University of British Columbia
CPSC 111, Intro to Computation
2009W2: Jan-Apr 2010

Tamara Munzner

Parameter/Scope Review II, Inheritance

Lecture 33, Fri Apr 9 2010

borrowing from slides by Kurt Eiselt

http://www.cs.ubc.ca/~tmm/courses/111-10

News

- final review session will be Mon Apr 26 10am-12pm, room WOOD 4
- pick up your midterm after class or in lab
  - also your first midterm if you haven't yet
  - no more corrections accepted after Thu afternoon next week, solutions released then
- cool talk 4-5:30pm today, DMP 110
  - The Funnest Job on Earth: A Presentation of Techniques and Technologies Used to Create Pixar's Animated Films (version 2.0)
  - Wayne Wooten, Pixar

News II

- don’t wait until the last minute for A3
  - due one week from today Fri 4/16
  - you get one extra day to finish up after lectures end
  - but remember there will be no DLC office hours on Fri 4/16 since classes are over

Reading

- This week is Chap 10 (Interfaces), except 10.8.3 and 10.9-10.11
  - not Chapter 11 (I/O and Error Handling) - typo!!
- Weekly due today
  - if it's on Chap 11, you'll get full credit, since announcement of typo came late, after Wed lecture
- Next week reading is 2.11-2.12, 9.5-9.8,10.9-10.10
  - 5.1-5.2, 11.5, 12.2-12.3 (2nd edition)
  - we might not get through all this material in lecture
    - in that case, minimal/no coverage on final
    - weekly reading question still due last class Wed 4/14

Recap: Parameter Passing

Consider the following program:

```
public class ParamTest1
{
  public static void main (String[] args)
  {
1   int number = 4;
2   System.out.println("main: number is " + number);
3   method1(number);
7   System.out.println("main: number is now " + number);
  }

  public static void method1(int x)
  {
4   System.out.println("method1: x is " + x);
5   x = x * x;
6   System.out.println("method1: x is now " + x);
  }
}
```

What's printed?

```
main: number is 4
method1: x is 4
method1: x is now 16
main: number is now 4
```

Recap: Parameter Passing

Consider the following program:

```
  public class ParamTest1
  {
    public static void main (String[] args)
    {
1     int number = 4;
2     System.out.println("main: number is " + number);
3     method1(number);
7     System.out.println("main: number is now " + number);
    }

    public static void method1(int x)
    {
4     System.out.println("method1: x is " + x);
5     x = x * x;
6     System.out.println("method1: x is now " + x);
    }
  }
```

Because when the value in the `int` variable `number` is passed to `method1`, what really happens is that a copy of the value (4) in `number` is assigned to the parameter `x`. It's the value in `x` that's being modified here -- a copy of the value in `number`. The original value in `number` is not affected.

Parameter Passing

Will this program behave differently? Why or why not?

```
public class ParamTest2
{
  public static void main (String[] args)
  {
    int number = 4;
    System.out.println("main: number is " + number);
    method1(number);
    System.out.println("main: number is now " + number);
  }

  public static void method1(int number)
  {
    System.out.println("method1: number is " + number);
    number = number * number;
    System.out.println("method1: number is now " + number);
  }
}
```

What's printed?

Parameter Passing

Will this program behave differently? Why or why not?

```
public class ParamTest2
{
  public static void main (String[] args)
  {
    int number = 4;
    System.out.println("main: number is " + number);
    method1(number);
    System.out.println("main: number is now " + number);
  }

  public static void method1(int number)
  {
    System.out.println("method1: number is " + number);
    number = number * number;
    System.out.println("method1: number is now " + number);
  }
}
```

What's printed?

```
main: number is 4
method1: number is 4
method1: number is now 16
?????????????????????????
```

Parameter Passing

Will this program behave differently? Why or why not?

```
public class ParamTest2
{
  public static void main (String[] args)
  {
    int number = 4;
    System.out.println("main: number is " + number);
    method1(number);
    System.out.println("main: number is now " + number);
  }

  public static void method1(int number)
  {
    System.out.println("method1: number is " + number);
    number = number * number;
    System.out.println("method1: number is now " + number);
  }
}
```

What's printed?

```
main: number is 4
method1: number is 4
method1: number is now 16
main: number is now 4
```

Parameter Passing

Will this program behave differently? Why or why not?

```
public class ParamTest2
{
  public static void main (String[] args)
  {
    int number = 4;
    System.out.println("main: number is " + number);
    method1(number);
    System.out.println("main: number is now " + number);
  }

  public static void method1(int number)
  {
    System.out.println("method1: number is " + number);
    number = number * number;
    System.out.println("method1: number is now " + number);
  }
}
```

Remember that a parameter declared in a method header has local scope, just like a variable declared within that method. As far as Java is concerned, `number` inside of `method1` is unrelated to `number` outside of `method1`. They are not the same variable.

