Readings
This Week: Ch 9.4-9.5 and into Ch 10.1-10.8
(Ch 11.4-11.5 and into Ch 13 in old 2nd ed)
(Reminder: Readings are absolutely vital for learning this stuff!)

Labs and Tutorials
This Week: Lab #9

Survey #2
- Dr. Ben Yu’s second survey for you all is up on WebCT now.
- Completely optional…
- But, there’s a bit of extra credit for doing all three surveys by their respective deadlines!
- The deadline for this one is March 29.
- Because of the server crash, if you did this survey before the crash, you’ll have to do it again. 😓

Programming Assignment 3
- Assignment 3 is up on WebCT!
- Click on the “Assignments” icon.
- Assigned Sunday evening – sorry for delay!
- Due at NOON, April 6 (Monday), via electronic hand in.
- Tips:
  - There is some Eclipse setup. Set-up ASAP!
  - Work in pairs. Some conceptual stuff.
  - Think carefully before coding. If concepts right, the coding is much much easier.

Learning Goals
By the end of class today you will be able to…
- Declare your own interfaces.
- Create subclasses by extending classes you have written already.
- Access methods and fields in the superclass from the subclass.
Different Roles

<table>
<thead>
<tr>
<th>Role</th>
<th>USB Analogy</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer</td>
<td>Learn what USB devices can do, look for the USB interface, and plug them in.</td>
<td>Read Java API for interfaces, declare variables with interface types, call the interface methods.</td>
</tr>
<tr>
<td>Producer</td>
<td>Understand the USB interface specification and build product that meets it.</td>
<td>Understand the interface declaration and implement its methods.</td>
</tr>
<tr>
<td>Standards Committee</td>
<td>Define USB standard.</td>
<td>Design and declare a new interface type.</td>
</tr>
</tbody>
</table>

Review: Using an Interface (Consumer)
- Declare variables and parameters using interface name instead of class name, e.g.:
  ```java
  Comparable smallest = theArray[j];
  ```
  instead of
  ```java
  String smallest = theArray[j];
  ```
  or
  ```java
  BigInteger smallest = theArray[j];
  ```
- Java lets us use only the methods of the interface:
  ```java
  if ( smallest.compareTo(theArray[j]) > 0 ) …
  ```
  but not
  ```java
  if ( smallest.length() > 0 ) …
  ```

Review: Implementing an Interface (Producer)
- Define a class as usual, but claim that it implements the interface, e.g.:
  ```java
  public class TalkingDoll0 implements Comparable {
      …
  }
  ```
- Define all methods needed for interface, e.g.:
  ```java
  public int compareTo(Object x) {
      …
  }
  ```

Defining an Interface (Standards Committee)
- If you want to create your own interface type, you must decide what public methods a class must provide, in order to implement your interface.
  ```java
  public interface InterfaceName {
      … method signatures …
  }
  ```

Example: Feedable
- Let's create a new interface type, called Feedable.
- All Feedable classes must have the following two methods:
  ```java
  public String getFavoriteFood() {
      Returns the name of the object’s favorite food
  }
  public void feedFood(String food) {
      Prints a message based on the food passed in.
  }
  ```
- In the interface, leave off the “public” since they’re always public.

Multiple Interfaces
- A physical object can have multiple interfaces
  E.g., a digital camera with a USB port, a Firewire port, an S-video output, a CompactFlash slot, etc.
- A Java object can implement multiple interfaces
- Let’s make TalkingDoll0 implement Feedable
Polymorphism

- Polymorphism is just a fancy word for the idea that the same program text will work for different kinds of objects, adapting to whatever object it's called on.
- The same call `x.feedFood("dog food")` can refer to 3 different methods.
- How does Java know which one to call?

Objects Know Their Class

- How did Java know which `feedFood` method to call?
- Java objects know what class they belong to.

Objects Know Their Class

- At compile time, Java only lets you use method/field names based on type of variable holding reference.
  - Can use typecast to assign to different variable.
- At run time, Java figures out which method to call (or field to access) based on the actual type of the object, not the variable holding the reference.

instanceof

- How did Java know which `feedFood` method to call?
- Java objects know what class they belong to.
- You can check which class an object belongs to using the `instanceof` operator:

  ```java
  if (x instanceof Dog) 
      x.feedFood("dog food");
  ```

Questions?

