**News**

- labs last week
  - still time to work through lab 7 (midterm correction)
  - can earn back up to 5 out of 70 points
- rest of Assignment 2 handed back at end of class
- Assignment 3 posted
  - due Friday Apr 7, 5pm
  - start now, don't wait!

**Reading**

- This week: Chap 13, except 13.8.3

**Recap: Polymorphism**

- reference to interface type can reference instance of *any* class implementing that interface
  - static type: type that variable declared to be
t    - determines which members of class can be invoked
  - dynamic type: type that variable actually references
    - determines which version of method is called

**Correction/Recap: Interfaces as Contract**

- Can write code that works on anything that fulfills contract
  - even classes that don’t exist yet!
- Example: Comparable
  - useful if you need to sort items
  - `compareTo(object)`
    - returns `< 0` if this object less than parameter
    - returns `0` if same
    - returns `> 0` if this object greater than parameter

**Recap: Wrappers**

- Many classes implement Comparable interface
  - Byte, Character, Double, Float, Integer, Long, Short, String
  - each implements own version of `compareTo`
- Wrapper classes
  - wraps up (encapsulates) primitive type
  - Double: object wrapping primitive double
    - No: `sort( double[] myData );`
    - Yes: `sort( Double[] myData );`
### Comparable
- sort method that works on array of objects of any type that implements Comparable
- type guaranteed to have compareTo method
- we need to sort
  - int
  - String
  - Bunny
  - Giraffe
  - ...

### Multiple Interfaces
- Classes can implement more than one interface at once
- contract to implement all abstract methods defined in every interface it implements

```java
public class MyClass implements Interface1, Interface2, Interface3 {
    ...
}
```

### Objectives
- Understanding inheritance
- and class hierarchies
- Understanding method overriding
- and difference with method overloading
- Understanding when and how to use abstract classes

### Vending Science Marches On...
- CokeMachine2 class had limited functionality
- buyCoke()
  - what if run out of cans?
- Let’s build the Next Generation
  - just like old ones, but add new exciting loadCoke() functionality
- How do we create CokeMachine2000

### One Way: Copy CM2, Change Name, ...
```java
public class CokeMachine2000 {
    private static int totalMachines = 0;
    private int numberOfCans;
    public CokeMachine2000() {
        System.out.println("Adding another machine to your empire with "+ numberOfCans + " cans of Coke");
        totalMachines++;
    }
    public CokeMachine2000(int n) {
        numberOfCans = n;
        System.out.println("Adding another machine to your empire with "+ numberOfCans + " cans of Coke");
        totalMachines++;
    }
    public static int getTotalMachines() { return totalMachines; }
    public int getNumberOfCans() { return numberOfCans; }
    public void buyCoke() {
        if (numberOfCans > 0) {
            numberOfCans --;
            System.out.println("Have a Coke");
        } else {
            System.out.println("Sold Out");
        }
    }
    public class MyClass implements Interface1, Interface2, Interface3 {
        ...
    }
}
```
public void loadCoke(int n)
    {    numberOfCans = numberOfCans + n;
    System.out.println("Adding "+ n " cans to this machine");
    }

...Then Add New Method

Update The SimCoke Program

public class SimCoke2000
{ public static void main (String[] args)
}
}

It Works!

> java SimCoke2000 Coke machine simulator Adding another machine to your empire with 10 cans of Coke Adding another machine to your empire with 237 cans of Coke ADDING ANOTHER... 9 cans remaining Have a Coke 236 cans remaining Have a Coke 6 cans remaining Adding 150 cans to this machine Have a Coke 149 cans remaining

Is There An Easier Way...

...to create a new and improved CokeMachine class from the old CokeMachine class without copying all the code?

No. OK, I lied. There is an easier way. I'm just checking to see if you're awake.

Here's how easy it is. We use the reserved word extends like this...

Is There An Easier Way...

...to create a new and improved CokeMachine class from the old CokeMachine class without copying all the code?

