Some Practice for CSE 1322:
Inheritance, Abstract classes, Interfaces and Polymorphism

Overview: These are common interview questions, so you should be able to solidly explain them.

Inheritance is when one class “absorbs” the properties of another class and extends its capabilities. You’ll hear terms of Superclass/subclass, base/derived classes, and parent/child classes. It all means the same thing. Probably the most famous example is the “Animal class” which defines basic things that all animals do (e.g. move, breath, eat, etc). We may have subclasses of Reptile and Mammal that derive from Animal. Mammals may define extra methods like “generateHeat” and additional properties like “fur type”. We can extend Mammal and create Dog, from Dog create Chihuahua and Poodle, and so on. If there are any methods that you inherit that you don’t like, you can redefine that method in the subclass. This is called **overriding** the inherited method (not to be confused with **overloading**, which is having two functions with the same name, but different parameters).

Then what is an **Abstract class** and why is it needed? Well, is there such thing as a Mammal? I would argue no. A mammal is a generic thing that has fur, generates heat, breathes eats and so on. Perhaps it’s just a giant ball of fur? So, we’d never likely want to create an instance of a Mammal like this:

Mammal m = new Mammal();

Because we never intend to actually instantiate a Mammal, we mark the class as **abstract** which means you can’t bring one to life; its purpose is to be inherited from. That is, we mark a class as abstract for design purposes when we’re thinking about class hierarchies.

So, then what is **polymorphism** and how is it related to inheritance? First, we can’t have polymorphism if we don’t have inheritance. Why? Let me explain through code. Polymorphism is when we put in an instance of a child class into a parent variable. What would that look like in code?

Mammal m = new Chihuahua();
m.move();  // moves like a Chihuahua

So our variable m is currently a Chihuahua, and when we tell it to do stuff, it behaves like a Chihuahua. Later, if we reassign m to be a different variable:

m = new Parrot();
m.move();  // moves like a Parrot

We can’t inherit from more than one class (in most languages), we have interfaces – which are nearly identical to abstract classes in that they have a bunch of abstract methods in them. Any class that “implements” that interface must override those methods – or is forced to become an abstract class. Why? Think about it. Let’s say that the move() method above is abstract (with no code). If I called that method, what would happen? It doesn’t make sense. So, if there’s an abstract method because you inherited one or implemented an interface, in general, you need to override it.
Questions:
1) Explain inheritance and give an example.
2) What's the purpose of abstract classes?
3) Design a class hierarchy in code that includes an abstract class as well as two subclasses
4) What is polymorphism and how is it different than inheritance?
5) Give an example of polymorphism in code.

Stacks and Queues
Overview: These are two basic data structures that are dynamic data structures – meaning the grow
and shrink in size (unlike an array, which is fixed). A stack is exactly what it sounds like. I always
envision the stack of trays a restaurant. Clean trays are pushed on the top and going into a giant
metal container. If I wanted the bottom tray, I have to pop all of the other trays off (because I can't
just grab the bottom tray). So, the two basic operations for stacks are push (to add a new element to
the top of the stack) and pop (to remove an element from the top of the stack). There’s also peek,
which only looks at the top element, but doesn’t remove it. We say that a stack is First-In, Last-Out
(FILO). Is that the same as Last-In, First-Out? Think about it. For code, you might have
something like:

Stack s1 = new Stack();
s1.push (4);     // stack now has a 4 on the top
s1.push (18);   // 18 on top, 4 below it
s1.push (3);    // 3 on top, 18 and 4 below
int x = s1.pop(); // x is now 3. 18 is on top and 4 is below
print (s1.peek()); // prints out 18. 18 is still on top and 4 is
below

A queue (pronounced “Q”) is similar in nature, but the order is different. It’s similar to the lines you
stand in at an amusement park. The first person to enter the line is the first to ride the ride. We
“enqueue” new data and “dequeue” the oldest data. We say that the data structure is First-In, First-
Out (or FIFO). The code would look like the following:

Queue q1 = new Queue();
q1.enqueue (4);    // queue has 4
q1.enqueue (18);  // queue has 4 at front, followed by 18
q1.enqueue (3);   // queue has 4, followed by 18 and 3
int x = q1.dequeue(); // x is 4
x = q1.dequeue();  // x is 18

In practice, both of these dynamic data structures are built in to the basic API (meaning if you
# include or import those libraries, you can access them). However, in this class, you’ll likely be
asked to build the stack or queue – usually the methods. This code is in the slides, so make sure you
review it (and pay attention in class!)

Questions:
1) Create a stack, push and pop a few numbers, then draw out what the stack looks like in
memory
2) Do the same thing for a queue using enqueue and dequeue
3) Using a basic Node class (from lecture) write the push, pop and peek methods
4) Do the same thing for queue, writing the enqueue and dequeue methods.