New Concept: Inheritance

- Inheritance in real life: stuff you get from your parents, e.g.:
  - Harry Potter inherited his father's hair and his mother's eyes.
  - I inherited my nearsightedness from both of my parents.
  - He inherited his mother's sense of humor.
- (Think of inheriting characteristics and behaviors, not money and real estate.)
Inheritance in Java

- You can declare a new class as an extension of an existing one.
- The new class automatically inherits all the instance fields and methods of the old class.
- The new class can add/change fields and methods.

```java
public class ChildClass extends ParentClass {
    ...
    put any additional fields and methods here
    ...
}
```

Silly Inheritance Example

- Let's make UBCStudent extend TalkingDoll…

Inheritance Terminology

- The child class extends the parent class.
- The parent class is called the superclass.
- The child class is called the subclass.
- The child class (subclass) inherits stuff from its parent (superclass), so it has more stuff (fields and methods).
- Mnemonic: your parents are your superiors.

In our silly example, which class was the parent? Which was the child?
Inheritance Terminology

- The child class extends the parent class.
- The parent class is called the superclass.
- The child class is called the subclass.
- The child class (subclass) inherits stuff from its parent (superclass), so it has more stuff (fields and methods).
- Mnemonic: your parents are your superiors.

What was a method that the subclass inherited from the superclass?

Inheritance Terminology

- The child class extends the parent class.
- The parent class is called the superclass.
- The child class is called the subclass.
- The child class (subclass) inherits stuff from its parent (superclass), so it has more stuff (fields and methods).
- Mnemonic: your parents are your superiors.

The subclass has more stuff than the superclass. Can you give an example?

Silly Inheritance Example

- Let's make UBCStudent extend TalkingDoll…

- Hey, UBCStudent and TalkingDoll both declare a private instance field called name. Let's eliminate the duplicate code…

Accessing the Superclass

- Even though the subclass has all the instance fields and methods of the superclass, Java still thinks of it as a separate class. So, the subclass can't directly access anything private!

Accessing the Superclass

- Even though the subclass has all the instance fields and methods of the superclass, Java still thinks of it as a separate class. So, the subclass can't directly access anything private!

- What do you do?
  - Use the public accessor/mutator methods.
  - Have the superclass make things public.
  - (Have the superclass make things protected.)

Overriding

- You can declare a new class as an extension of an existing one.
- The new class automatically inherits all the instance fields and methods of the old class.
- The new class can add/change fields and methods.

```java
public class ChildClass extends ParentClass {
    anything new here is added to the ChildClass

    anything with same signature as in ParentClass overrides the ParentClass
}
```
Recap: Variable Scope and Shadowing

- Scope tells you which declarations you can see from which points in the program.
- The scope of a variable is the places in the program where the variable can be accessed.
- Or you can think of scope as the places where you can see some variable.

Shadowing and Overriding

- These are basically the same concept:
  - If you have two declarations of the same signature (name and parameters), the closer declaration wins.

Shadowing

```java
public class foo {
    String name;
    public void setName(String name) {
        // "name" is shadowed!
        this.name = name;
    }
}
```

Overriding

```java
public class bar {
    public void a(double b) {...}
}
public class foo extends bar {
    public void a(double x) {...} // override?
}
```
**super and this**

- The keyword **super** works just like the keyword **this** to get around the shadowing.
  - **this** means the current object (implicit parameter)
  - **super** means the superclass object.

**Using super to Avoid Overriding**

- These are basically the same concept:
  - If you have two declarations of the same signature (name and parameters), the closer declaration wins.
- **Overriding**
  ```java
  public class bar {
    public void a(double b) {...}
  }
  public class foo extends bar {
    public void a(double b) { // override!
      super.a(3.14); // bypasses override
    }
  }
  ```

**Special Case: Superclass Constructor**

- Recall the special use of **this** in constructors?
  - **this** as first line of constructor calls a different constructor for the same object
  - E.g., UBCStudent class

**Special Case: Superclass Constructor**

- Recall the special use of **this** in constructors?
  - **this** as first line of constructor calls a different constructor for the same object
  - E.g., UBCStudent class
- Similar trick to call a superclass constructor using the **super** keyword:
  - **super** call must be first line of constructor

**How Child Objects Are Constructed**

- The child object contains everything the superclass object does, so the superclass constructor needs to get called.
- Java automatically calls the superclass’s constructor, before the child’s constructor starts working.
  - The parameters it passes to the superclass constructor are the same as were passed to the subclass constructor.
- The special case of calling **super** on the first line tells Java to call the superclass constructor in the way you specify, instead of automatically with the same arguments.