Recap: Parameter Passing
Consider the following program:

```java
class Ptest {
    public static void main(String[] args) {
        int x = 4;
        int[] x = {4};
        System.out.println("main: x is now: "+x[0]);
        System.out.println("method1: x is now: "+x[0]);
    }
}
```

What's printed?
```java
main: x is now: 4
method1: x is now: 4
```

Parameter Passing
Will this program behave differently? Why or why not?
```java
class ParamTest2 {
    public static void main(String[] args) {
        int number = 4;
        System.out.println("main: number is 4");
        System.out.println("method1: number is 4");
        method1(number);
    }
}
```

What's printed?
```java
main: number is 4
method1: number is 4
```

Parameter Passing
Will this program behave differently? Why or not?
```java
class ParamTest2 {
    public static void main(String[] args) {
        int number = 4;
        System.out.println("main: number is 4");
        System.out.println("method1: number is 4");
        number = number * number;
        System.out.println("method1: number is now "+number);
    }
}
```

What's printed?
```java
main: number is 4
method1: number is 4
method1: number is now 16
```

Parameter Passing
Now consider this program.
```java
class Ptest {
    public static void main(String[] args) {
        int x = 4;
        int[] x = {4};
        System.out.println("main: x is now: "+x[0]);
        System.out.println("method1: x is now: "+x[0]);
    }
}
```

What's printed?
```java
main: x is now: 4
method1: x is now: 4
```

Parameter Passing
Now consider this program.
```java
class Ptest {
    public static void main(String[] args) {
        int x = 4;
        int[] x = {4};
        System.out.println("main: x is now: "+x[0]);
        System.out.println("method1: x is now: "+x[0]);
    }
}
```

What's printed?
```java
main: x is now: 4
method1: x is now: 4
```

Parameter Passing
Now consider this program.
```java
class Ptest {
    public static void main(String[] args) {
        int x = 4;
        int[] x = {4};
        System.out.println("main: x is now: "+x[0]);
        System.out.println("method1: x is now: "+x[0]);
    }
}
```

What's printed?
```java
main: x is now: 4
method1: x is now: 4
```

Parameter Passing
Now consider this program.
```java
class Ptest {
    public static void main(String[] args) {
        int x = 4;
        int[] x = {4};
        System.out.println("main: x is now: "+x[0]);
        System.out.println("method1: x is now: "+x[0]);
    }
}
```

What's printed?
```java
main: x is now: 4
method1: x is now: 4
```

Parameter Passing
Now consider this program.
```java
class Ptest {
    public static void main(String[] args) {
        int x = 4;
        int[] x = {4};
        System.out.println("main: x is now: "+x[0]);
        System.out.println("method1: x is now: "+x[0]);
    }
}
```

What's printed?
```java
main: x is now: 4
method1: x is now: 4
```

Parameter Passing
Now consider this program.
```java
class Ptest {
    public static void main(String[] args) {
        int x = 4;
        int[] x = {4};
        System.out.println("main: x is now: "+x[0]);
        System.out.println("method1: x is now: "+x[0]);
    }
}
```

What's printed?
```java
main: x is now: 4
method1: x is now: 4
```

Recap: Parameter Passing
Consider the following program:
```java
class Ptest {
    public static void main(String[] args) {
        int number = 4;
        int[] number = {4};
        System.out.println("main: number is now "+number);
        System.out.println("method1: number is now "+number);
    }
}
```

What's printed?
```java
main: number is now 4
method1: number is now 16
main: number is now 4
```

Parameter Passing
Now consider this program.
```java
class Ptest {
    public static void main(String[] args) {
        int number = 4;
        int[] number = {4};
        System.out.println("main: number is now "+number);
        System.out.println("method1: number is now "+number);
    }
}
```

What's printed?
```java
main: number is now 4
method1: number is now 16
```

Parameter Passing
Now consider this program.
```java
class Ptest {
    public static void main(String[] args) {
        int number = 4;
        int[] number = {4};
        System.out.println("main: number is now "+number);
        System.out.println("method1: number is now "+number);
    }
}
```

What's printed?
```java
main: number is now 4
method1: number is now 16
```

Parameter Passing
Now consider this program.
```java
class Ptest {
    public static void main(String[] args) {
        int number = 4;
        int[] number = {4};
        System.out.println("main: number is now "+number);
        System.out.println("method1: number is now "+number);
    }
}
```

