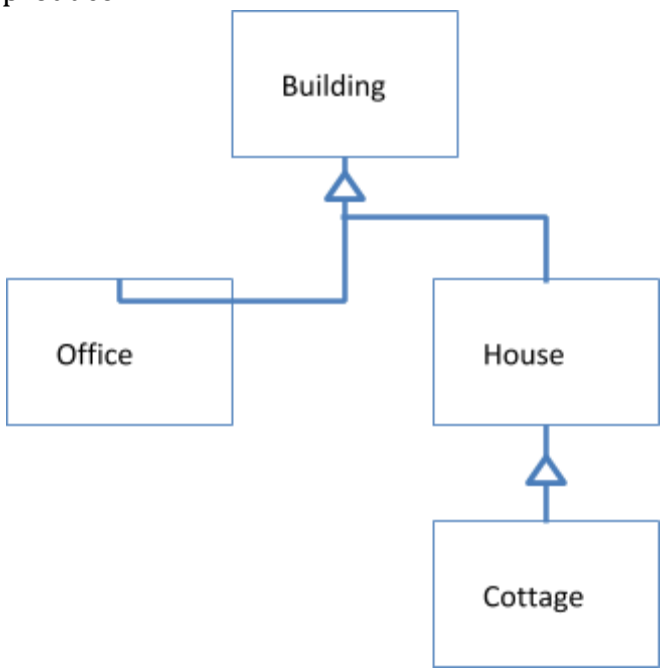
Module 7: Creating Classes Using Inheritance and Polymorphism

Introduction

One of the many benefits of object-oriented design using classes is the ability to create classes that inherit much of their state and behavior from existing classes. This type of systematic software reuse can save time, make debugging and refactoring easier, and takes advantage of the object-oriented principle of polymorphism that's covered in some of this module's video lessons.

This assignment teaches you how to create an inheritance hierarchy consisting of several classes stored in several files that demonstrate the relationships between a super class and its subclasses. The abbreviated UML diagram below will give you an idea of what classes to produce.



After creating these classes, you should test them by running the JUnit tests supplied with the Android Studio project that instantiate (or in other words "create") your code and exercise some of its behaviors (methods).

Learning outcomes

- After you have completed this exercise you will be able to create classes (called "subclasses") that are extend a super class
- Build methods that rely on polymorphism and "late binding" to accept several different classes as parameters
- Create your own testing program for ensuring your class implementation works correctly

Resources

Along with this specification document, you are provided with an Android Studio project to download and use on your computer. This project contains Java files organized into the following two directories:

- ***app/src/main/java/mooc/vandy/java4android/buildings/logic*** -- This directory contains several files you need to implement, as described in the "**What You Need to Do**" section below. It also contains several files whose class implementations are provided for you. In particular, the *BuildingList* and *Logic* classes are provided with this assignment to test the classes you implement. In particular, the *Logic.process()* method in the *Logic.java* file creates a *BuildingList* object and calls its *getHouses()* method to populate an array of *House* and *Cottage* objects and its *getOffices()* method to populate an array of *Office* objects. The *Logic* class modifies the content of the arrays created in *BuildingList* during the grading of this exercise, so running these tests will give you a good idea about the correctness of your class files. In this assignment we will be show how the *Cottage, Office, House,* and *Building* class definitions--along with the use of the *toString()* overridden method-- illustrate a clear example of polymorphism, as well as the advantages this Java language feature provides.

- ***app/src/main/java/mooc/vandy/java4android/building/ui*** -- This directory contains a class and an interface that are provided for you. The *MainActivity.java* file contains the Android Activity that defines UI for this app and calls the *Logic.progress()* method to test your class implementations. You don't need to know anything about the contents of this file. The *OutputInterface.java* file contains a Java interface called *OutputInterface* that defines methods (which are implemented by *MainActivity*) that the classes you write can use to print various messages to the UI. We therefore recommend you examine *OutputInterface* to learn what methods are available for use in your classes.