No.
**Testing With SimCoke**

```java
public class SimCoke2000 extends CokeMachine2
{
    public SimCoke2000()
    {
        super();
    }
}
```

<table>
<thead>
<tr>
<th>Location: class SimCoke2000</th>
</tr>
</thead>
</table>

**Easier Way (First Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public CokeMachine2000()
    {
        super();
    }
    Error: numberOfCans has private access in CokeMachine2
}
```

**Easier Way (Second Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public CokeMachine2000()
    {
        super();
    }
    Error: numberOfCans has private access in CokeMachine2
}
```

**Easier Way (Third Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public CokeMachine2000()
    {
        super();
    }
    Error: numberOfCans has private access in CokeMachine2
}
```

**Easier Way (First Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public void loadCoke(int n)
    {
        numberOfCans = numberOfCans + n;
        System.out.println("Adding " + n + " cans to this machine");
    }
}
```

**Easier Way (Second Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public void loadCoke(int n)
    {
        numberOfCans = numberOfCans + n;
        System.out.println("Adding " + n + " cans to this machine");
    }
}
```

**Easier Way (Third Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public void loadCoke(int n)
    {
        numberOfCans = numberOfCans + n;
        System.out.println("Adding " + n + " cans to this machine");
    }
}
```

**Easier Way (First Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public void loadCoke(int n)
    {
        numberOfCans = numberOfCans + n;
        System.out.println("Adding " + n + " cans to this machine");
    }
}
```

**Testing Second Pass**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public CokeMachine2000()
    {
        super();
    }
    public CokeMachine2000(int n)
    {
        super(n);
    }
    public void loadCoke(int n)
    {
        numberOfCans = numberOfCans + n;
        System.out.println("Adding " + n + " cans to this machine");
    }
}
```

**Testing Second Pass**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public CokeMachine2000()
    {
        super();
    }
    public CokeMachine2000(int n)
    {
        super(n);
    }
    public void loadCoke(int n)
    {
        numberOfCans = numberOfCans + n;
        System.out.println("Adding " + n + " cans to this machine");
    }
}
```

**Testing Second Pass**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public CokeMachine2000()
    {
        super();
    }
    public CokeMachine2000(int n)
    {
        super(n);
    }
    public void loadCoke(int n)
    {
        numberOfCans = numberOfCans + n;
        System.out.println("Adding " + n + " cans to this machine");
    }
}
```

**Testing Second Pass**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public CokeMachine2000()
    {
        super();
    }
    public CokeMachine2000(int n)
    {
        super(n);
    }
    public void loadCoke(int n)
    {
        numberOfCans = numberOfCans + n;
        System.out.println("Adding " + n + " cans to this machine");
    }
}
```

<table>
<thead>
<tr>
<th>Location: class CokeMachine2000</th>
</tr>
</thead>
</table>

**Easier Way (First Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public class CokeMachine2000
    {
        public void loadCoke(int n)
        {
            numberOfCans = numberOfCans + n;
            System.out.println("Adding " + n + " cans to this machine");
        }
    }
}
```

**Easier Way (Second Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    private static int totalMachines = 0;
    private int numberOfCans;
}
```

**Easier Way (Third Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    private static int totalMachines = 0;
    private int numberOfCans;
}
```

**Easier Way (First Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public CokeMachine2000()
    {
        super();
    }
    Error: numberOfCans has private access in CokeMachine2
}
```

**Easier Way (Second Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public CokeMachine2000()
    {
        super();
    }
    Error: numberOfCans has private access in CokeMachine2
}
```

**Easier Way (Third Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    private static int totalMachines = 0;
    private int numberOfCans;
}
```

**Easier Way (First Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public class CokeMachine2000
    {
        public void loadCoke(int n)
        {
            numberOfCans = numberOfCans + n;
            System.out.println("Adding " + n + " cans to this machine");
        }
    }
}
```

**Easier Way (Second Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    public class CokeMachine2000
    {
        public void loadCoke(int n)
        {
            numberOfCans = numberOfCans + n;
            System.out.println("Adding " + n + " cans to this machine");
        }
    }
}
```

**Easier Way (Third Pass)**