What's printed?
```java
main: number is now 4
method1: number is now 16
```

Parameter Passing
Now consider this program.
```java
class Ptest {
    public static void main(String[] args) {
        int x = 4;
        int[] x = {4};
        System.out.println("main: x is now: "+x[0]);
        System.out.println("method1: x is now: "+x[0]);
    }
}
```

What's printed?
```java
main: x is now: 4
method1: x is now: 4
```
Parameter Passing
Now consider this program.

```java
class Ptest {  public static void main(String[] args)  {    int x = 4;    System.out.println("main: foo is now: "+x[0]);    // No, it's the reference, or pointer, to the object.  A copy of that reference is passed to method1 and assigned to x.  The reference in foo and the reference in x both point to the same object.  
  }  public static void method1(void)  {    // A copy of the object reference is passed to method1 and assigned to x.  What's in foo?  Is it the int[] array object?  No, it's the reference, or pointer, to the object.  A copy of that reference is passed to method1 and assigned to x.  
    int x = 16;    System.out.println("method1: x is now: "+x[0]);    // No, it's the reference, or pointer, to the object.  A copy of that reference is passed to method1 and assigned to x.  
  }  public static void method2(void)  {    // No, it's the reference, or pointer, to the object.  A copy of that reference is passed to method1 and assigned to x.  What's in foo?  Is it the int[] array object?  No, it's the reference, or pointer, to the object.  A copy of that reference is passed to method1 and assigned to x.  
    int x = 16;    System.out.println("method2: x is now: "+x[0]);    // No, it's the reference, or pointer, to the object.  A copy of that reference is passed to method1 and assigned to x.  
  }
}
```

Midterm Q4 from 04W2
```java
public class CokeMachine4 {  private int numberOfCans;   public CokeMachine4()  {    numberOfCans = 2;    System.out.println("CokeMachine4: " + numberOfCans);  }  public CokeMachine4(int numberOfCans)  {    this.numberOfCans = numberOfCans;  }  public int getNumberOfCans()  {    return numberOfCans;  }  public void setNumberOfCans(int numberOfCans)  {    this.numberOfCans = numberOfCans;  }  public void sayHowTall()  {    System.out.println("CokeMachine4 sayHowTall: "+this.numberOfCans);  }  public static void main(String[] args)  {    CokeMachine4 machine4 = new CokeMachine4();    machine4.sayHowTall();    CokeMachine4 machine4Copy = machine4;    machine4Copy.sayHowTall();  }}
```

Review: Static Fields/Methods
- Static fields belong to whole class
- Static methods belong to instantiated object
- Static methods can only use static fields
- Nonstatic methods use either nonstatic or static fields

```java
class Giraffe {  static int numGiraffes;  public int getGiraffeCount() {    return numGiraffes;  }  public void setGiraffeCount(int numGiraffes) {    this.numGiraffes = numGiraffes;  }  }  public class Giraffe2 extends Giraffe {  public static void main(String[] args)  {    Giraffe2 giraffe2 = new Giraffe2();    giraffe2.setGiraffeCount(11);    Giraffe.getGiraffeCount();    Giraffe2.getGiraffeCount();  }}
```

Variable Scope
- Scope of a variable (or constant) is that part of a program in which value of that variable can be accessed

```java
class EmailDomains {  private int numberOfEmails;  public EmailDomains()  {    System.out.println("EmailDomains: "+this.numberOfEmails);  }  public int getNumberOfEmails()  {    return numberOfEmails;  }  public void setEmailDomains(int numberOfEmails)  {    this.numberOfEmails = numberOfEmails;  }  }  public class EmailDomainsTest {  public static void main(String[] args)  {    EmailDomains emailDomains = new EmailDomains();    emailDomains.setNumberOfEmails(10);    System.out.println("EmailDomainsTest: "+emailDomains.getNumberOfEmails());  }}
```

Midterm Q4 from 04W2
```java
public class CokeMachine2 {  private int[] numberOfCans;   public CokeMachine2()  {    numberOfCans = new int[3];    numberOfCans[0] = 2;    numberOfCans[1] = 1;    numberOfCans[2] = 0;  }  public CokeMachine2(int[] numberOfCans)  {    this.numberOfCans = numberOfCans;  }  public int[] getNumberOfCans()  {    return numberOfCans;  }  public void setNumberOfCans(int[] numberOfCans)  {    this.numberOfCans = numberOfCans;  }  public void sayHowTall()  {    System.out.println("CokeMachine2 sayHowTall: "+getNumberOfCans()[0]);  }  public static void main(String[] args)  {    CokeMachine2 machine2 = new CokeMachine2();    machine2.sayHowTall();    CokeMachine2 machine2Copy = machine2;    machine2Copy.sayHowTall();  }}
```
Variable Scope

public class CokeMachine

private int numberOfCans;

public CokeMachine()