In addition to these files, there are also unit tests in the ***app/src*** directory. Running these unit tests will provide you feedback on the correctness of your class implementations. They are also the same unit tests used by the auto-grader.

If you choose to have your solution evaluated by your peers (which is optional and doesn't count towards your final grade on this assignment) they will need to download and compile your code. Your code should therefore be importable into Android Studio, should compile without error, and should then run correctly on an emulated Android device.

What You Need to Do

Create each of the classes described in the UML diagrams below. Recall that a '-' symbol before a method or field indicates that it should be private, whereas a '+' symbol indicates the method or field is public. Adhere to the "don't repeat yourself" principle, which means writing a minimal amount of code by depending on inheritance and reusing methods that

have already been written elsewhere in your class implementations. After you have completed the class files, it is recommended that you run the Junit tests to ensure everything is working properly. The Junit tests can be run by simply right-clicking on the LogicUnitTests.java file and then choosing Run 'LogicUnitTests' menu option.

Turn in: **Building.java**

Building
- mLength : int **- mWidth : int** **- mLotLength : int** **- mLotWidth : int**
+ Building(int length, int width, int lotLength, int lotWidth) // constructor **+ getLength() : int** **+ getWidth() : int** **+ getLotLength() : int** **+ getLotWidth() : int** **+ setLength(int) : void** **+ setWidth(int) : void** **+ setLotLength(int) : void** **+ setLotWidth(int) : void** **+ calcBuildingArea() : int** **+ calcLotArea() : int** **+ toString() : String**

Notes on ***Building****:*

- The setter and getter methods for Building are public since the auto-grader will call these methods to test that they work, so they must be public. You can envision a circumstance where a lot size *could* be changed after the creation of the object, this would simply require a public method that call *setLotLength()* and *setLotWidth()* and would require careful input validation. That circumstance is not included in this assignment, however.
- Create the *toString()* method to simply returns a string representation of the Building object (see the example output at end of this file for suggested formatting).

Turn in: **House.java**

House extends Building
- mOwner : String **- mPool: boolean**
+ House(int length, int width, int lotLength, int lotWidth) // constructor **+ House(int length, int width, int lotLength, int lotWidth, String owner) // constructor** **+ House(int length, int width, int lotLength, int lotWidth, String owner, boolean pool) // constructor** **+ getOwner() : String** **+ hasPool() : boolean** **+ setOwner(String) : void** **+ setPool(boolean) : void**

+ toString() : String **+ equals(Object) : boolean**

Notes on ***House****:*

- *mPool* is a Boolean that indicates whether the house has a pool or not
- The first constructor leaves *mOwner* as *null* and *mPool* as *false.* Use *super* to take advantage of code already written. The second sets the *mOwner*, the third the *mOwner* and *mPool* status.
- Override the *toString()* method of the *Building* class by defining your own *toString()* method in the *House* class. Formatting suggestions can be seen in the sample output at the bottom of the file. Use the calcBuildingArea() and calcLotArea() methods to determine if the lot has a so-called big "open space" by checking if the lot area is larger than the building area. If the lot does have such a big open area, then append "; has big open space" to the string output.
- Override the default *equals()* method. Two houses are equal if their building areas are equal and their pool status is the same.

Turn in: **Cottage.java**

Cottage extends House
- mSecondFloor: boolean
+ Cottage(int dimension, int lotLength, int lotWidth) // constructor **+ Cottage(int dimension, int lotLength, int lotWidth, String owner, boolean secondFloor) //** **+ hasSecondFloor() : boolean** **+ toString() : String**

Notes on ***Cottage****:*

- *mSecondFloor* can not be changed once a *Cottage* is created to indicate that we do not permit adding a second floor to a cottage that is already built
- a *Cottage* (in this assignment) is always square and so, the *length* is equal to the *width* and are both described by the parameter dimension. Use *super* to construct a cottage and send the appropriate data.
- Override the *toString()* method of the House class by defining your own *toString()* method in the *Cottage* class. Formatting suggestions can be seen in the sample output at the bottom of this file.

