COMP-202
Inheritance and Polymorphism

Last week: Recursion

-writing a big problem in terms of a small problem
-base cases

This week: Inheritance and polymorphism

-Inheritance
-Interface
-Polymorphism
-Overloading vs Overriding

The point in these examples will be to be able to share structure between classes.

Generally, there are 2 types of relationships classes can have in Java:

“Has-a relationships”
“Is-a relationships”

Is-a relationships vs. Has-a relationships

So far, we have been working mainly with “has-a” relationships. For example:
-A *HockeyTeam* has two *Goalies*, 6 *Defenseman,* and 12 *Fowards*
-a *CSTeacher* has a *BadJokeGenerator*

Sometimes, in Java, we will want a class relationship to represent a “Is-A” relationship.

For example:
-*Rangers* is-a *HockeyTeam*

or

-Comp202Teacher is a *CSTeacher*

Motivating Example:

Suppose I have a class *Apartment*

An Apartment is defined as follows

```
public class House {
    Kitchen k;
    Bedroom[] bedrooms;
    Closet[] closets;
    Room[] otherRooms;
}
```

I have been using the class Apartment in many programs, but now I realize that I want to *extend* the class House to create a class Mansion.

A Mansion “is-a” House, so it will have all the same things, but more

```
public class Mansion {
	//first part is same as before
	Kitchen k;
	Bedroom[] bedrooms;
	Closet[] closets;
	Room[] otherRooms;

	Servant[] servants;
	Amphitheatre theatre;
	Gymnasium gym;
	GolfCourse[] golfCourse;
	SwimmingPool[] pools;
}
```

This approach is viable, but isn't ideal because the two classes House and Mansion will operate independently of each other.

This is bad for a few reasons

1)Code reuse: We have to write the same code again
2)Bug finding: If we find a bug in one, we now have to remember to fix it in the other as well
3)There is no way to link the two classes. For example, I can't create an array that stores both House and Mansion

Sometimes, it is possible to completely eliminate one class and “merge” the two. We could do this by giving all the fields of Mansion to House as well and them making them null.

However, this is a bit clumsy. We just made this class more difficult to maintain. Plus, what if we want to make other classes like Shack, Cottage, HauntedHouse, or TVCharacter. We don't want to affect our House class so much.

```
public class HauntedHouse {
        //first part is same as before
        Kitchen k;
        Bedroom[] bedrooms;
        Closet[] closets;
        Room[] otherRooms;

        Ghost g;
        Skeletons[] bones;
        Rats[] creepy;
        Cobwebs[] cobWebs;
        Dust[] dust;
}
```

```
public class House {
        Kitchen k;
        Bedroom[] bedrooms;
        Closet[] closets;
        Room[] otherRooms;
        Servant[] servants;
        Amphitheatre theatre;
        Gymnasium gym;
        GolfCourse[] golfCourse;
        SwimmingPool[] pools;
        Ghost g;
        Skeletons[] bones;
        Rats[] creepy;
        Cobwebs[] cobWebs;
        Dust[] dust;
}
```

Solution: Inheritance!

```
public class Mansion extends House {
        Servant[] servants;
        Amphitheatre theatre;
        Gymnasium gym;
        GolfCourse[] golfCourse;
        SwimmingPool[] pools;
}
```

All fields and methods that were a part of the class House are automatically now a part of the Mansion class

```
Mansion m = new Mansion();
```

m now has all the attributes of a House and a Mansion!

One major benefit we get right off the bat is we don't have to write any of the methods or attributes already defined in House again.

private vs protected

So far, we have talked mainly about the difference between private and public

Remember that when you omit a modifier before an attribute or behavior in a class, it is actually something called “package private”

There is actually a 4th description called *protected*

Normally, a private attribute/behavior can only be access from inside the same class.

This is true even if you have an inherited class.

So if I tried to access the field *bedrooms* inside of the Mansion class, I would not be able to because it is private.

protected will make it so that the attribute/behavior is invisible to all classes except those that are either:

a)part of the same package
b)inherited from the class

```
public class House {
	protected Kitchen k;
	protected Bedroom[] bedrooms;
	protected Closet[] closets;
	protected Room[] otherRooms;
}
```

Java coolness:

Because a Mansion *is a* House, you can actually store a Mansion into a House variable!

For example:

```
House h;
Mansion m = new Mansion();
h = m;
ArrayList<House> h2 = new ArrayList<House>();
h2.add(m);
```

Overloading a method:

In general in a class, you can overload a method. This means that you define a method with the same name, but a different # of input arguments or different types. For example:

System.out.println can take as input a

int, double, boolean, String, etc

Warm-up exercise:

What is the difference between an “is-a” relationship and a “has-a” relationship?

For each of the following objects, write the name of a class that could **extend** the class. Write the name of a class that the class could have

Cat

Car

Television

Stove

Warm-up exercise:

What is the difference between an “is-a” relationship and a “has-a” relationship?

For each of the following objects, write the name of a class that could **extend** the class. Write the name of a class that the class could have

Cat	-Leg, Head, Yarn
Car	-Wheel[], Engine
Television	-RemoteSensor,Screen
Stove	-Burner[]

Warm-up exercise:

What is the difference between an “is-a” relationship and a “has-a” relationship?