Parameter Passing

Now consider this program.

```
public class Ptest
{
  public static void main(String[] args)
  {
    int[] foo = new int[1];
    foo[0] = 4;
    System.out.println("main: foo is now: " + foo[0]);
    method1(foo);
    System.out.println("main: foo is now: " + foo[0]);
  }

  public static void method1(int[] x)
  {
    System.out.println("method1: x is now: " + x[0]);
    x[0] = x[0] * x[0];
    System.out.println("method1: x is now: " + x[0]);
  }
}
```

What's printed?

Parameter Passing

Now consider this program.

```
public class Ptest
{
  public static void main(String[] args)
  {
    int[] foo = new int[1];
    foo[0] = 4;
    System.out.println("main: foo is now: " + foo[0]);
    method1(foo);
    System.out.println("main: foo is now: " + foo[0]);
  }

  public static void method1(int[] x)
  {
    System.out.println("method1: x is now: " + x[0]);
    x[0] = x[0] * x[0];
    System.out.println("method1: x is now: " + x[0]);
  }
}
```

What's printed?

```
main: foo is now: 4
```

Parameter Passing

Now consider this program.

```
public class Ptest
{
  public static void main(String[] args)
  {
    int[] foo = new int[1];
    foo[0] = 4;
    System.out.println("main: foo is now: " + foo[0]);
    method1(foo);
    System.out.println("main: foo is now: " + foo[0]);
  }

  public static void method1(int[] x)
  {
    System.out.println("method1: x is now: " + x[0]);
    x[0] = x[0] * x[0];
    System.out.println("method1: x is now: " + x[0]);
  }
}
```

What's printed?

```
main: foo is now: 4
method1: x is now: 4
```

Parameter Passing

Now consider this program.

```
public class Ptest
{
  public static void main(String[] args)
  {
    int[] foo = new int[1];
    foo[0] = 4;
    System.out.println("main: foo is now: " + foo[0]);
    method1(foo);
    System.out.println("main: foo is now: " + foo[0]);
  }

  public static void method1(int[] x)
  {
    System.out.println("method1: x is now: " + x[0]);
    x[0] = x[0] * x[0];
    System.out.println("method1: x is now: " + x[0]);
  }
}
```

What's printed?

```
main: foo is now: 4
method1: x is now: 4
method1: x is now: 16
```

Parameter Passing

Now consider this program.

```
public class Ptest
{
  public static void main(String[] args)
  {
    int[] foo = new int[1];
    foo[0] = 4;
    System.out.println("main: foo is now: " + foo[0]);
    method1(foo);
    System.out.println("main: foo is now: " + foo[0]);
  }

  public static void method1(int[] x)
  {
    System.out.println("method1: x is now: " + x[0]);
    x[0] = x[0] * x[0];
    System.out.println("method1: x is now: " + x[0]);
  }
}
```

What's printed?

```
main: foo is now: 4
method1: x is now: 4
method1: x is now: 16
?????????????????????
```

Parameter Passing

Now consider this program.

```
public class Ptest
{
  public static void main(String[] args)
  {
    int[] foo = new int[1];
    foo[0] = 4;
    System.out.println("main: foo is now: " + foo[0]);
    method1(foo);
    System.out.println("main: foo is now: " + foo[0]);
  }

  public static void method1(int[] x)
  {
    System.out.println("method1: x is now: " + x[0]);
    x[0] = x[0] * x[0];
    System.out.println("method1: x is now: " + x[0]);
  }
}
```

What's printed?

```
main: foo is now: 4
method1: x is now: 4
method1: x is now: 16
main: foo is now: 16
```

