**CSE 1325 – Spring 2013**

**Object-Oriented and Event-driven Programming in Java**

**Lab 3**

Assigned: Thursday 11:59 pm Feb 28, 2013

Design Document Due: Tuesday 11:59 pm Mar 12, 2013

Due: Tuesday 11:59 pm Mar 19, 2013

**Objective:**

The theme of this project is to continue designing and updating classes and methods for managing an apartment complex using object-oriented design (OOD) principles. In the industry of software development, updating existing code is a very common task, which you will be experiencing. Feel free to re-code your classes. You will have the freedom to implement the internal functionality of your objects but the class name and signature of each method will be defined in a standard Javadoc.

**Description:**

Now that you have experienced structuring your own program and designing your own tests, you will now you adhere to these testing specifications as you would in the real world by learning to refactor your code (by changing method names, parameters, etc) to meet the requirements. Specifically, the code that will be most modified will be:

* Method names
* Method parameters
* Specific, testable return types such as Boolean
* Failing conditions

Design classes, fields (attributes) and methods for the problem description below. Make sure you understand and differentiate between public/private/protected fields and methods. Use static fields/methods where appropriate. Make sure you have at least one constructor per class. Each class should have appropriate get and set methods. Each class should have a toString() method (and other formatting methods as needed).

Files accompanying this project:

* Javadocs describing the minimum methods that need to be implemented
* info.txt a sample file
* Constants.java
* *This* document
(Note: that was a Java joke)

**ApartmentComplexTest**

As in the previous lab, a “test” class will drive the application and will now be named, ‘ApartmentComplexTest’ (if it wasn’t already named that). The test class should open and process the info.txt, print the menu seen below, accept user input, and execute methods to perform tasks, and resulting understandable output. All inputs (as appropriate for the menu) should be accepted from the terminal window and should prompt the user for the input with appropriate messages. If bounds are appropriate, please indicate it in the prompt and check for input errors. Expect the user to enter the proper data type. As seen in the class descriptions, nearly every setter method must return a boolean type, and the ApartmentComplexTest must act on it the possibility that the method might fail; even if it only displays a message such as, ‘Unable to Pay Employees). In other words don’t let methods fail silently.

1) Lease an apartment to a tenant

2) Terminate a lease of a tenant

3) Hire an employee

4) Pay an employee

5) Remove (fire) an employee

6) Display all employee details (along with their specialization)

7) Display all tenant details (including lease information, carport/garage)

8) Display all apartment details (occupied apts. and available apts. separately)

9) Display Bank account information for the Apartment Complex and all Humans

10) Collect Rent from tenants

11) Show Apartment Lease History

0) Exit program

**The ApartmentComplex Class**

As stated earlier you may implement more methods than those listed in the accompanying JavaDoc. However those found in the Javadoc are the minimum and must function properly in order to receive full credit.

Note: The typical structure of a main function would be a do while loop containing the menu and switch statement. To keep the code clean it is suggested that each switch case call another static function.

**The Apartment class**

There should be no limit on the number of apartments. Once an apartment has been created, it can’t be destroyed. When asked to list unoccupied apartments, list apartments in which the leases have run out.

**The Constants Interface**

The Constants.java file is given with this assignment. Use the keyword, ‘implement’ to use it in your classes.

**The sample info.txt file.**

The info.txt file is also given as a template to test your file input method. The best way to effectively test your code is to make your own info.txt file.

Note: When reading in a file you have to change the ‘Working Directory’ to the location of your info.txt file. This configures Netbeans to execute your code at that location in the filesystem. This allows your java program to find your data file with a relative file and path name like, ‘info.txt’ as opposed to an absolute path name such as, ‘C:\Users\etc\...\info.txt’.



**The Human class**

Is an Abstract class which cannot be directly instantiated.

**The Tenant class**

This class should not have its own BankAccount. It will already inherit one from the Human class.

**The Employee class**

This class will also inherit a Human methods to work with money including methods like despoit() and withdraw().

**The BankAccount class**

This class will be used to exchange currency between objects.

Note: In the real world float and double datatypes should NEVER be used for money because the data structure inherently loses precision. You will learn more about this in a Numerical Analysis or Computational Methods class (or you could just ask Sean during his office hours). If you implement BankAccount without the use of float and double in a safe way you will get **Bonus** points. Some safe ways include, the BigDecimal object, or BigInteger with an offset of two so you could effectively count pennies. Regardless of how you implement BankAccount you must have the defined methods fully functional (in other words, your class must still input and output float datatypes).

Javadoc

Submit the files generated by the Javadoc for your program. Every method should have its meaningful description for use by someone else. Use the relevant Javadoc tags specified in -http://en.wikipedia.org/wiki/Javadoc or <http://java.sun.com/j2se/javadoc/writingdoccomments/>

If you are having difficulty generating Javadocs (Run -> Generate Javadoc), then you might need to create a new package then move your classes into that package.