For each of the following objects, write the name of a class that could **extend** the class. Write the name of a class that the class could have

Cat	-Siamese,
Car	-RaceCar, FlyingCar
Television	-PlasmaTv, BigScreenTv
Stove	-GasStove, ElectricStove

Last class:

-Inheritance
-Basics of polymorphism
-Is-a relationship vs Has-a relationship
-Overloading a method vs overriding

Useful resource:

http://pages.cs.wisc.edu/~cs368-1/JavaTutorial/NOTES/Inheritance-intro.html

Overriding a method:

Overriding a method is when you have a subclass---for example a class that inherits from another class---that defines a method with the exact same arguments that does the behavior differently.

Recall that we had a base class of type House and another class HauntedHouse that extended the House class.

Now, suppose the House class has a method *talk()*

```
public void talk() {
        System.out.println("I'm a house you
dummy. I don't know how to talk!");
}
```

For a HauntedHouse, however, we want the talk() method to perform a different behavior.

```
public void talk() {
        System.out.println(“BOOOO!!”);
}
```

```
House h = new House();
HauntedHouse hh = new HauntedHouse();



h.talk(); //prints the talk from House
hh.talk(); //prints the talk from Haunted

House h2 = hh;

h2.talk() ; //what do you think it prints?
```

Remarkably, it will print

“BOOOOO”

Java knows that even though you are using a variable of type House, that in reality the object there is a HauntedHouse

Exception: You can't call attributes/behaviors that are only present in a HauntedHouse.

For example, if we defined inside HauntedHouse the behavior creak()

We could not say

```
h2.creak();
```

Exception: You can't call attributes/behaviors that are only present in a HauntedHouse.

For example, if we defined inside HauntedHouse the behavior creak()

We could not say

```
h2.creak();
```

Java has what is known as *dynamic dispatch*

What this means, is that it decides at *run-time* which method to call.

However, the *compiler* does NOT have any sort of dynamism. So you *have* to call a method that exists on the variable of the exact type (not subclasses)

Types and subtypes

We say that type *a* is a *subtype* of the type *b* if every instance of *a* is-a *b*

For example, a HauntedHouse is-a House.

Thus HauntedHouse is a subtype of House.

Note: conceptually we could think of an int as being a subtype of double (since an int is a double), but in Java doesn't connect them

Converting between subtypes and basetypes:

To go from a subtype to a base type, you can simply assign the variable. There will be an implicit cast done.

```
House h = new HauntedHouse();

//assigns a HauntedHouse to House
//which is OK since a HauntedHouse is
//a House
```

To go from a base type to a subtype is a bit more complicated:

```
HauntedHouse hh = new House();
```

will not compile.

This is a bit counter-intuitive, because a HauntedHouse has what's in a House and more.

The trick to remember is that Java only stores the **address** of the element each time.

House h = new HauntedHouse()

Doing this creates an Object somewhere in memory that stores all the fields of a HauntedHouse. It assigns the address to h.

Because every field in a House is also in a HauntedHouse, any compiler check that works on a House, will also pass on a HauntedHouse.

h.*something* is guaranteed to be defined on the new HauntedHouse we created

HauntedHouse hh = new House()

Doing this creates an Object somewhere in memory that stores all the fields of a House. It assigns the address to hh.

There are some fields in hh that are defined on a HauntedHouse but not on a House.

Suppose Java compiler allowed this: then when you write hh.creak(), the computer will go to the address stored in hh and execute the behavior “creak” which is not defined since it's a House

To go from a base type to a subtype is a bit more complicated:

```
HauntedHouse hh = new House();
```

will not compile.

This is a bit counter-intuitive, because a HauntedHouse has what's in a House and more.

If you really know that you actually have a HauntedHouse, then you can use a cast.

For example:

```
House h = new HauntedHouse();
//   HauntedHouse hh = h;
HauntedHouse hh = (HauntedHouse) h;
```

However, if you don't *actually* have a HauntedHouse, the cast will lead to a run-time error:

```
House h = new Mansion();
HauntedHouse hh = (HauntedHouse) h;
```

Run time error!

If you aren't sure, you have 2 options.

1)Use a try-catch statement to check (clumsy)
2)Add an if-statement (cleaner)

```
if (h instanceof HauntedHouse) {
   HauntedHouse hh = (HauntedHouse)h;
}
```

Calling a parent class method from an overridden method.

Sometimes, you will want to override a method, BUT also do the default behavior.

For example, you may want to do everything that the original base class method did, plus more. In this case you can call the parent method from your overridden method.

Keyword: super

```
//inside HauntedHouse

public void talk() {
	super.talk();
	System.out.println(“booooo!”);
}
```

Keyword: super in a constructor

```
//inside HauntedHouse

public void HauntedHouse() {
	super(); //HAS to be 1st statement!
	//rest of constructor
}
```

Useful resource:

http://www.easywayserver.com/blog/java-multiple-inheritance-example/

Last class

-More on polymorphism
-super
-Introduction to interfaces

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.

Assignment 3: HumanPlayer vs ComputerPlayer

On assignment3, we defined an interface Player.

All that a Player had to do was ChooseMove(); ComputerPlayer and HumanPlayer did so differently.

This made it easier than doing if statements

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.

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 as in Assignment 4?

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: 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)