```java
public class CokeMachine2000 extends CokeMachine2
{
    private static int totalMachines = 0;
    private int numberOfCans;
}
```
Easier Way (Third Pass)

Simple fix: change access modifier to **protected** in superclass definition
- protected variables can be directly accessed from declaring class and any classes derived from it

```java
public class CokeMachine2000 extends CokeMachine2 {
    public CokeMachine2000() {
        super();
    }
    public CokeMachine2000(int n) {
        super(n);
    }
    public void loadCoke(int n) {
        numberOfCans += n;
        System.out.println("Adding " + n + " cans to this machine");
    }
}
```

Testing With SimCoke

```java
public class SimCoke2000 {
    public static void main(String[] args) {
        System.out.println("Coke machine simulator");
        CokeMachine2 cs = new CokeMachine2();
        CokeMachine2 engr = new CokeMachine2(237);
        CokeMachine2000 chan = new CokeMachine2000(1);
        cs.buyCoke();
        engr.buyCoke();
        chan.buyCoke();
        chan.loadCoke(150);
        chan.buyCoke();
    }
}
```

Some Coke Machine History

- **early Coke Machine**
  - mechanical
  - sealed unit, must be reloaded at factory
  - no protection against vandalism

- **Coke Machine 2000**
  - electro-mechanical
  - can be reloaded on site
  - little protection against vandalism

- **Coke Machine UA***
  - prototype cyberhuman intelligent mobile autonomous vending machine
  - can reload itself in transit
  - vandalism? bring it on

* Urban Assault
Some Coke Machine History

Coke Machine UA

Assuming that previous generation CokeMachine simulations have wimpy vandalize() methods built-in to model their gutless behavior when faced with a crowbar-wielding human, how do we create the UA class with true vandal deterrence?

Method Overriding

If child class defines method with same name and signature as method in parent class

say child's version overrides parent's version in favor of its own

```java
public class CokeMachine2000 extends CokeMachine2 {
    public CokeMachine2000() {
        super();
    }

    public void vandalize() {
        System.out.println("Stop it! Never mind, here's my money");
    }
}
```

```java
public class CokeMachineUA extends CokeMachine2000 {
    public CokeMachineUA() {
        super();
    }

    public void vandalize() {
        System.out.println("Eat lead and die, you slimy Pepsi drinker!!");
    }
}
```

```java
public class CokeMachine2 {
    private static int totalMachines = 0;
    protected int numberOfCans;

    public void loadCoke(int n) {
        numberOfCans = n;
        System.out.println("Adding another machine to your empire with "+ numberOfCans + " cans of Coke");
        totalMachines++;
    }

    public int getNumberOfCans() {
        return numberOfCans;
    }

    public void buyCoke() {
        if (numberOfCans > 0) {
            numberOfCans = numberOfCans - 1;
            System.out.println("Have a Coke");
            System.out.println("cans remaining");
        } else {
            System.out.println("Sold Out");
        }
    }

    public void vandalize() {
        System.out.println("Please don't hurt me...take all my money");
    }
}
```

```java
public class CokeMachine2000 extends CokeMachine2 {
    public CokeMachine2000() {
        super();
    }

    public void loadCoke(int n) {
        numberOfCans = numberOfCans + n;
        System.out.println("loading " + n + " cans");
    }

    public void vandalize() {
        System.out.println("Stop it! Never mind, here's my money");
    }
}
```

```java
public class CokeMachineUA extends CokeMachine2000 {
    public CokeMachineUA() {
        super();
    }

    public void vandalize() {
        System.out.println("Stop it! Never mind, here's my money");
    }
}
```
Method Overriding

```java
public class SimVend {
    public static void main (String[] args) {
        CokeMachine2[] mymachines = new CokeMachine2[5];
        mymachines[0] = new CokeMachine2();
        for (int i = 0; i < mymachines.length; i++) {
            if (mymachines[i] != null) {
                mymachines[i].vandalize();
            }
        }
    }
}
```