Parameter Passing

Now consider this program.

```
public class Ptest
{
  public static void main(String[] args)
  {
    int[] foo = new int[1];
    foo[0] = 4;
    System.out.println("main: foo is now: " + foo[0]);
    method1(foo);
    System.out.println("main: foo is now: " + foo[0]);
  }

  public static void method1(int[] x)
  {
    System.out.println("method1: x is now: " + x[0]);
    x[0] = x[0] * x[0];
    System.out.println("method1: x is now: " + x[0]);
  }
}
```

Why not 4?

```
main: foo is now: 4
method1: x is now: 4
method1: x is now: 16
main: foo is now: 16
```

Parameter Passing

Now consider this program.

```
public class Ptest
{
  public static void main(String[] args)
  {
    int[] foo = new int[1];
    foo[0] = 4;
    System.out.println("main: foo is now: " + foo[0]);
    method1(foo);
    System.out.println("main: foo is now: " + foo[0]);
  }

  public static void method1(int[] x)
  {
    System.out.println("method1: x is now: " + x[0]);
    x[0] = x[0] * x[0];
    System.out.println("method1: x is now: " + x[0]);
  }
}
```

What's in `foo`? Is it the `int[]` array object?

Parameter Passing

Now consider this program.

```
public class Ptest
{
  public static void main(String[] args)
  {
    int[] foo = new int[1];
    foo[0] = 4;
    System.out.println("main: foo is now: " + foo[0]);
    method1(foo);
    System.out.println("main: foo is now: " + foo[0]);
  }

  public static void method1(int[] x)
  {
    System.out.println("method1: x is now: " + x[0]);
    x[0] = x[0] * x[0];
    System.out.println("method1: x is now: " + x[0]);
  }
}
```

What's in `foo`? Is it the `int[]` array object? No, it's the reference, or pointer, to the object.

Parameter Passing

Now consider this program.

```
public class Ptest
{
  public static void main(String[] args)
  {
    int[] foo = new int[1];
    foo[0] = 4;
    System.out.println("main: foo is now: " + foo[0]);
    method1(foo);
    System.out.println("main: foo is now: " + foo[0]);
  }

  public static void method1(int[] x)
  {
    System.out.println("method1: x is now: " + x[0]);
    x[0] = x[0] * x[0];
    System.out.println("method1: x is now: " + x[0]);
  }
}
```

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`.

Parameter Passing

Now consider this program.

```
public class Ptest
{
  public static void main(String[] args)
  {
    int[] foo = new int[1];
    foo[0] = 4;
    System.out.println("main: foo is now: " + foo[0]);
    method1(foo);
    System.out.println("main: foo is now: " + foo[0]);
  }

  public static void method1(int[] x)
  {
    System.out.println("method1: x is now: " + x[0]);
    x[0] = x[0] * x[0];
    System.out.println("method1: x is now: " + x[0]);
  }
}
```

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`. The reference in `foo` and the reference in `x` both point to the same object.

Parameter Passing

Now consider this program.

```
public class Ptest
{
  public static void main(String[] args)
  {
    int[] foo = new int[1];
    foo[0] = 4;
    System.out.println("main: foo is now: " + foo[0]);
    method1(foo);
    System.out.println("main: foo is now: " + foo[0]);
  }

  public static void method1(int[] x)
  {
    System.out.println("method1: x is now: " + x[0]);
    x[0] = x[0] * x[0];
    System.out.println("method1: x is now: " + x[0]);
  }
}
```

When the object pointed at by `x` is updated, it's the same as updating the object pointed at by `foo`. We changed the object that was pointed at by both `x` and `foo`.

