Accelerated Object Oriented Design (CS210X) Notes

Christopher Myers

June 13, 2020

Contents

1 What is OOP?

Object Oriented Programming is a way of designing software using objects. It is more a "craft" than a specific style. There are various tradeoffs in OOP, granting much freedom.

1.1 Case Study - Match.com

> "I'll be a woman today... seeking a man between 60 and 110"

When one signs up for a site like Match.com, you give a lot of information to the site about yourself. This can thought about from an OOP perspective – people are objects, they have members, etc. The same thing could be about profiles, which a person has.

There's more to this case study (like messages between them), but the point is made - real-life things can be represented as objects in a way that makes intuitive sense. An example for a simple, age- based "matching" system is below:

```
ListOfPeople lop2 = new ListOfPeople();
for (int i = 0; i < listOfPeople.length(); i++)
{
   if (seeker.age <= listOfPeople.getPerson(i).getMaxAge() &&
       seeker.age >= listOfPeople.getPerson(i).getMinAge())
       lop2.addPerson(listOfPeople.getPerson(i));
}

return lop2;
```

2 Notation

- Classes should be capitalized and camel case.
- Instance/local variables, functions, etc. should be lower case and camel case.
- Constants should be all caps with underscores for spaces.

3 Interfaces

Interfaces can be used when an entire abstract class might be too clunky or otherwise inappropriate to use. In Java, interfaces are structures that define a series of methods[1] but no implementation. Classes then can in turn implement the interface. Code can in turn assign an object to an interface type so long as the corresponding class implements the relevant interface (see "Interfaces as Types")

An interface example is below:

```
public class MyClass implements MyInterface {
   public void myFunc();
   ...
}
```

Interfaces may be more useful than inheritance in many contexts as Java does not allow multiple inheritance but it *does* allow classes to implement multiple interfaces. Better still, interfaces can "inherit" (i.e. extend) other interfaces.

[1] These can also be described as *method signatures*.

3.1 Interfaces as Types

Interfaces can be declared as variables but cannot be used to create an object, like so:

```
MyInterface obj1 = new ConcreteObject(); //Valid
MyInterface obj2 = new MyInterface();    //Invalid
```

This only works so long as ConcreteObject implements the entirety of MyInterface.

Note that when an interface function is called, dynamic dispatch is used to determine the right function to call. This is done at runtime, so there might be a small performance decrease. Additionally, if an object is stored as an interface type (e.g. MyInterface obj = new MyObject() will only let you call the functions granted my MyInterface). This part, however, is done at compile time as part of static type checking.

4 Abstract Classes

Abstract classes are when you have a class with maybe a few functions defined (maybe some that can be overridden, with virtual), but that have at least one function set to "abstract" (in addition to being marked as "abstract" themselves. Such abstract functions do not have an implementation, in much the same way an interface's functions lack an implementation. Any classes that inherit from abstract classes *must* implement any abstract functions.

Generally speaking, when trying to give two classes the same type, it's better to use an interface over an abstract class as interfaces are less restrictive than abstract classes (multiple implementation is allowed, but not multiple inheritance is not).

5 Type Safety

When using interfaces, abstract classes, etc., objects can sometimes have multiple types. For example, in the SimpleDate->AbstractDate->Date hierarchy, a SimpleDate can be treated as a SimpleDate, an AbstractDate, *and* a Date. Only the members of the declared type of an object can be used - this checking is performed at compile time.

To work around some cases of this, objects can be casted from one type to another. There are two kinds of casting, upcasting and downcasting. Upcasting is creating a new base class type and assigning it equal to some child class, while downcasting is essentially the reverse. Note that not only can you cast to a class type, you can also cast to an interface type.

6 Generics

ArrayList can be used in "type-unsafe" mode where it accepts anything of an "object" type, but this brings along its own problems (alongside all the problems with dynamic typing). For one, ArrayList in this mode will return Objects only, so downcasting is needed. Second, it brings in increased possibility of human error.

To solve this problem, generics and type parameters can be used. Simply specifiy the type of an ArrayList between angled braces and the type returned by the ArrayList has been set:

```
ArrayList<Person> people = new ArrayList<Person>();
```

Writing type parameters in is extremely easy:

```
class ArrayList<T>{
   public int size() {...}
   public void add(T newObject) {...}
}
```

7 Algorithm Analysis

Algorithms can be analyzed by how much time they take to execute and how much space they need during execution. Instead of using actual concrete values (seconds, bytes, etc), algorithm analysis is done in more abstract terms.

Efficiency is given as a function of n. Usually analysis is broken into three cases:

- Worst case – the worst case performance of the algorithm.
- Best case – the best case performance of the algorithm, or the quickest possible that the algorithm will execute.
- Average case – how the algorithm will behave *on average*. In practice the average case is often difficult to compute.

Generally speaking, the quicker the time function increases, the worse the algorithm. That means that T=1 (constant) is the most ideal case, while T=n! is probably one of the worst.

8 Hash Tables

Hash tables are a kind of associative array where a given "key" associated with some other piece of data is mapped to some memory slot somewhere. This is accomplished by using a hash algorithm on the key, allowing for any given key (even a string) to be turned into some way of identifying space in memory.

8.1 Hash collisions

Hash collisions – where two keys lead to the same memory location – can be dealt with in two principal ways

1. Chaining – at each slot in the array, store a linked list of elements.
2. Open addressing – if a slot is already occupied, try finding another slot. Repeat as necessary.

9 Nonlinear Data Structures

Nonlinear data structures don't store data in "linear" format, i.e. in almost contiguous, ordered memory. Primarily, these are trees.

Trees are defined by a set of nodes given by a root node, where every node has only one parent but can have multiple children. Cycles are prohibited - only one path may exist between any pair of nodes. A node with no children is called a leaf, while a node with at least one child is called an internal node.

The height of a tree is the maximum depth of any node in the tree. A level of the tree is any set of nodes in the tree of the same depth (where depth is just how many jumps must be made from the root node to get to the given node).

9.1 Binary Trees

Binary trees are special case of a tree, where every node on the tree may have *at most* two children. A binary tree of height h is full if every node at depth d $<$ h has two children. Binary tress are complete if every level of the tree is completely filled except possibly thelast, and the last level is (partially) filled from left to right.

Binary trees can be rebalanced by "rotating" them about a certain node determined to be the "center" of the tree. To do this, each node must now have a "balance" value associated with it, which is always set to zero for every new node added in.

9.2 Heaps

A heap is a kind of tree that offers fast access to the largest element in a collection of related objects. A max-heap is an abstract data type for storing data such that the largest element can be found fastest, while a min-heap allows fastest to the opposite.

Heaps must satisfy the heap condition, which is that the root of every sub-tree is no smaller than any node in its sub-tree. In effect, this means that to find the largest element, you need only grab the root node in the heap. The tricky part is to *maintain* this condition.

To maintain the heap condition, "bubble up" elements that are greater than their original parent until they are no longer bigger than its parent. In practice, this need only involve a single swap of relevant elements.

This process is very efficient, as it is only bounded by the height of the tree... which varies with $\log_2 n$ of the number of elements in the heap.

To remove the largest node, replace the original highest with the largest child node. Continue swapping down for all remaining children until the heap condition is restored. This process runs in $\log_2 n$ time.