Recursion
Overview: Recursion is just another way to repeat something, but is actually more powerful. There are some problems that can only be solved using recursion (unless you combine loops and stacks, but we’re not going there). I’d recommend that you watch the video on recursion, because it visualizes a lot of things going on in memory.

https://www.youtube.com/watch?v=PORo1ut9kMs

A recursive function has three requirements: 1) the function calls itself, 2) it has a base case, and 3) the problem size that it’s working on gets smaller so it makes progress towards getting to the base case. I’m going to show you some simple functions that will build up to these three requirements. Let’s start with requirement #1. If we implement that by itself, you’d have:

```java
void myFunction ( ) {
    print ("Hello!");
    myFunction( );  // recursive call - the function calls itself
}
```

If we were to call this function, it would be similar to an infinite loop. However, this doesn’t go on “forever” because we actually run out of memory (which is allocated each time a function is called and remains on the “function stack”). So let’s modify the code to have base case:

```java
void myFunction (int i) {
    print ("Hello!");
    if (i == 0) {
        // here’s our base case
        return;
    } else {
        myFunction(i-1); // otherwise, recurse
    }
}
```

If we were to call this function now and pass it the value 10, what would happen? Well, it would behave the same as before because we didn’t make progress towards 0 (our base case). What would that look like?

```java
void myFunction (int i) {
    print (i + " Hello!"); // Let’s print out i also
    if (i == 0) {
        return;
    } else {
        myFunction(i-1); // note the change here
    }
}
```

Now, if we were to call myFunction (10), we’d see “10 Hello”, “9 Hello” down to “0 Hello”
Note that this function has a void return type. Where it gets a little trickier is when we use the results of the recursive calls to find the solution to a large problem. As a simple example of what this would look like, we’ll sum up the numbers between 1 and 10.

```c
int sumNums (int i) {
    if (i == 0) { // base case
        return 0;
    } else {
        return i + myFunction(i-1); // huh?
    }
}
```

If we called sumNums(0), this would be trivial. It would return a 0. What happens when you call it with 1? Well, it’s going to return 1 plus whatever myFunction (0) returns. So, we have to call that function and wait for it’s answer before we can give the overall answer.

Questions:
1) Write a function that simply calls itself and prints something. Run it and see what happens.
2) Write a function that takes in a number and recurses that many times. That is, if it takes in the number 50, it will call itself 50 times and stop.
3) Write a function that takes in a number, recurses that many times, and prints out the squares of those numbers. For example, passing it a 5 should print 1, 4, 9, 16, 25
4) **Write a function that takes in only one number (the number of times to recurse) and returns the sum of the squares of the number. You should modify the solution for question 3 to do this. So, it should print out the squares (1, 4, 9, 16, 25) and then print 55

If you can do #4, you understand the basics of recursion because you can take the solutions of the recursive call to solve a larger problem.

**Exceptions**

Overview: Things can go wrong. You could try to open a file that doesn’t exist. You could try to connect to a server using the wrong IP. You could accidently divide by 0. You could… well, you get the picture. Because of that the idea is that we wrap our code in a protective statement. That is, we “try” to do some code and then “catch” an exception (i.e. an error) when/if it occurs.

The layout of this is super easy:

```c
try {
    // code that could blow up
} catch (Exception e) {
    // variable “e” is our error and we can ask it what happened
    print (e.message);
}
```

The “e.message” code will vary depending on the language, but that’s the idea.

The flow of the code is what’s important. We try to do the code and one of two things happens:
1) The code runs cleanly and no exceptions are thrown. We skip the catch section.
2) The code doesn’t run cleanly, throws an exception and the catch statement runs. Most of the time, you try to fix whatever happened here (e.g. retry or shutdown the program). From a programmer’s standpoint, it’s best to print out whatever message is available.

That’s the basics of exception handling. Let’s move to the next level. Let’s say we have different kinds of exceptions that can be thrown: IOExceptions and DivideByZeroExceptions. We can handle that. We can also add in a statement that says “no matter if an exception was thrown or not, do this code”. That’s called the “finally” statement.

```
try {
    // code that could blow up
} catch (IOException e) {
    // run this code if an IOException was thrown
    print ("IOException thrown: " + e.message);
} catch (DivideByZeroException f) {
    print ("Divide by zero! " + f.message);
} finally {
    // do code whether an exception was thrown or not
}
```

Moving on to level 3, we can “throw” exceptions from our own functions. So for example, you may have a function that does a simple division:

```
float divide (int x, int y) {
    return x/y;
}
```

The danger here is that, if y is 0, the code blows up. So, we can modify the code so that doesn’t happen:

```
float divide (int x, int y) throws DivideByZeroException {
    if (y == 0) {
        throw (new DivideByZeroException());
    } else {
        return x/y;
    }
}
```

Notice that we never do the divide by zero, so our code doesn’t just crash… So, then to call that code:

```
void main ( ) {
    int a = 7, b = 0;
    try {
        divide (a, b); // an exception will be thrown
    } catch (DivideByZeroException e) {
        print ("You can’t divide by zero, fool!");
    }
}
```
Questions:
1) Imagine we have a function that has the potential to throw a “FunkyException”. Show how to correctly call that function.
2) Imagine we have a function that throws some kind of Exception. Show the code of how to call it, but no matter what, prints out “Thank you for playing”
3) Write a function that takes in a string. If the string is “Bob”, the function should throw an IOException.
4) Call the function above correctly.