Parameter Passing

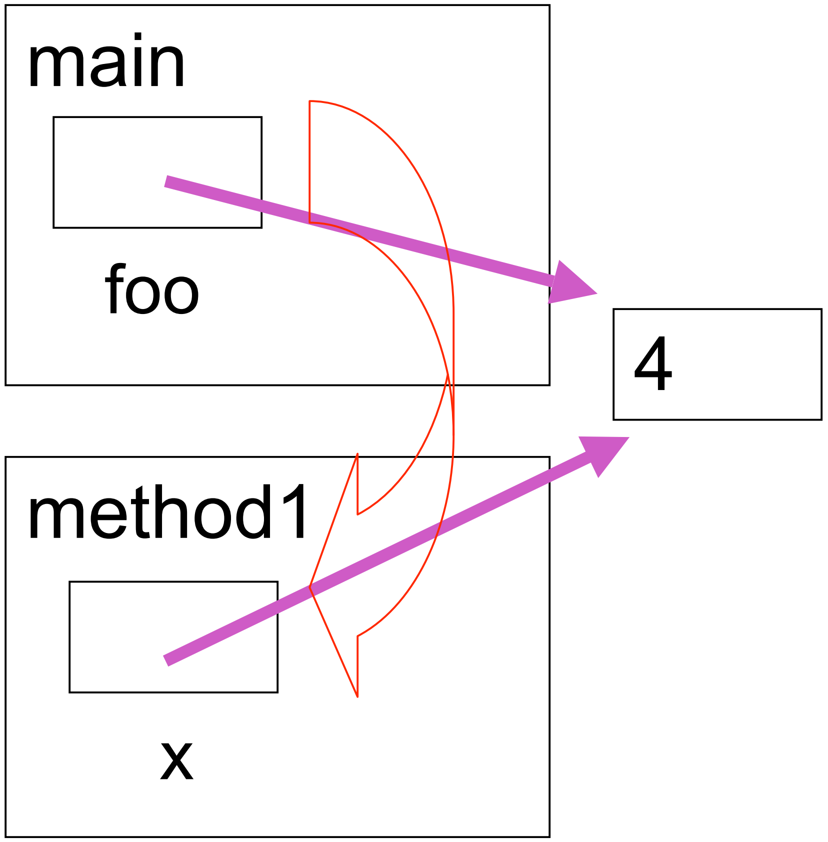
- Passing primitive types (int, double, boolean) as parameter in Java
  - "pass by value"
  - value in variable is copied
  - copy is passed to method
  - modifying copy of value inside called method has no effect on original value outside called method
    - modifying aka mutating

Parameter Passing

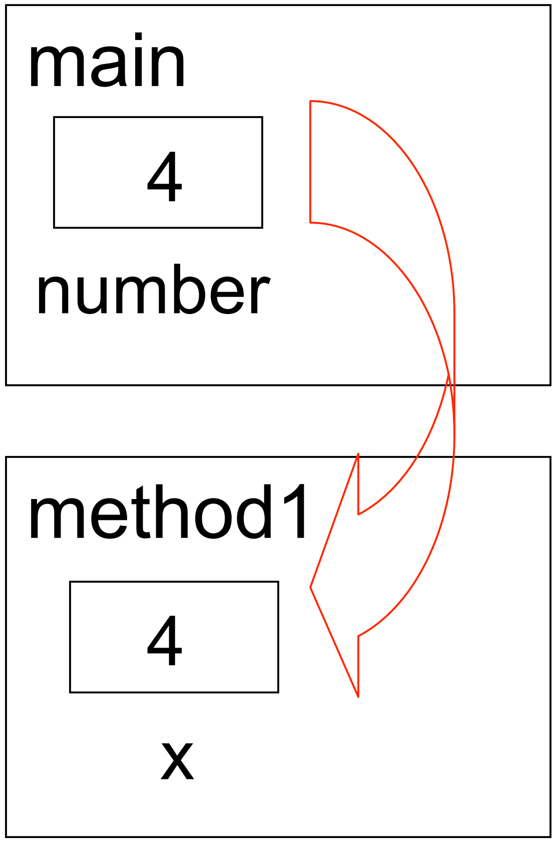
- Passing object as parameter in Java
  - "pass by reference"
  - objects could be huge, so do not pass copies around
  - pass copy of the object reference
    - object reference aka pointer
  - modifying object pointed to by reference inside calling method **does** affect object pointed to by reference outside calling method
    - both references point to **same object**

Parameter Passing Pictures

object as parameter:
copy of pointer made



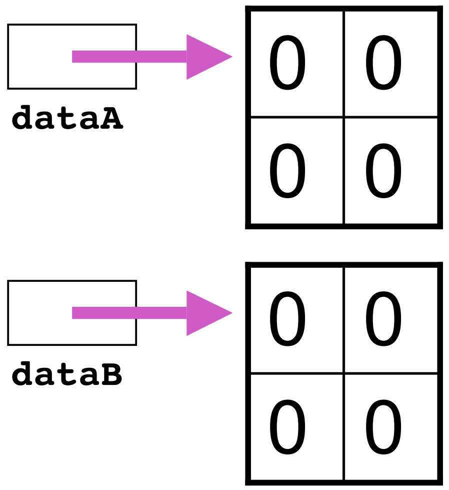
prim as parameter:
copy of value



Midterm Q4 from 04W2

```
    int[][] dataA = { { 0, 0 }, { 0, 0 } };
    int[][] dataB = { { 0, 0 }, { 0, 0 } };
    process( dataA, dataB );
```

