Object-oriented study guide

Here is an outline of my notes on object-oriented programming…

1. Highlights of object-oriented thinking – when solving a problem, we look for nouns before looking for verbs. Usually a problem is a story about one or more big nouns, which become our classes. Other nouns in the problem may be objects of this class, attributes of the class, or simply primitive types.

   When we define a variable, we need to decide on a type. Sometimes the type we want already exists in the language or in the API. Examples are String, array and ArrayList. The idea is not to re-invent the wheel.

   However, often we need a custom-made data type, such as a playing card or a planet, and this will be a class all by itself. Don't be bashful about employing several classes within one program, but at the same time, don't go out of your way to define a class if it's going to have just one attribute and will only have one instance.

2. A benefit of object-oriented programming is that we can enforce *encapsulation*. Classes (such as a Driver) do not need to be concerned with the exact implementation or representation of objects that it uses. This is why attributes are declared private. We don't want code outside the class to mess around with the internal representation of our objects.

   For example, if we wanted to simulate a Bingo game, we could use 3 classes – a Driver or Game class, a Bingo Bag to specify the numbers 1-75, and a Bingo Card to specify the 5x5 array the user will play with. The driver only cares about the names of methods and how to call them. In real life, classes may be written by different people, so the only information you may have is some interface – a file which lists methods to be implemented by the class, their parameters and return types.

3. If you are working on a large project that will involve many (say, dozens of) classes, then it may be worthwhile to incorporate levels of abstraction in your design – this means have something that looks like a bureaucracy among the classes. For example, in designing a house, there are many different entities to consider, such as the laundry room, fence, light fixtures, etc. Maybe we should organize these classes according to location or function first.

4. *Aggregation* is possible relationship that can exist between two classes. For example, a House may have several Rooms. A Fleet may have many Ships. It's simple to create this relationship. The larger class simply includes as one of its attributes some collection of objects of the smaller class. A Hotel class can have an array of Room objects as one of its attributes. A team has an array of Players. A College has an ArrayList of Students.

   Suppose you wanted to calculate the cost of tiling and carpeting a House in the object-oriented way:
   - A Driver class would create a House object. Then, call a method of the House class that computes the total cost of tiling and carpeting the house.
   - The House class contains a collection of Rooms. For each room, we can invoke a room.findCost() method and add up all the results.
   - The Room class would have a formula to determine its own cost to be tiled or carpeted.

5. A Java *interface* can help you design a class. Creating an interface is completely optional. Even though an interface is specified in its own .java file, an interface is not a class! An interface is a "to-do" list of operations that we'd like to implement in a class. It forces you to actually do everything on the list, because the program won't compile until the class specifies an implementation for each method listed in the interface. Another nice feature about an interface is that it makes it convenient to think of alternate implementations. For example, you might want to experiment between an array and ArrayList representations of the same object.

   Because classes and interfaces are both written as Java files, I recommend using the word "Interface" in the name of your interface to make it clear that it is not a class. For example, ShapeInterface.java.

   As another example, to do encryption, we would need tospecify 2 functions – encrypt() and decrypt(). Then, we can implement many forms of encryption. Each one will specify the details of how these two functions work. For example, a Caesar cipher and an exclusive-or cipher.

   Think of a "shape" interface, such as a 2-D geometrical shape. We may want to have the ability to find the area and perimeter. Different actual classes like Rectangle and Circle would implement these required methods, but in ways unique to their own nature.

6. *Inheritance* is a technique we use to help us create more classes out of existing ones. There are 2 reasons why we may want to use inheritance.

   - To improve on some existing class, add some functionality (you like a class, but wish it could do more). For example, we like the existing Random class, because it generates random numbers. But if we also want to generate random letters and words, we can create a new class such as MyRandom, where we would add this functionality. We improvise.

   - To make a class that is a more specific/distinct version of an existing class (such as you would find in biology or in some hierarchy). Books and Magazines are special kinds of Publications. Fish, Reptiles and Birds are special kinds of Animals. And after creating a Player class, we can make more specialized classes like FootballPlayer.

   When class B inherits attributes and operations from a more general class A, we say that class B *extends* A. And `extends` is a keyword in Java. A is called the *superclass*, and B is the *subclass*. Here is some alternative terminology: Sometimes we also say A is the parent class and B is the child class.

   The subclass is free to *override* anything that it has inherited from its superclass. For example, the Animal class might have an attribute canFly that is set to false, since most animals don't fly. But when we create the Bird class, which extends Animal, we would probably reset the inherited attribute canFly to true to override Animal's definition.

   And if a subclass is free to override something from the parent, then it is also free <u>not</u> to. For example, suppose Bird defines a feed() method, and Bird has a subclass called Albatross that does not implement a feed() method. What happens when we create an Albatross object and tell it to feed()? The absence of the method means we automatically go up to its superclass and look there.

So, the effect would be to enter Bird's feed() method. If the superclass in turn did not have the expected method, we continue looking upward at further superclasses (going up the chain of command) until either we find the method, or we run out of superclasses, in which case we have an error.

7. The Java keyword `super` allows a subclass to call a method in its parent class. This is most commonly done in constructors. Usually, the first statement in a subclass constructor is to call super(), which is the constructor of the superclass. See the Zoo program example.

8. Sometimes in Java we are curious what type an object is. Is x a String? Or is x a Bird? The Java language includes a special keyword `instanceof` that can answer this question. The resulting expression is of type boolean. For example, we may write `if (x instanceof Bird)` to ask if x is a Bird.

