Assignment 1 (Total: 100 points)

**Objective**: This purpose of this assignment is to reinforce the advanced OOP concepts that you have picked in your prior programming courses. You will need to refresh your knowledge on concepts like Inheritance, Polymorphism, Abstract classes before attempting this assignment.

**Problem Statement**

You have been tasked to design the blueprint for several buildings in a neighborhood. A neighborhood can consist of several types of buildings – houses, cottages, offices etc. Obviously, the simplest solution would be to design a separate class for each of these buildings, each with their unique specification. However, after looking at the requirements, you realize that most of these buildings share a lot of common properties with each other as well as some their own unique properties. Can you come up with an efficient design using the OOP concepts you have picked up in class?

**Problem Description**

The overall inheritance hierarchy looks something like this

Diagram

Description automatically generated

This can be read as - the Building class defines a generic blueprint of the buildings in the area. The House and Office classes are specific types of Buildings. The Cottage class is a specialized form of a house.

You need to design these classes using the specifications below.

Building.java

Graphical user interface, text

Description automatically generated

Notes

1. Please refer to the UML rules to interpret the symbols in the UML diagram. Most of the attributes, methods are self-explanatory.
2. We will assume all buildings are rectangular in nature, and their dimensions are defined by the length and width of the building.
3. The lot length and width refer to the dimensions of the land the building is built on top of. Obviously, the area of the lot must be larger than the area of the building. You can assume the inputs to your programs will respect these constraints.
4. The Building class should contain ONLY one parametrized constructor defined above. There should be no other constructor.
5. In addition to the getters and setters, you will also have two methods as shown above to compute the area of the building and the area of the lot.
6. Finally, the toString () method should return a String representation of the building. The text should look like “A 120X35 Building”

House.java

Text

Description automatically generated

Notes

1. House should inherit all public/protected properties and methods from Building.
2. In addition, every House has two additional properties – a owner name and the presence/absence of a pool.
3. The House class will have 3 constructors. You must use techniques like super and this to reduce as much redundant code as much as possible. There should be no repeating lines of code across the 3 constructors.
   1. The first constructor will leave the owner as null and the pool as false.
   2. The second one will accept the owner information from the user and leave the pool information as false.
   3. The third one will accept both the user and pool information and set these properties accordingly.
4. The getters and setters are self-explanatory.
5. The toString method should be overridden and extended to contain
   1. the owner information like “; Owner: John Doe”. In case, the house does not have an owner, it should look like “; Owner: n/a”
   2. the presence of a pool by appending “; has a pool” in addition to printing the building dimensions.
   3. the open space information. If the area of the open land area (i.e., lot area that is NOT covered by the house) is larger than the building area itself. In such cases, you should add “; has a big open space” to string representation.
6. Define a equals () method. 2 houses are considered equal, if the building areas are the same and they both have the same pool information.

Cottage.java

Text

Description automatically generated with medium confidence

1. Cottage should inherit all public/protected properties and methods from the House class
2. In addition, it should also define a property mSecondFloor indicating the presence or absence of a second floor.
3. A cottage will have the same length and width (i.e. it is shaped as a Square)
4. There are 2 constructors for Cottage. You must use techniques like super and this to reduce as much redundant code as much as possible. There should be no repeating lines of code across any of the constructors.
   1. The first only accepts the single dimension for length and width along with the lot length and width and sets these properties accordingly.
   2. The second accepts the owner and second floor information and sets these properties accordingly.
5. To comply with the neighborhoods’ safety standards, you are prohibited from adding a second floor to a cottage after its creation. Define the getters/setters accordingly.
6. The toString method should be overridden and extended to specify that the object of this class “; is a cottage”. If the cottage has a second floor, specify it as “; is a two story cottage”