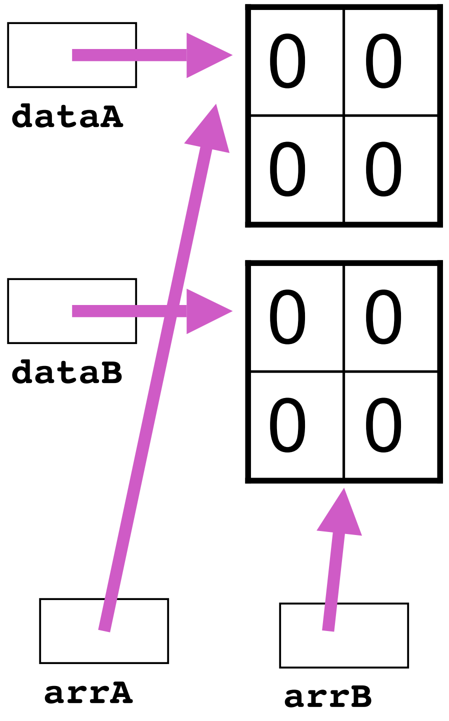
```
    public void process( int[][] arrA, int[][] arrB )
    {
        int row;
        int col;
        int[][] arrC = { { 1, 1, 1 }, { 1, 1, 1 } };
        arrA = arrC;
        for( row = 0; row < arrB.length; row++ )
        {
            for( col = 0; col < arrB[ row ].length; col++
        )
            {
                arrB[ row ][ col ] = row + col;
            }
        }
    }
```



Midterm Q4 from 04W2

```
    int[][] dataA = { { 0, 0 }, { 0, 0 } };
    int[][] dataB = { { 0, 0 }, { 0, 0 } };
    process( dataA, dataB );
```

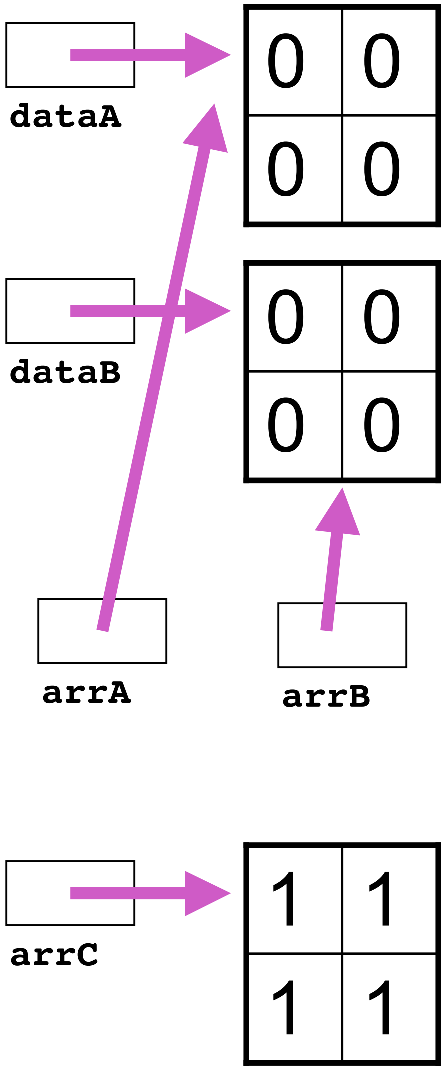
```
    public void process( int[][] arrA, int[][] arrB )
    {
        int row;
        int col;
        int[][] arrC = { { 1, 1, 1 }, { 1, 1, 1 } };
        arrA = arrC;
        for( row = 0; row < arrB.length; row++ )
        {
            for( col = 0; col < arrB[ row ].length; col++
        )
            {
                arrB[ row ][ col ] = row + col;
            }
        }
    }
```