Turn in: **Office.java**

Office extends Building
- mBusinessName : String **- mParkingSpaces: int** **- sTotalOffices: int //static variable**
+ Office(int length, int width, int lotLength, int lotWidth) // constructor **+ Office(int length, int width, int lotLength, int lotWidth, String businessName)**

// constructor **+ Office(int length, int width, int lotLength, int lotWidth, String businessName, int parkingSpaces) // constructor** **+ getBusinessName() : String** **+ getParkingSpaces() : int** **+ setBusinessName(String) : void** **+ setParkingSpaces(int) : void** **+ toString() : String** **+ equals(Object) : boolean**

Notes on ***Office****:*

- *sTotalOffices* is a private static field that should be initialized to 0 in the class declaration.
- The first constructor sets the building and the lot size, leaving the *mBusinessName* null and *mParkingSpaces* 0. Increment the *sTotalOffices* static variable by 1.
- The other constructors also increment the *sTotalOffices* static variable by 1. Use super or this to take advantage of constructor code already written.
- Override the *toString()* method of the *Building* class. Formatting suggestions can be seen in the sample output shown at the bottom of this file. If the *mBusinessName* is null, output the string *"unoccupied"*.
- *O*verride the default *equals()* method. Two office buildings are equal if their building area and number of parking spaces is equal.

It is important (and helpful) for you to test your class files before moving on to use them in client files or send them out to be used by other programmers. Testing a class often involves running your class through a series of method calls designed to test each behavior of the class. You also need to test constructors by simply instantiating objects and then exploring their state. You will not turn in any test files that you may create, but use them to help you debug your class files as necessary.

Turn in: **Neighborhood.java**

Create a Java utility class called *Neighborhood.java* that provides static helper methods the print a *BuildingList* and calculate the area of a *BuildingList* . A Java utility class should always be declared as final and have a private constructor, as described at https://en.wikipedia.org/wiki/Utility_class.

Neighborhood
+ print(Building[] buildings, String header, OutputInterface out) : void // static method
+ calcArea(Building[] buildings) : int // static method

Notes on ***Neighborhood****:*

- The *print()* method takes as an input parameter an array of polymorphic *Building* objects. Also, pass a string to print out a header for the objects, which works since the *toString()* method is the only method you'll need for these objects , and it's inherited by the House, *Cottage,* and *Office* classes from the *Building* class (and actually overridden) .

You will therefore never need to determine if you were sent a *House* array, *Cottage* array, or an *Office* array and you will never need to cast in this method. The *House* array can actually contain both House and Cottage objects because one extends the other. A *Cottage* (from the subclass) can act as a House (which is the super class).

- Create another method called *calcArea()* that accepts an array of *Building* objects and returns the total (lot) area of the objects in the array. Since the *Building.calcLotArea()* method relies on polymorphism it's the only method you'll need since it's inherited by all the classes extended from the *Building* class.

Sample Output

```
Houses
----------
Owner: George Washington; has a pool; has a big open space
Owner: John Adams
Owner: Thomas Jefferson; has a big open space; is a two story cottage
Owner: James Madison; has a pool; has a big open space
Owner: James Monroe; is a cottage
Owner: John Quincy Adams; has a pool
Owner: Andrew Jackson; has a pool; has a big open space
Owner: n/a
Owner: n/a; has a pool

Offices
----------
Business: unoccupied (total offices: 16)
Business: Bridgestone/Firestone; has 100 parking spaces (total offices: 16)
Business: Caterpillar; has 100 parking spaces (total offices: 16)
Business: Cracker Barrel (total offices: 16)
Business: unoccupied; has 50 parking spaces (total offices: 16)
Business: Nissan (total offices: 16)


Total neighborhood area: 982128
```