Lecture 22: Inheritance/Polymorphism

- Today we're going to talk about another aspect of object-oriented programming
  - The idea of "inheritance" - related to interfaces, but kind of different
- Let's say we have a company, and we have different type of people in the company

```
                  Employee
                /    |     \
              /      |       \
    Secretary     Lawyers    Engineers
         |
         |
 Legal Secretary
```

  - Everyone is an employee, but we can divide them further into secretaries, lawyers and engineers
  - There is a variation on a secretary called a legal secretary
  - This forms a ***hierarchy*** of employees (obviously not complete for a real company, but an example)
- There might be an orientation that all new employees attend - get a 20 page booklet describing company policies (ex. vacation, retirement, insurance...)
  - By default, these policies apply to all employees
  - But the next day, they may get told additional details
  - For example, there is a special lawyer orientation the next day where lawyers are given an additional three page booklet
    - The booklet may have new procedures, or redefine others
    - Ex. told that "when applying for vacation time, use the pink form, not the yellow one you were told yesterday" --> *"overriding"*
- This is like what happens in Java with ***inheritance***
  - You express this is-a relationship with the *extends* keyword

```
class B extends A {
    ...
}
```

  - And you draw the relationship with B as a child of A in the class hierarchy
  - For example, we'd say that Lawyer *extends* Employee
  - Class B is called the ***subclass*** and class A is called the ***superclass***
  - The superclass is more generic, and the subclass is more specific
  - This is opposite to how we use the words in English
    - "Would you like a cheeseburger, or a super cheeseburger?"
    - I would assume that the super cheeseburger has lots of cheese, but what would be the *superclass* of cheeseburger? Something more generic - a regular burger with no cheese!

```
   burger
     |
     |
cheeseburger
```

- What happens if you don’t have an “extends” statement in your class definition (like for most of the classes that we have written)?
    - Java automatically makes your class extend a generic class known as Object
    - So really any hierarchy looks like this:

```
Object
  |
  |
  A
  |
  |
  B
```

    - The Object class is like Employee in our company hierarchy - all classes inherit from it, either directly (class A) or indirectly (class B)
- In our company, we had a 20-page booklet that applied to all employees
    - Similarly, any state and behavior (fields/methods) in the superclass is automatically included in the subclass
    - “extends Foo” gives a class every field/method of the Foo class
    - So every class in Java has the fields/methods of the Object class - what are they?
        - e.g. toString()
        - e.g. equals()
        - Something we’ll talk about next week called hashCode()
- A subclass can (1) add new fields/methods, and (2) redefine (“override”) methods inherited from the superclass
    - Like the 3-page booklet that lawyers are given
    - For example “use the pink form instead of the yellow” is overriding a behavior of lawyers
    - Just like we sometimes override the toString() method of a class, providing our own
- Another issue: what is a “substitute” in the inheritance world
    - Consider a different hierarchy: vehicles
    - You might have a hierarchy that looks something like this:

```
             Vehicle
             /      \
            /        \
          Car         Bike
           |
           |
         Honda
           |
           |
         Accord
           |
           |
       Accord LX
           |
           |
Accord LX package 319
```

    - The most generic description appears at the top of the hierarchy, with the most specific at

the bottom
    - Each level potentially adds more behavior
    - When can one object substitute for another?
        - The more-specialized object can substitute for the more-generic object
        - This makes logical sense: if I expect to get a generic Accord, and I get an Accord LX, I'm happy - I got something even better than what I wanted
        - But if I payed for the luxury model, and get a regular Accord, I'm going to be upset
    - Something near the bottom of the hierarchy can substitute for anything above it in the hierarchy
    - But you can't do the reverse - a more-generic object cannot substitute for a more-specific object
- You also can't substitute across
    - If I was expecting an Accord, I'm not going to be happy with a Bike
    - You can't substitute across the tree - only up the tree
- We think of these different levels as "roles"
    - An object can fill multiple roles - an Accord LX package 319 can fill the role of an Accord LX, or an Accord, or a Honda, or a Car, or a Vehicle
    - We say an Accord LX package 319 *is-a* Car
    - Similarly, the legal secretary *is-a* secretary, and can fill the role of a normal secretary even though they have more specialized behaviors
- In Java, we often see things like this:

```
A x = new A();
B y = new B();
```