Office.java

Text

Description automatically generated

1. The Office class will inherit all public/protected properties/methods from the building class.
2. In addition, it should contain a property to store the name of the business occupying the office and the number of parking spaces.
3. It should also contain a static counter that will keep track of the number of offices created in the neighborhood.
4. The Office class will have 3 constructors. You must use techniques like super and this to reduce as much redundant code as much as possible. There should be no repeating lines of code across the 3 constructors.
   1. The first constructor will accept the dimensions of the office and lot and leave the business name and parking spaces to their default system values.
   2. The second constructor will also accept the business name and set the appropriate property.
   3. The third should accept both the business name and the number of parking spaces.
5. You will need to increment the static office counter, every time an office is created. Similar to the constructors, avoid doing this in multiple places. Ideally you should only have to define it once in one of the constructors.
6. The toString () method should be overridden and extended to include
   1. the Business name like “; Business: Walmart“. If not defined, it should contain “unoccupied” as the name.
   2. the parking space information. If the business contains a non-zero number of parking spots, you should append this information like “; has 50 parking spaces”
   3. the total number of offices in the following format “(total offices: 10)”
7. Override the equals method. 2 offices are equal if the building area is the same and they have the same number of parking spots.

Neighborhood.java

This will be your tester class, that will contain the following **static** methods. You do not need to create any instances of any Building here.

1. **String [] getInfo**(Building buildings[]): This should accept an array of building objects and returns a String array containing the String representation for each building. Note: If implemented correctly you do not need to determine the type of building (house, cottage, office) that is passed in.

For example, given the appropriate input, the contents of the String array returned by the above method will look like this

A 3X3 Building; Owner: Harry Potter; has a pool; has a big open space

A 13X31 Building; Owner: Sirius Black

A 10X15 Building; Owner: Severus Snape; has a big open space; is a two story cottage

A 20X3 Building; Owner: Ron Weasley; has a pool; has a big open space

A 5X5 Building; Owner: Dean Thomas; is a cottage

A 5X5 Building; Owner: Draco Malfoy; has a pool

A 5X5 Building; Owner: Victor Krum; has a pool; has a big open space

A 3X3 Building; Owner: n/a

A 10X10 Building; Owner: n/a; has a pool

A 60X35 Building; Business: unoccupied (total offices: 6)

A 35X60 Building; Business: Honeydukes; has 100 parking spaces (total offices: 6)

A 75X75 Building; Business: Three Broomsticks; has 100 parking spaces (total offices: 6)

A 75X75 Building; Business: Ollivanders (total offices: 6)

A 30X30 Building; Business: unoccupied; has 50 parking spaces (total offices: 6)

A 30X30 Building; Business: Gringotts (total offices: 6)

1. **int calcArea**(Building buildings[]): This should accept an array of building objects and return the total **lot** area occupied by each of the buildings.

Unit Tests

Since this is your first assignment, I have attached a Unit Test suite that you can run to test your code against different test scenarios that your program will be tested on. While grading, I will be using a different test suite, so please make sure you are storing things (especially the String representations) in exactly the format specified above. Misplaced or extra whitespaces will cause the unit tests to fail.

To run the Unit Test, simply put in the UnitTest.java into your /src folder (ideally these should go in a separate folder, but for this simple assignment this setup will suffice). When you first import in the code, you may see IntelliJ highlight a bunch of errors. These are because some of the packages needed for the UnitTest to run may need to be added into your project. Simply hover over the highlighted errors, and you should see IntelliJ automatically suggesting you add the appropriate package to the project library. Once the errors have been removed, run the UnitTest.java file from IntelliJ to execute the tests. You should see the results popping up on the bottom right. It will look something like this.

Text

Description automatically generated

If any of the tests fail, you can inspect the output pane on the right, to see which cases failed and what the expected output is. The unit tests are simple methods which runs some method in your code and asserts them against a known output. You can also look into the individual methods themselves and manually compute what the output should be.

**Submission**

1. The .zip for the entire project
2. A .txt file telling me how many of the 18 unit tests passed.

Grading Breakdown

50% - Number of Unit Tests Passed

20% - Documentation

30% - Code Quality – I will be specifically checking for redundant lines of code in this criteria.