Warm up question:

-Suppose I have 2 classes : Bar and Brew-pub

The class BrewPub extends Bar

Which of the following will compile?

```
Bar benelux = new Bar();
BrewPub dieuDuCiel = new BrewPub();
Bar troisBroiseurs = new BrewPub();
BrewPub threeBrewers = new Pub();
```

Last class

-Polymorphism
-Inheritance

Interfaces

Sometimes in Java, we will have 2 classes that both share a similar structure, but neither of them is clearly the parent or contained in the other.

In addition, I don't **ever** want to be able to create any base class

Interfaces

Interface will be used often when we have a *concept* that can be implemented in many different ways.

Example: Shape

There are many kinds of Shape

-Polygon
-Circle
-Concave
-Ellipse
-Quadrilateral (extends Polynomial maybe!)
-Rectangle (extends Quadrilateral maybe!)
-Square (extends Rectangle maybe!)

Every single one of these Objects would have defined on it 2 methods:

computeArea()

computerPerimeter()

Each Object would implement these 2 methods in a different way.

Additionally, some of the Objects may have extra attributes.

e.g. Triange may have computeHypotenuse()

An important thing to note is that I never want to actually create a Shape that is not one of these more specific shapes.

i.e. I don't want to do

```
Shape s = new Shape();
```

But I do want to do

```
Shape s = new Rectangle();
double area = s.computeArea();
```

This is what interfaces will do.

When I define the Shape class, I will not write public class Shape as we've been doing normally. Instead I will write:

```
public interface Shape {
	......
}
```

Inside of the class, I will simply write the method header for each attribute or behavior that should be defined on a Shape

```
public interface Shape {
	public double computeArea();
	public double 
computePerimeter();
}
```

By defining Shape as an interface, I make it so that the compiler will give me an error if I ever write

```
new Shape();
```

Implementing an interface.

To implement an interface, you just write implements after the class:

public class Rectangle implements Shape

(note: a class can implement more than 1 interface—put a comma b/w)

Since the Rectangle class is implementing the interface Shape, it must have all the methods defined in the interface Shape

```
public class Rectange implements Shape {
     double width, height;
      public double calculatePerimeter()
{... }
   public double calculateArea() {...}

}
```

Advantage of Interfaces:

The benefit of using an interface comes from the *dynamic dispatch* as well.

If I know that class *a* and *b* implement *c*, then that means *a is-a* c, and *b is-a c*.

This means we can store an object of type a or b into c.

Advantage of Interfaces:

We then know that everything that implements *c* MUST have the methods outlined in c defined.

Inheritance vs Interfaces : Similarities

-both define a *subtype* of a parent class
-both allow you to store subtypes into variables of the parent/base type
-both allow you to share structure between classes
-both allow you to add additional attributes/behaviors to the subtype

Inheritance vs Interfaces : Differences

-With an interface you can't create an instance of the base class. The base class is just a template.
-A class can implement more than 1 interface.

When do you use interfaces?

You will use an interface when you have some sort of common *tasks* that you want different classes to share, but they do not necessarily perform the tasks in the same way.

You use inheritance when one class is an extension of another class.

Inheritance vs Interfaces : Motivations

-Typically you will use inheritance when you have 2 (or more) similar classes and one class is completely contained in the other. You will also be likely to do this if the smaller class was already implemented and you don't want to change it.
-Interfaces will be used when the classes with common structure do not have a clear "parent"

Every reference type extends an Object !

-In Java, by default if you don't write anything, it is the same as writing

“extends Object”

This means that by default, several methods are defined for you:

for example

-toString()
-clone()

We have seen that System.out.println works on anything. How is this possible?

The people who wrote the method System.out.println() **overloaded** the method so that it was defined on:

int, double, float, long, boolean, char, byte, short

String

AND

Object

Now, suppose you write a class Foo

Foo by default extends the class Object

This means that it has the method

toString() defined on it.

So the people who wrote println could have written:

```
public void println(Object o) {
	println(o.toString());
}
```

What if I write the toString method in my code?

In this case, you are **overriding** the toString() method defined in Object.

By default, the toString() method just prints the address of the reference type.

Example of interfaces: AbstractList

http://download.oracle.com/javase/1.4.2/docs/api/java/util/AbstractList.html

There are many different types of “Lists”

Each of them should have features such as:

add(int index, Object element)
add(Object o)
addAll(int index, Collection c)
clear()
equals(Object o)
get(int index)
.....

What about a Collection?

A Collection is also an interface. Many classes implement this interface.

This is useful because sometimes a method may return a list, but not an ArrayList. You as a user don't care whether it returned an ArrayList or a HashSet---it still is a list

Comparable interface

A very common application involves sorting a complex class according to one of it's fields.

For example: I may have a class Vector and want to sort all the elements by their magnitude

To implement that Comparable interface, you just need to write a class with the method

```
int compareTo(Object o)
```

defined on it.

To implement that Comparable interface, you just need to write a class with the method

```
int compareTo(Object o)
```

defined on it.

```
public class Vector {
        double x, y,z;
        ...
        public int compareTo(Object o) {
                double m1 = this.computeMagnitude();
        double m2 = ((Vector)o).computeMagnitude();
                        if (m1 < m2) return -1;
                        else if (m2 < m1) return 1;
                        else return 0;
        }
}
```

Collections class

There is a class Collections which has the static method sort() defined on it. You can use it by writing

Collections.sort(any list of a type that implements comparable)

Example:2d ArrayList

-storing an ArrayList of ArrayList

-setting all the values

-adding up all the values

Example:Sorting an ArrayList using recursion

Assignment 4: Determinant

Assignment 4: Derivative

Midterms:

-If you have questions on the midterm or concerns, please contact me.