COMS W1007x: Honors Intro to Computer Science
Fall, 2012
Assignment 1: Due Tuesday, September 25, 2011, 1:10pm, in class

Part 1: Theory (40 points)

The following problems are worth the points indicated. They are generally based on the book's exercises at the ends of Chapters 1 and 2, under the section heading "Exercises" and "Programming Projects". They cover computer and Java fundamentals and internals, plus the basics of applets.

The instructor is mindful of the necessity for learning Unix, Emacs, and Java infrastructures, and has taken this cultural transition into account. This assignment therefore gives more credit than will be usual for the programming section, and it has more than the usual interconnectedness between the Theory Part and the Programming Part.

"Paper" programs are those which are written on the same sheet that the rest of the theory problems are written on, and are not to be text-edited, compiled, executed, or debugged. For fairness, there will be no extra credit for doing any work beyond the design and coding necessary to produce on paper the objects or object fragments that are asked for, so computer output will have no effect on your grade. The same goes for the Theory Part in general: clear handwriting is sufficient, and computer-generated documents will earn no extra credit.

1) (5 points) Aspect oriented design: Start at the excellent tutorial found at http://www.developer.com/design/article.php/3308941 online. Then give four examples of "aspects" of design that object-oriented design does not address. (Just list the examples; you don't have to explain them.) Then, say what "advice" is. Then, say what a "weaver" does. Then, give some convincing reason, from a design perspective, why aspect-oriented programming is so clumsy to integrate with object-oriented programming.

2) (5 points) Using the web, find a block diagram for the iPhone 4. Draw it, and make sure to cite your source, in order to document which diagram you are using, and in order to avoid charges of academic dishonesty. Now, compare its components with those shown in Figure 1.1 on page 3 of the text. Which are extra? Which are omitted? Would the object-oriented approach of Java work well with such a new view of a computer: why or why not?

3) (3 points) Look up the prefixes for *negative* powers of 10, and what they mean in base 2 and base 10. Again, make sure you cite your source. Is there evidence in this table for an effort at standardization for the design of these prefixes, even if that effort was late? Explain.

4) (4 points) API familiarity: Find the official Java API (official name: "Java(tm)Platform, Standard Edition 7 API Specification"). Note that Java is owned by the Oracle Corporation, and note that the if you ask Google "Java API", you still get Edition 6, which is no longer the latest version. Bookmark the API, as you will be using it a lot in the course. Copy as part of your answer its URL. Then, give the names (just the names) of the *non-deprecated* fields, constructors, and methods, complete with their parameters and return values, for the class Date. Be careful, there are *two* classes called Date! Use the original one. You do not need to give any text descriptions, nor any of those fields or methods that have been *inherited* from any superclasses. This is meant to be an easy question, to get you familiar with the API; just copy

down those parts requested. But now, answer this: What does this particular class say about the design of Java: in what ways is it good, or bad, or both?

5) (10 points) Using the "Tester" and "Worker" object-oriented design given in class, write the paper program for a three-class application (Tester, Worker, System) that that does "Hello, World", except that it says, four times in a row, "Hello, S! It is N milliseconds since January 1, 1970.", where S is a *constant* string given from the Tester, and N is a *field* integer computed in the Worker. The three successive values of N should differ. Hint: the static class System will make this easy; look it up in the API.

First, Draw the UML for your design, including that for the class System--although you only have to show those fields and methods in System that you actually use. You do not have to draw any classes that System uses.

Then, write your "paper" program. You do not have to text edit, compile, execute, or debug. Write it on the same answer sheets as the rest of this Theory Part. However, *unlike* most other theory assignments, you will later be asked in this assignment to *execute* a variation of this paper program. That variation will include proper Javadoc. This full documentation and execution will be part of Part 2, as an exercise in learning CUIT Unix and Java--and also of using object-oriented design and reuse. In general, you won't *have* to use Javadoc for any theory parts, although you will have to use it in the Programming parts, so for this first assignment you might as well write it down as part of your answer here as well.

6) (5 points) Consider the following requirements document from a client. Indicate what are the potential classes, their methods, their fields, and their constants: "Our pizza shop sells many pizzas: cheesy ones, meaty ones, normal size ones, even very small ones. We cook them, then sell them cheaply. We sell drinks, too, regular and diet. But we don't charge for napkins."

7) (5 points) Using the rule that 2^10 == 10^3, show how the book on page 71 derives the min and max values for the primitive type called float.

8) (3 points) Scanner is a recently added class in Java that shows some of the encapsulation and inheritance properties of the language, as it depends on the design pattern called Iterator which is also part of the Java language. Look up both Scanner and Iterator in the API, and say why Scanner violates the Principal of Least Surprise in a serious way. Hint: what three methods does Iterator require?

Part 2: Programming (60 points)

This assignment is intended to walk you through the mechanics of CUIT, Unix, Emacs, etc., and then give you a beginning exercise in web-based programming.

Please recall that all programs must compile, so keep a working version of each part before your proceed to the next.

Step 1 (12 points): Getting started. Unlike future homeworks, this particular Step is not related to the following Steps; basically, this is an immigration exercise to make sure everyone is up to speed in using CUIT and Cunix. Implement the "Hello, World" program you wrote for the Theory Part.