If you are missing classes or methods in your Javadocs make sure the keyword public is being used.

**What and How to Submit**

For Lab #3 WRITE A DESIGN DOCUMENT FIRST. The design must include

 a) a UML class diagram of all the classes you expect to write,

 b) brief (one line) descriptions of each data member of each class

 c) brief (one line) descriptions of each method (member function) of each class,

 d) some indication of the relationships between the classes.

The parts b), c), and d) of the design document may be written as lines of text or as diagrams (such as a diagram that start with the main function at the top and all others below it, a tree, UML, etc.) or as some combination of those, but it must include the information listed above. Each class should encapsulate one thing and each piece of data at a lower level should have a more specific purpose than the class that owns it. Be sure to include all the classes that are described in this lab assignment. See the website or this assignment for the DESIGN DOCUMENT due date. It is usually ONE WEEK PRIOR to the lab due date.

A Lab #3 Design Document must be turned in in order for your Lab #3 assignment to be graded.

**Standard Java conventions and coding style:**

Be sure to observe standard Java naming conventions and style. These will be observed across all projects for this course, hence it is necessary that you understand and follow them correctly. They include:

* Class names begin with an upper-case letter, as do any subsequent words in the class name.
* Method names begin with a lower-case letter, and any subsequent words in the method name begin with an upper-case letter. This is called "camelCase".
* Class, instance and local variables begin with a lower-case letter, and any subsequent words in the name of that variable begin with an upper-case letter.
* Each set of opening/closing curly braces must be in the same column. All user prompts must be clear and understandable
* Give meaningful names for classes, methods, and variables even if they seem to be long. The point is that the names should be easy to understand for a person looking at your code
* All static final variables will be upper case identifiers

In addition, ensure that your code is properly documented in terms of comments and other forms of documentation wherever necessary.

**What and How to Submit**

For this project please submit using the Blackboard system at elearn.uta.edu. Please submit a .zip file that contains:

* Each .java file used in development with the appropriate name for the included class
* A script file of the session on omega
* A development notes file (standard text or word file) that describes:
	+ least three logical errors that you found while developing your code,
	+ how you solved them, and the
	+ Total amount of time you spent on this project in terms of hours.
	+ List all the parts of your program that are not functioning
* The JavaDocs for each class

The .zip file containing your project must have the name format of: **proj3\_1000number\_firstname\_lastname**

Upload this code to elearn.uta.edu

**A Few Helpful Tips:**

* Netbeans by default recompiles your class every time you save your file. If you want to force everything to be recompiled, press the ‘clean and build’ button.
* Ctrl + space in most IDE’s will show a list of available members (or variables) and methods (or functions).

**Grading Rubric:**

**Code: (62%)**

 Class header and method headers for all classes (4 points)

 Comments (line comments and block comments) (3 points)

 Style (following Java conventions, indentation, consistency, meaningful identifiers, lateral separation of code from line comments, etc.) (3 points)

 Correct renaming of methods as described by Javadoc documentation given for the lab (11 points)

 Correct creation of new classes as described including constructors and the toString function (10 points)

 Correct development of methods which check for failure and which return testable results (8 points)

 Correct implementation of file input (8 points)

 Correct use of information in Javadoc material (7 points)

 Test plan includes at least two tests for each of the required

 functions of lease, terminate, hire, pay, and fire and uses the

 display functions to verify test results (8 points)

**Output: (38%)**

 User clearly understands what is being requested for input (3 points)

 Required tasks performs correctly

 (lease, terminate, hire, pay, fire, display (3), exit) (15 points)

 Set functions handle inputs correctly including error checking (6 points)

 Get functions provide values safely (no ref passing) (4 points)

 Output gives clear information to explain the values to the user (3 points)

 Output demonstrates all elements of test plan (7 points)

**Grading Deductions:**

 Use of Java public variables (C equivalent to global variables) will result in an overall grade of 0 (zero) [-100 deduction]

 Use of unstructured control elements, like the break (except in the case of switch statements) or continue keywords, will result in an overall grade of 0 (zero) [-100 deduction]

 Labs which have errors in them and **do not terminate normally** will receive an overall grade of 0 (zero) [-100 deduction]

 Late submission of softcopy of code and/or script file or other documents to Blackboard will result in an overall grade of 0 (zero) UNLESS student has obtained prior instructor approval [-100 deduction]

 Use of Java language elements not yet discussed in class by the lab due date will result in potential deduction of points - discuss with instructor before using.

Miscellaneous: If you have questions, e-mail Dr. T (tiernan@uta.edu) and the TA (seanp@uta.edu )