Midterm Q4 from 04W2

```
    int[][] dataA = { { 0, 0 }, { 0, 0 } };
    int[][] dataB = { { 0, 0 }, { 0, 0 } };
    process( dataA, dataB );
```

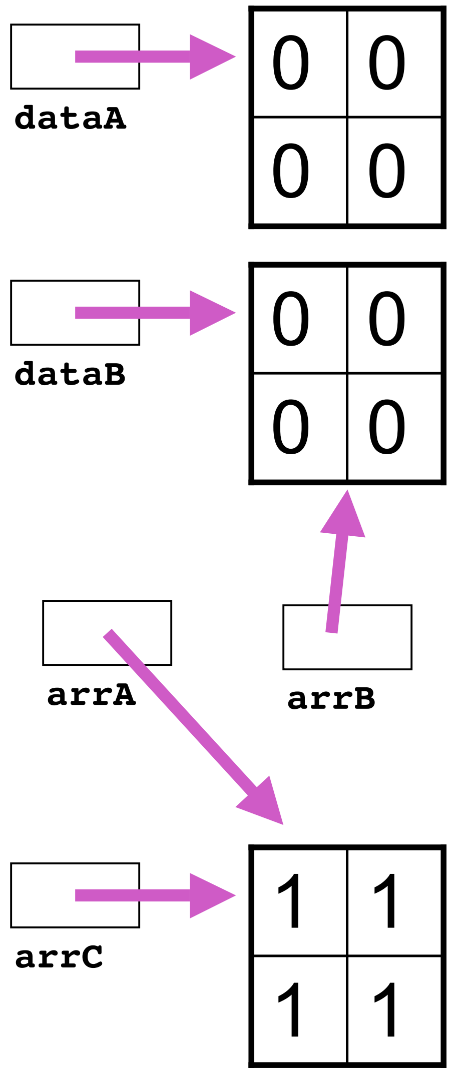
```
    public void process( int[][] arrA, int[][] arrB )
    {
       int row;
       int col;
       int[][] arrC = { { 1, 1, 1 }, { 1, 1, 1 } };
       arrA = arrC;
       for( row = 0; row < arrB.length; row++ )
       {
          for( col = 0; col < arrB[ row ].length; col++
       )
          {
             arrB[ row ][ col ] = row + col;
          }
       }
    }
```