This is not a trick question; it is supposed to be easy. This part of the assignment is intended to get you started with something that you know works, to give you the infrastructure for writing applications, and to get you acquainted with how to do it on the CUIT systems if you haven't done so already. Again, you should have a "worker" class and a "tester" class.

Turn in a listing of your code, and some trial runs. Most importantly, use the javadoc conventions to properly document your classes and your method,s and create and submit the javadoc outputs for this part. See Appendix I.

Step 2 (13 points): An exercise in simple applet design. Implement the Snowman applet exactly as it is given in the book-- except with the proper Javadoc added. The code for both the Snowman.html and Snowman.applet is available for download from the book's "Companion Website".

Again, this part is intended to get you familiar with the applets and with the website that has code we will be using and modifying throughout the course.

You get separate and additional credit for this Step, and if you do it, you should submit it in addition to Step 1. Give your code, and capture a screen shot and print it out, and submit the Javadoc.

Step 3 (18 points): Redesign. The code for Snowman is a mess. It has very many imbedded constants. Only two constants are properly declared, the MID and TOP (which refer to the *head*, although this is not clear in the code). Change the code in two fundamental ways. First, make sure all the parameters have no integer literals in them. For example,

```
page.fillOval (MID-20, TOP, 40, 40)
```

should really be:

```
page.fillOval (MID-HEADDIAMETER/2, TOP, HEADDIAMETER, HEADDIAMETER)
```

or, better:

```
int headDiameter = 4*UNIT;
page.fillOval (MID-headDiameter/2, TOP, headDiameter, headDiameter)
```

where UNIT is a constant that is of useful size, here, 10.

Second, although MID and TOP allow you to easily move the snowman left, right, up, and down, they do not allow you to change its width and height. This will be important for the creativity step below. So, redo the applet so that a snowman works as a shape that can be placed anywhere, and with any width and height. Conceptually, you want something like page.drawSnowperson(x, y, w, h);

Make sure you use plenty of comments to indicate what your redesign does.

If you do this step, you still have to submit your output from Step 1, but not any separate output from Step 2. But the output for this step is the same as required for Step 2: code listing, trial runs, Javadoc.

Step 4 (17 points): Creativity Step. Further stretch the boundaries of your object-oriented design. If you did Step 3, this should be relatively easy.

Make your system draw three snowpeople of the *same* height but *varying* Body Mass Index, or BMI. You will have to look up what BMI means. So, by changing *only one* constant, the height, have your system draw an underweight snowperson (BMI==15), a healthy snowperson (BMI==20), and an overweight snowperson (BMI==25). Display them next to each other, and use drawString to label them. Please note that BMI is computed by a number that scales height*height*width, so you will have to determine how to position your snowpeople and compute the appropriate widths for a given BMI correctly.

To be creative, you should also have the snowpeople vary in some way, depending on their BMI: they can be different colors, or have different facial expressions, etc. Make sure you document your design.

Do this for three different heights. Note that this Step should further illustrate the ways in which object-oriented design tools actually work to make things easier: in order to make a skinny snowperson, you start with a regular snowperson, and then change it a bit. This is a primitive form of inheritance.

If you do this part, you still have to submit your output from Step 1, but not from Steps 2 or 3, as your output on Step 4 takes care of those two predecessor steps. But it will require the same things that are detailed in Step 2.

General Notes:

For each Step, design and document the system, text edit its components into files, compile them, debug them, and execute them on your own test data. When you are ready, submit the text of its code, a copy of your testing, and whatever additional documentation is required. Make sure that the source code is clear and any user interface is comprehensive and informative.

Write your classes clearly, with a large block comment at the beginning describing what the class does, how it does it, and justify how it will be tested. Clear programming style will account for a substantial portion of your grade for both the Theory Part and the Programming Part of this assignment. For one reasonable set of suggested practices and stylistic guidelines, see the book’s Appendix F. But it is far more important to be consistent: PoLS, everybody!

Checklist:

Here is a checklist for submission. Please note that this is designed to be a *general* guide, and you are responsible for *all* things asked for in the text of the assignment above, whether or not they appear below.

For theory: Hard copy, handed in to Prof or TA by the beginning of class. No extra points for using a text editor. Neatly stapled! Name and UNI attached.

For programming: Hard copy, handed in to Prof or TA by the beginning of class. All code, all testing runs (cut and pasted from console, and/or captured screenshots), all Javadoc if Javadoc is required. Neatly stapled, and separately stapled from the theory! Name and UNI attached.

Also for programming: Soft copy, submitted to Courseworks by the beginning of class, in a tarball created by CUIT Unix tar command, given below. Note that "myUNI" should be replaced with your UNI (duh!), and that HW1 will change in further assignments, becoming HM2, etc. *Please* use this convention as it makes the TAs’ job much easier:

```
tar -cvzf myUNI_HW1.tar.gz whateverMyDirectoryIs
```

Also for programming: Include the softcopy of javadoc html. Name and UNI included as comments in every class. The code must compile. Your last electronic submission before deadline is the official one that will be executed if needed.