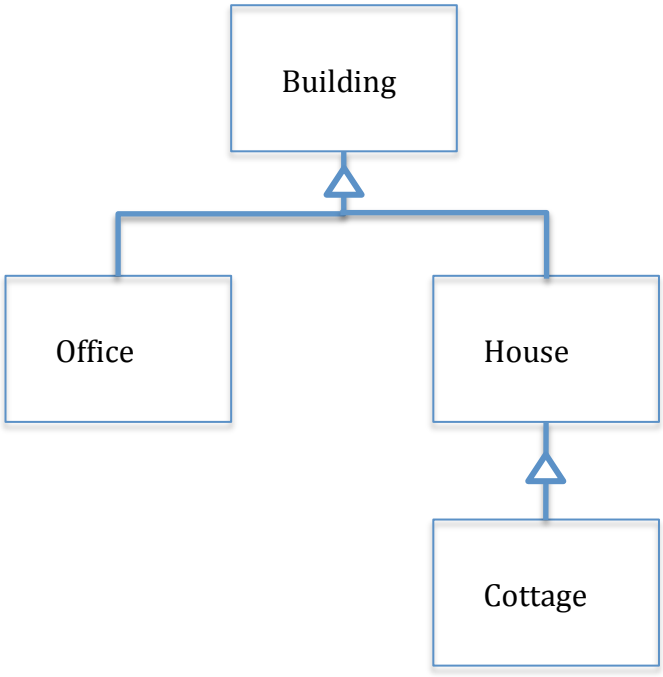
Module 7: Creating classes; relying on inheritance and polymorphism

For this exercise you will be creating a hierarchy consisting of several class files. The abbreviated diagram below will give you an idea of what you will produce. Once these classes have been created, you should test them by creating simple test files that instantiate (or create) your object and use some of its behaviors (methods). Then move on to create the client program described further in this document.

Learning outcomes:

When you have completed this exercise you will be able to

- Create classes that are derived from 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 testing a class



Introduction:

One of the many benefits of designing with classes is the ability to create classes that inherit much of their state and behavior from previously designed classes. This can save time, makes debugging and refactoring easier, and takes advantage of the object oriented principle of polymorphism as described in some of this module's video lessons. This assignment will focus on building four classes that demonstrate the relationships between a super class and its derived classes.

Resources:

Along with this specification file, you are provided with a starter project. Follow the attached instruction to download necessary files and upload your submission.

In order for evaluators to do their job 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 will do

Turn in: **Bulding.java**

Create each of the classes as described in the detailed UML diagrams below. Recall that a (-) before a state or behavior indicates that it should be private; (+) indicates public. Adhere to the adage "don't repeat yourself" which translates to writing a minimal amount of code by depending on inheritance and the use of methods that have already been written elsewhere in your files. When you have completed the class files, it is recommended that you create a run a simple test file that uses the Building object to be sure you have constructed robust a class file.

Building
- length : int - width : int - lotLength : int - lotWidth : 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 set methods for Building are public. In the "real world", when a building is created the size of the structure and the lot is set and should not be altered, so making those set methods private is probably more realistic. But for this course, the auto-grader will attempt to call your set methods and thus, they need to 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 called on setLotLength and setLotWidth and would include many checks. That circumstance is not included in this exercises however.
- Use set methods when creating your constructors
- Create the `toString()` method. This method simply returns a string representation of the `Building` object (see first line of example output below).

Turn in: **House.java**

House extends Building
- owner : String - pool: 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:

- `pool` indicates if the house has a pool or not
- The first constructor leaves owner as null and pool as false. Use `super` to take advantage of code already written. The second sets the owner, the third the owner and pool status.
- Override the `toString()` method of the `Building` class by defining your own `toString()` method in the `House` class. Formatting specifics can be seen in the sample output. Use `calcBuildingArea()` and `calcLotArea()` to determine if the lot "`has a big open space`" because building area < lot area, or not.
- Override the default `equals()` method. Two buildings are equal if their building areas are equal and their pool status is the same.

Turn in: **Cottage.java**

```
Cottage extends House
- secondFloor: 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:

- `secondFloor` 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 specifics can be seen in the sample output.

Turn in: **Office.java**

```
Office extends Building
- businessName : String
- parkingSpaces: int
- totalOffices: 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:

- `totalOffices` is a private static variable. It should be initialized to 0
- The first constructor sets the building and the lot size, leaving the `businessName` null and `parkingSpaces` 0. Increment the `totalOffices` static variable by 1.
- The other constructors also increment the `totalOffices` 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 specifics can be seen in the sample output. If business name is null, output "`unoccupied`".
- Override 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 program called `Neighborhood.java` . In this program you will use `BuildingList` class that is provided with this assignment. Create an object of this type and call the `getHouses()` method to populate an array of `House` and `Cottage` objects and the `getOffices()` method to populate an array of `Office` objects. We will modify the content of the arrays created in the `BuildingList` class during the grading of this exercise, but running this program will give you a good idea about the correctness of your class files. With these arrays, print out each object and determine the overall size of the neighborhood (in terms of lot area). To print the array content, you could write a method that is passed both arrays and will print the `House` and `Cottage` objects and then the `Office` objects. But there is a better way, using polymorphism.

Create a method called `print()` that takes as an input parameter an array of `Building` objects. Also, pass a string to print out a header for the objects. Since the `toString()` method is the only method you will need for these objects , and it is inherited by the `House, Cottage` and `Office` classes from the `Building` class (and actually overridden) , this will work. Therefore, you will never have 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. You will notice that the `House` array can actually contain both `House` and `Cottage` objects because one is derived from the other. A `Cottage` (from the derived class) can act as a `House` (which is the super class). Call the `print()` method twice, first passing the array of `House` objects (and its header) and then another call passing the array of `Office` objects.

Relying on polymorphism as you did to print the objects, create another method called `calcArea()` that accepts an array of `Building` objects and returns the total (lot) area of the objects in the array. Again, the `calcLotArea ()` method is the only method you will need, and it is inherited by all of the classes derived from the `Building` class. Note : this is not an interactive program; it should not prompt the user for any input as it executes. This diagram can help you to organize the Neighborhood class:

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