Lecture 3: More on OOP

Review Exercises

1. Given the following code, what happens when you call factorial(−1)?

```java
public static int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

2. Write a recursive function that takes a double $x$ and a non-negative integer $n$ and returns $x^n$. What is the runtime and space utilization of your function?

3. Assume that Child extends Parent and that both have no-argument constructors. Which of the following lines will give a compile-time error?

```java
1 Child c = new Child();
2 Parent p = c;
3 c = p;
4 p = new Parent();
5 c = p;
```

Is there a way to force Java to make it compile?

4. Is it possible to construct a class that cannot be extended (i.e., the compiler will give you an error if you try)?

Review Solutions

1. If we call factorial(−1) then the program will keep recursing until we run out of stack space and the program exits (called a stack overflow).

2. Below are some implementations:

```java
public class PowerFunctions {
    public double pow(double x, int n) {
        if (n == 0) return 1;
        return x * pow(x, n - 1);
    }
    public double fasterPow(double x, int n) {
        if (n == 0) return 1;
```
if (n % 2 == 0) {
    double d = fasterPow(x,n/2);
    return d*d;
} else return x*fasterPow(x−1,n);

The runtime and space of pow is $\Theta(n)$. The runtime and space of fasterPow is $\Theta(\log n)$.

3. Lines 3 and 5 give compile-time errors since it will only allow you to assign a descendent reference to a base type reference, and not vice-versa. A set of mnemonics that sometimes help when dealing with OOP are is-a and has-a. If a class $C$ is a descendent of class $P$ then we say that a $C$ is a $P$. Consider the case where we had a Student and a CIMSSStudent. It makes sense to say a CIMSSStudent is a Student, since it is a specific kind of Student that adds on extra behavior/fields. We use has-a when describing fields. For example, we say a Position has a Stock.

In this example we can resolve the compile-time errors using casts:

```java
1 Child c = new Child(1,2,3);
2 Parent p = c;
3 c = (Child)p;
4 p = new Parent(4,5);
5 c = (Child)p;
```

The compiler will not complain now, but you will get a ClassCastException when you try to run line 5 (line 3 will run just fine). Java will let you test whether a cast will work using the instanceof keyword that we will see later.

4. We could produce this behavior by creating a class with only private constructors, but Java provides a much more flexible way allowing for arbitrary constructors. By using the final keyword we can declare a class that cannot be extended:

```java
final class Leaf {}
```

The keyword final has another usage in Java. If you put final in front of a variable you are declaring then its value cannot change. For references, final means the variable cannot be repointed. For example, if we have `final int[] arr = {1,2,3};` then the values in the array that arr points at can change, but you cannot point arr at a different array object. When people declare constants in a class (for instance `Math.PI`) they would make it a public static final variable.

### Interfaces and Abstract Classes

Unlike some other languages that support OOP, in Java a single class cannot extend more than 1 other class. To somewhat makeup for this limitation, Java allows a class to implement
any number of interfaces. But before we get to interfaces, we first describe abstract classes which will segue into interfaces. If you use the keyword abstract when declaring a class, then Java makes it impossible to make objects of that class. The purpose of an abstract is to extend it. Within an abstract class you can create methods without bodies and label them as abstract. This has the effect of forcing derived concrete (non-abstract) classes to override the abstract methods. Other than enabling abstract methods, there is no difference between abstract classes and regular classes. Here is an example:

AbstractClasses.java

```java
//abstract class InterestCalculator {
    interface InterestCalculator {
        abstract public String getName();
        abstract public double getInterest(double principal,
            double rate, int periods);
        default String wowString() { return getName(); } }

//class StandardCompounding extends InterestCalculator {
    class StandardCompounding implements InterestCalculator {
        public String getName(){ return "Standard Compounding"; }
        public double getInterest(double principal,
            double rate, int periods) {
            return principal * Math.pow(1+rate,periods);
        }
    }

class ContinuousCompounding implements InterestCalculator {
    public String getName(){ return "Continuous Compounding"; }
    public double getInterest(double principal,
        double rate, int periods) {
        return principal * Math.exp(rate*periods);
    }
}

class Account {
    private double balance;
    public Account() { balance = 0; }
    public void addMoney(double m) { balance += m; }
    public void removeMoney(double m) { balance += m; }
    public double getBalance() { return balance; }
    public void accrueInterest(double rate, int periods,
        InterestCalculator calc) {
        System.out.printf("Using a %s calculator\n",calc);
        balance = calc.getInterest(balance,rate,periods);
    }
}

public class AbstractClasses {
    public static void main(String[] args) {
```
Here InterestCalculator is an abstract class defining 2 abstract methods and 1 concrete method. Note that the concrete method toString calls getName which will be overridden in some subclass. The method accrueInterest in Account illustrates the power of polymorphism. By taking an InterestCalculator, it allows any class that extends InterestCalculator to be used. This includes the two derived classes we have written here, and any new ones we decide to write in the future. Also note that on line 34 that we cannot make any objects of type InterestCalculator.

An interface is like an abstract class, but all methods are public, and you cannot have any fields unless the fields are final and static (final means the value cannot be changed). Even more importantly, you can implement as many interfaces as you want:

```
interface A {
    void a();
    static void hmm() { System.out.println("Hmm"); }
}

interface B {
    void b();
    default void c() { System.out.println("Wow"); }
}

class Wow {}

public class Interfaces extends Wow implements A,B {
    public void a() { }
    public void b() { }
    public static void main(String[] args) {
        Interfaces i = new Interfaces();
        A a = i;
        B b = i;
        A.hmm();
    }
}
```

Here we see how to implement an interface, and that you can implement multiple interfaces. If an interface method is not labeled as default, then it is abstract (labeling as default is a Java 8 feature). An interface may also have static methods with implementations (also Java 8). Any interfaces that a class implements are considered its ancestors, so you can
Abstract Classes and Interfaces Exercises

1. What two modifiers would make no sense if they were both on a class at the same time?

Abstract Classes and Interfaces Solutions

1. It would make no sense for a class to be final and abstract. The compiler will actually give you an error if you try it.