    - But consider this code:

```
A x = new B();
B y = new A();
```

    - One of these is valid, and one is not
    - Remember that class B extends A, so we say that B *is-an* A, and a B can fill the role of an A.
    - So the first line is OK - we're creating a B, and saying that we're going to treat it like an A (like we have an Accord, and saying that we're going to treat it like a car)
    - But the second line is not OK - a generic A cannot fill the role of a more-specific B (like we can't treat a regular Accord as an Accord LX, because it may not have all the capabilities of the luxury model)
- Another concept: "renegotiating"
    - Say that the original company we have was a temp agency
    - Clients can hire secretaries for $10/hour and legal secretaries for $15/hour
    - A client asks for a secretary, but all the secretaries are taken that day. However, there is a legal secretary who wouldn't otherwise have work, so you send the legal secretary
    - This works because a legal secretary can do everything a regular secretary can do, and more! (substitution)
    - But we'll only pay them $10/hour because they're not doing specialized work
    - During coffee break, the client discovers that the secretary is actually a legal secretary and

        says “Great! I have some legal work for you to do!”
    - Is this ok?
        - No, because the client is only paying $10/hour for a generic secretary
        - Better: renegotiate the contract to pay $15/hour for legal work
- We can “renegotiate” in Java too - it’s called “casting”
    - Similar to what we had before with interfaces
    - We’ll see some examples in just a minute
- So how IS this different from interfaces?
    - When you extend, it’s more than just a certification of behaviors - you acquire implementations of all the code in the superclass
    - You can only have 1 superclass, whereas you can implement many interfaces (have many certifications)
- An example:
    - (pass out the handout)
    - You’ll have a problem like this on the final
        - The goal is to help you really think about what it means for one class to inherit from another
        - The TAs will give you some helpful strategies in section tomorrow
    - Look at the class definitions - let’s construct a hierarchy - what classes inherit from which others?

```
          One
         /   \
        /     \
      Two    Three
               |
               |
              Four
```

    - The next step to approaching these problems is to come up with a table that indicates the output for each method for each class
        - (build up the table for each class, starting at the top of the hierarchy)

	method1	method2	method3
One	One1	---	---
Two	One1	---	Two3
Three	One1	Three2 method1()	---
Four	Four1 One1	Three2 method1()	Four3

    - We see something called *“polymorphism”* when we have one method call another
        - When method2 calls method1, it will call the *most local* version of method1
        - So if I’m a Four, my method2 will call MY method1, not the one inherited from the

        Three class
        - But if we make a superclass call, then we know instantly what the result will be (e.g. method1 of the Four class) - this is not polymorphism because each class has exactly one superclass
    - this.method1() == method1()
- A strategy for figuring out what a method call will produce as output
    - First see if you pass the compiler. If there is no casting involved, the compiler looks at the <u>*type*</u> of the variable. If there is a cast, then the compiler uses that type instead. The compiler makes sure that the type involved (the <u>*role*</u>) includes the <u>*method*</u> you are calling. If not, you get a <u>compiler error</u>.
    - Next see if there is a possible <u>runtime error.</u> This happens *only when a cast is involved.* When you cast, you are saying to Java, "Trust me. The object will be of this other type." Java isn't a trusting language. By casting, you can delay the check, but the runtime system will make sure that your claim is true (that the object in question can fill the role that you are casting it into). If the *cast is inappropriate* for the actual object involved, then you get a runtime error (a ClassCastException). This is like my example of renegotiating the contract from a secretary to a legal secretary.
    - If you pass both of those tests, then you simply *execute the method in question*. Here you pay attention to what kind of object you have. The variable type and the cast don't matter at all. *All that matters is what kind of object you have*. Objects always behave in the same way no matter what the variable type is and no matter what kind of casting you've done.