9. Java has a surprise for you. You have been using inheritance all along without even knowing it! It turns out that there is a built-in class called `Object`. It is basically the superclass of every class in Java. So, when you created any class in Java such as Mountain, and you began its declaration with "public class Mountain", there was an implicit assumption that Mountain "extends Object".

   Why does the Object class exist? There may be times when you know that you want some object, but do not yet know what its type is. For example, you might decide to have an ArrayList. But you don't know what the base type should be. As a placeholder, you can say ArrayList<Object>. In fact, by default, if you define an ArrayList and do not specifiy the base type, Object is the default.

   Another practical effect of the Object class is that it includes two built-in methods that interest us: equals() and toString().
   - The Object equals() method returns true if the two objects have the same address in memory. In other words, if they are aliases of each other. This is unlikely to be the case! In general, you should expect equals() to return false, unless you override equals() yourself when you create a class.
   - The Object toString() method returns a string that identifies the hexadecimal address of the memory location of the object. Not useful information, so once again, it's definitely a method we would like to override. So, if you call an object's toString() method and you see hexadecimal jibberish, this probably means you forgot to implement your own toString()!

10. The essence of *polymorphism* is that a variable can have more than one type, thanks to inheritance (or, less often, an interface). The type of the variable is determined at run-time, and this will determine exactly which version of a method to employ. For example:

```
Animal pet;
pet = new Bird();
pet.feed();
pet = new Fish();
pet.feed();
```

    Here, the Bird and Fish classes can each have a specialized feed() method that overrides the more general one in the Animal class.

11. In software engineering, much more time is spent on making sure the *design* is correct before moving on to the implementation. Historically, there have been two major ways to approach software engineering. They are the waterfall and the spiral philosophies. The *waterfall* model says design everything first before doing any implementation. The *spiral* model is more like coats of paint: Begin with a simple design, then implement it. Go back to the design and add the next feature, implement that, and so on.

    The waterfall model states that we have definite phases of production from analysis of the problem, to design, implementation, testing and maintenance. However, on large projects, it's unrealistic to expect to design everything before typing the first line of code. Thus, today more often people will adopt the spiral technique: Typically, there is a cycle of design and implementation as each version of the software is an enhancement of the previous one. This procedure has different variants that have names such as "rapid prototyping", "stepwise refinement" and "agile programming". You may learn more about these in future courses.

    Goals of design – The program should be:
    correct, robust, useable, maintainable, portable (or reusable), efficient

12. In our focus on design, we pay close attention to 2 aspects:
    - What classes we want, and their relationships. The most common relationships are inheritance (is-a) and aggregation (has-a). But we should not over-do inheritance. For example, do we really want to say that a car's tire is a special kind of circle? If we wanted to take advantage of the tire's circular property, it would probably be sufficient to have just a circle attribute (or simply its radius),
    - What events the user/environment will trigger, and what states the computation will be in. Consider how your program should respond to input.

    When it's time to implement, it's usually easiest to go bottom-up – start implementing the small classes that don't depend on other classes.

13. Examples of programming design – a store, a clock, a game, an airline reservation, etc.

    Basically our procedure involved the following issues: Write out the classes, their attributes and operations. Discuss their relationships. (e.g. a customer has 2 bank accounts, a bank has multiple customers, etc.)

14. It is often highly desirable to sort a collection of objects. For example, I might want to alphabetize a list of players, or sort a list of accounts in descending order of value. Java has built-in sort methods to perform this sorting. Remember that Java's default sorting mode is ascending.

    - If a is an array of primitive values, then we would use:
      ```
      Arrays.sort(a);
      ```
    - If a is an array of objects, then we use
      ```
      Arrays.sort(a, comparatorObject);
      ```
    - If a is an ArrayList of objects, then we use
      ```
      Collections.sort(a, comparatorObject);
      ```

Notice that when we sort objects, it is necessary for us to teach Java how to compare the objects. Java is already smart enough to know how to sort numbers. But not our custom-made class types. We need to create a comparator. The purpose of a comparator is to tell Java how to compare 2 values that belong to the same class.

Here is what we need to do. For clarity, I will use a "Mountain" class as an example. Assume that the Mountain class has already been implemented.

First, create the comparator class. In this case it would be called MountainComparator.java. This class "implements Comparator". The Comparator interface contains just one method. You guessed it: compare(). It takes 2 Object parameters and returns int. A simple way to compare two objects is to return their difference. But in general, our goal is to return:

- A negative number (e.g. –1) to indicate the first value is "less than" the second,
- A positive number (e.g. +1) to indicate "greater than", and
- Zero to indicate they are equal

Second, when you are ready to sort your array or ArrayList of objects, you can write:

```
Collections.sort(a, new MountainComparator());
```

It's interesting to note that we are calling the default constructor of the comparator class, and we never wrote a constructor! This is inheritance in action again. If a method you are looking for does not exist, then look in the superclass. Here, by not writing a constructor, we rely on the Object constructor, which basically does nothing.

15. To review, can you explain these terms? For each one, what does it do or why is it useful?
    - aggregation
    - attribute
    - comparator
    - constructor
    - encapsulation
    - `extends`
    - `implements`
    - `instanceof`
    - interface
    - inheritance
    - instance method
    - `Object`
    - override
    - polymorphism
    - spiral
    - static method
    - `super`
    - waterfall