

This program will be graded based on correctness, style, and testing. Programs that do not compile will get little or no credit. However, an incomplete program can get some correctness points if it has partial functionality (you document the partial functionality in the `README`, discussed above). This grading policy is to encourage frequent testing of subsets (discussed earlier in the section on incremental development). Also, for incomplete programs, the style score will be scaled according to how much is completed.

We have published a more complete set of style guidelines for the course on the assignments page, but here are a few things to pay particular attention to for this program:

- documentation. You need to supply an overall comment for each class, and detailed comments about the interface of each method (so called method comments). For the main program you need to supply a comment describing what the program does, and how to run it. (We have already provided the interface comments for the `CoinTossSimulator` and `Bar` classes since we specified those interfaces for you.) This was described in more detail in lecture and the textbook (see Section 3.2.4). Use in-line comments to explain any confusing code ("this is a for loop" type comments are *not* helpful).
- named constants. There are some numbers mentioned in the assignment description as well as other values that are described to be fixed in the program. Each of these should be given a descriptive name (e.g., `BAR_WIDTH`) in the program so it would be easy to change the value later. Named constants in Java are discussed in section 4.1.2 and programming tip 4.1 in the textbook.
- private data. You should never need public data. Rather, clients should only be able to access data through methods The rationale for this is discussed in section 3.1.3 of the textbook.
- good identifier names. Use descriptive names for variables, parameters, and methods. Also use Java naming conventions. Details in item 6 of course style guidelines. Sections 2.2.3 and 4.1.2 of the textbook discuss more about naming.
- good/consistent indenting. Use the conventions from the textbook or lecture.

For this program only, you do not have to worry too much about method length (guideline #7), and while you should document any instance variables that are not obvious from their names, you do not have to worry about representation invariants (item #15).

Implementing the required class design and answering the `README` questions will also be part of your style/documentation score.

How to turn in your assignment

Make sure your name, NetID, course, assignment, and semester are at the top of each file you submit (for source files, they would be inside of comments), for any assignment you submit for this course. You will lose a point on the assignment if this information is missing.

The files you need to submit are the ones shown in bold in the earlier section on assignment files.

No matter where/how you developed the code, we will be grading it on Vocareum using the java compiler and virtual machine there.

If you developed your program outside of Vocareum, for example, using Eclipse on your laptop, you'll need to upload your code to Vocareum and retest it completely before you submit it. *Do not wait until the final due date/time is imminent before testing it on Vocareum.* Please read the earlier section on using another IDE for more details on this.

How to submit your program When you are ready to submit the assignment press the big "Submit" button in your PA1 Vocareum work area. *Do not wait until the final due date/time is imminent before attempting to submit for the first time.* You are allowed to submit as many times as you like, but we will only grade the last one submitted.

What happens when you click submit. Vocareum will check that you have the correct files in your work area and whether they compile. *Passing* these submit checks is not necessary or sufficient to submit your code (the graders will get a copy of what you submitted either way). (It *would* be necessary but not sufficient for getting full credit.) However, if your final submitted code does not pass all the tests we would expect that you would include some explanation of that in your README. One situation where it might fail would be if you only completed a subset of the assignment (and your README should document what subset you completed.)

The results of the submit checks will appear on your terminal window. You can also access them by going to the "Details" menu, and choosing "View Submission Report".

If you are unsure of whether you submitted the right version, there's a way to view the contents of your last submit in Vocareum after the fact: see the item in the file list on the left called "Latest Submission".