```java
> java SimVend
Adding another machine to your empire with 10 cans of Coke
Adding another machine to your empire with 10 cans of Coke
Adding another machine to your empire with 10 cans of Coke
Stop it! Never mind, here's my money.
Eat lead and die, you slimy Pepsi drinker!!
```

Method Overriding

- If child class defines method with same name and signature as method in parent class
  - say child's version overrides parent's version in favor of its own
    - reminder: signature is number, type, and order of parameters
- Writing our own `toString()` method for class overrides existing, inherited `toString()` method
- Where was it inherited from?

Method Overriding

- Where was it inherited from?
  - All classes that aren't explicitly extended from a named class are by default extended from `Object` class
    - `Object` class includes a `toString()` method
  - so... class header
    - public class myClass
  - is actually same as
    - public class myClass extends Object

Overriding Variables

- You can, but you shouldn't
- Possible for child class to declare variable with same name as variable inherited from parent class
  - one in child class is called shadow variable
  - confuses everyone!
- Child class already can gain access to inherited variable with same name
  - there's no good reason to declare new variable with the same name

Another View of Polymorphism

- From Just Java 2 by Peter van der Linden:
  - Polymorphism is a complicated name for a straightforward concept. It merely means using the same one name to refer to different methods. "Name reuse" would be a better term.
  - Polymorphism made possible in Java through method overloading and method overriding
  - remember method overloading?
Method Overloading and Overriding
- Method overloading: "easy" polymorphism
  - in any class can use same name for several different (but hopefully related) methods
  - methods must have different signatures so that compiler can tell which one is intended
- Method overriding: "complicated" polymorphism
  - subclass has method with same signature as a method in the superclass
  - method in derived class overrides method in superclass
  - resolved at execution time, not compilation time
  - some call it true polymorphism

A New Wrinkle
- Expand vending machine empire to include French fry machines
  - is a French fry machine a subclass of Coke Machine?

If We Have This Class Hierarchy...

...Does This Make Sense?

Does This Make More Sense?
- Want generic VendingMachine class
  - don't actually use to generate objects
  - use as template for specific actual classes like FrenchFryMachine and CokeMachine
Does This Make More Sense?

- Want generic VendingMachine class
- don't actually use to generate objects
- use as template for specific actual classes like FrenchFryMachine and CokeMachine
- One way: make a VendingMachine interface like last week
- Another way...

Abstract Classes

- Abstract classes serve as place holders in class hierarchy
- Abstract class typically used as partial description inherited by all its descendants
- Description insufficient to be useful by itself
  - cannot instantiated if defined properly
- Descendent classes supply additional information so that instantiation is meaningful
  - abstract class is generic concept in class hierarchy
  - class becomes abstract by including the `abstract` modifier in class header

```java
public abstract class GenericVendingMachine
{
    private int numberOfItems;
    private double cashIn;
    
    public GenericVendingMachine()
    {
        numberOfItems = 0;
    }

    public boolean vendItem()
    {
        boolean result;
        if (numberOfItems > 0)
        {
            numberOfItems--;
            result = true;
        }
        else
        {
            result = false;
        }
        return result;
    }
}
```

```java
public class CokeMachine3 extends VendingMachine
{
    public CokeMachine3()
    {
        super();
    }

    public CokeMachine3(int n)
    {
        super();
        this.loadItems(n);
    }

    public void buyCoke()
    {
        if (this.vendItem())
        {
            System.out.println("Have a nice frosty Coca-Cola!");
            System.out.println(this.getNumberOfItems() + " cans of Coke remaining");
        }
        else
        {
            System.out.println("Sorry, sold out");
        }
    }
}
```

```java
public void loadCoke(int n)
{
    this.loadItems(this.getNumberOfItems() + n);
    System.out.println("Adding " + n + " ice cold cans of Coke to this machine");
}
```