{    numberOfCans = 2;   
}

Variable Scope

public class CokeMachine2

private static int totalMachines = 0;

private int numberOfCans;

public CokeMachine2(int

{    totalMachines++;    numberOfCans = 0;   
}

Variable Scope

Objectives

■ Understanding inheritance

■ and class hierarchies

■ Understanding method overriding

■ and difference with method overloading

■ Understanding when and how to use abstract classes

Variable Scope

Reminder: CokeMachine2

Questions?

■ Static? Instance? Local? Parameters?

Objective

It

Variable Types

■ Static variables

■ declared within class

■ associated with class, not instance

■ Instance variables

■ declared within class

■ associated with instance

■ accessible throughout object, lifetime of object

■ Local variables

■ declared within method

■ accessible within method, lifetime of method

■ Parameters

■ declared in parameter list of method

■ accessible within method, lifetime of method

Variable Types

Is There An Easier Way...

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

No.

Variable Types

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...

Easier Way (First Pass)

public class CokeMachine

extends CokeMachine2

{    numberOfCans = 2;   
}

...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...

Easier Way (First Pass)

public class CokeMachine extends CokeMachine2

{    numberOfCans = 2;   
}

...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...

Easier Way (First Pass)
Easier Way (First Pass)

■ Variables and methods in CokeMachine2 class definition are included in the CokeMachine2000 definition
■ even though you can't see them
■ just because of word extends

Testing With SimCoke

public class CokeMachine2000
  public static void main (String[] args)  {
    System.out.println("Coke machine simulator");
    sim.buyCoke();
    sim.vandalize();
  }

Some Coke Machine History

early Coke Machine
• mechanical
• sealed unit, must be reload at factory
• no protection against vandalism

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

Easier Way (Second Pass)

■ Subclass (child class) inherits all methods except constructor methods from superclass (parent class)
■ Using reserved word super in subclass constructor tells Java to call appropriate constructor method of superclass
■ also makes our intentions with respect to constructors explicit

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

Method Overriding

public static void main (String[] args)  {
  System.out.println("Coke machine simulator");
  sim.buyCoke();
  sim.vandalize();
}

Method Overriding

public static void main (String[] args)  {
  System.out.println("Coke machine simulator");
  sim.buyCoke();
  sim.vandalize();
}

Method Overriding

public static void main (String[] args)  {
  System.out.println("Coke machine simulator");
  sim.buyCoke();
  sim.vandalize();
}

Method Overriding

public static void main (String[] args)  {
  System.out.println("Coke machine simulator");
  sim.buyCoke();
  sim.vandalize();
}

Method Overriding

public static void main (String[] args)  {
  System.out.println("Coke machine simulator");
  sim.buyCoke();
  sim.vandalize();
}

Method Overriding

public static void main (String[] args)  {
  System.out.println("Coke machine simulator");
  sim.buyCoke();
  sim.vandalize();
}

Method Overriding

public static void main (String[] args)  {
  System.out.println("Coke machine simulator");
  sim.buyCoke();
  sim.vandalize();
}

Method Overriding

public static void main (String[] args)  {
  System.out.println("Coke machine simulator");
  sim.buyCoke();
  sim.vandalize();
}

Testing Second Pass
Method Overriding

```java
public class CokeMachineUA extends CokeMachine2000 {
    public CokeMachineUA() {
        super();
    }
    // overrides the vandalize method from parent
    System.out.println("Eat lead and die, you slimy Pepsi drinker!!");
}
```

Overriding Variables

- You can, but you shouldn't
- Possible for child class to declare variable with same name as variable inherited from parent class
- 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

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?

A New Wrinkle

- Expand vending machine empire to include French fry machines
- Is a French fry machine a subclass of Coke Machine?

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

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
  - Class becomes abstract by including the abstract: modifier in class header
Abstract Classes

```java
public abstract class VendingMachine {
    public abstract void loadItems(int n);
    public abstract int getNumberOfItems();
}
```

```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");
}
```