Midterm Q4 from 04W2

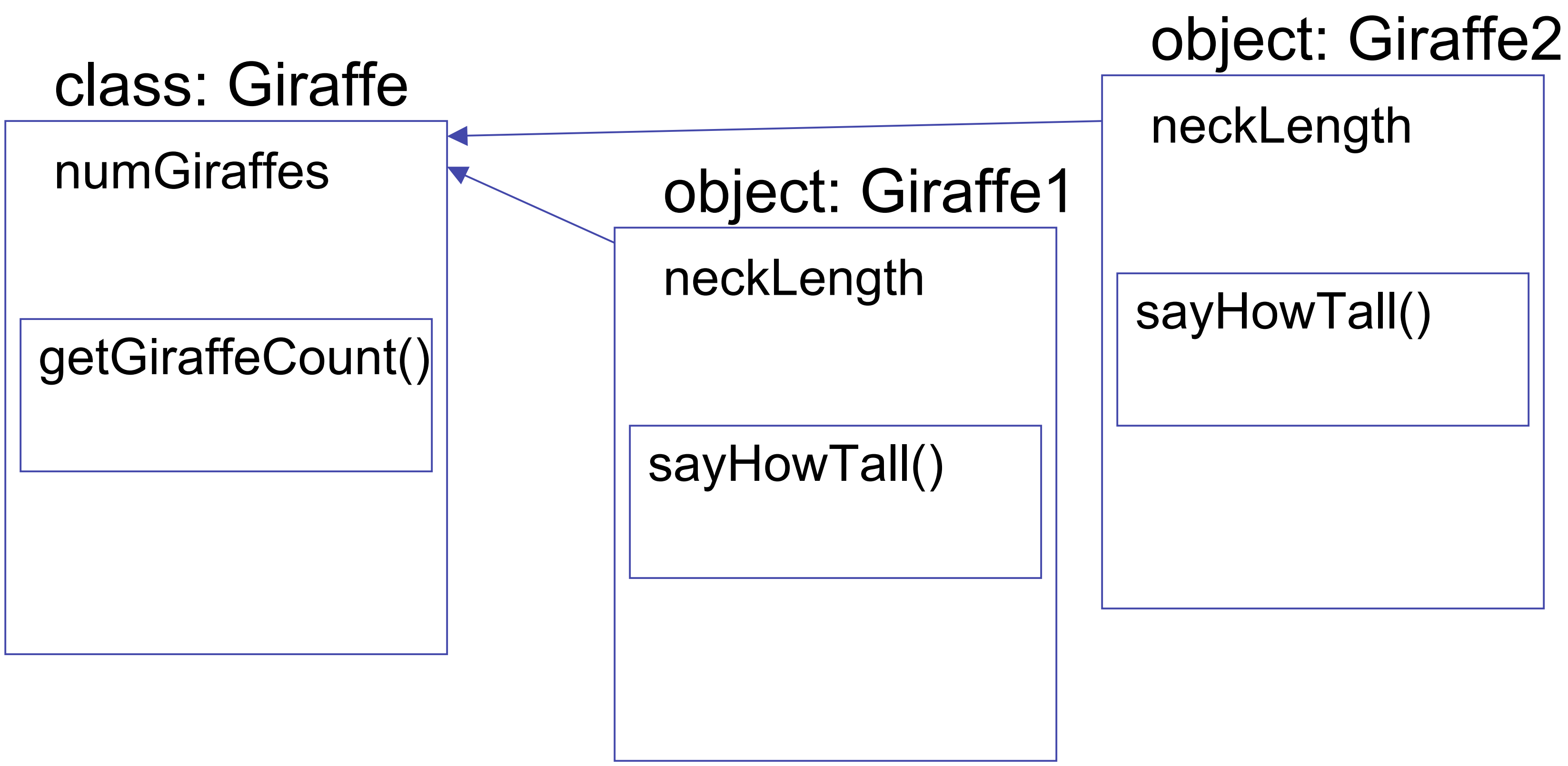
```
int[][] dataA = { { 0, 0 }, { 0, 0 } };
int[][] dataB = { { 0, 0 }, { 0, 0 } };
process( dataA, dataB );
```

```
public void process( int[][] arrA, int[][] arrB )
{
    int row;
    int col;
    int[][] arrC = { { 1, 1, 1 }, { 1, 1, 1 } };
    arrA = arrC;
    for( row = 0; row < arrB.length; row++ )
    {
        for( col = 0; col < arrB[ row ].length; col++
    )
        {
            arrB[ row ][ col ] = row + col;
        }
    }
}
```



Review: Static Fields/Methods

- Static fields belong to whole class
  - nonstatic fields belong to instantiated object
- Static methods can only use static fields
  - nonstatic methods can use either nonstatic or static fields



Review: Variable Scope

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

Variable Scope

```
public class CokeMachine4
{
  private int numberOfCans;

  public CokeMachine4()
  {
    numberOfCans = 2;
    System.out.println("Adding another machine to your empire");
  }

  public int getNumberOfCans()
  {
    return numberOfCans;
  }

  public void reloadMachine(int loadedCans)
  {
    numberOfCans = loadedCans;
  }
```

- numberOfCans variable declared inside class but not inside particular method
  - scope is entire class: can be accessed from anywhere in class

Variable Scope

```
public class CokeMachine4
{
  private int numberOfCans;

  public CokeMachine4()
  {
    numberOfCans = 2;
    System.out.println("Adding another machine to your empire");
  }

  public double getVolumeOfCoke()
  {
    double totalLitres = numberOfCans * 0.355;
    return totalLitres;
  }

  public void reloadMachine(int loadedCans)
  {
    numberOfCans = loadedCans;
  }
```

- totalLitres declared within a method
  - scope is method: can only be accessed from within method
  - variable is local data: has local scope

Variable Scope

```
public class CokeMachine4
{
  private int numberOfCans;

  public CokeMachine4()
  {
    numberOfCans = 2;
    System.out.println("Adding another machine to your empire");
  }

  public int getNumberOfCans()
  {
    return numberOfCans;
  }

  public void reloadMachine(int loadedCans)
  {
    numberOfCans = loadedCans;
  }
```

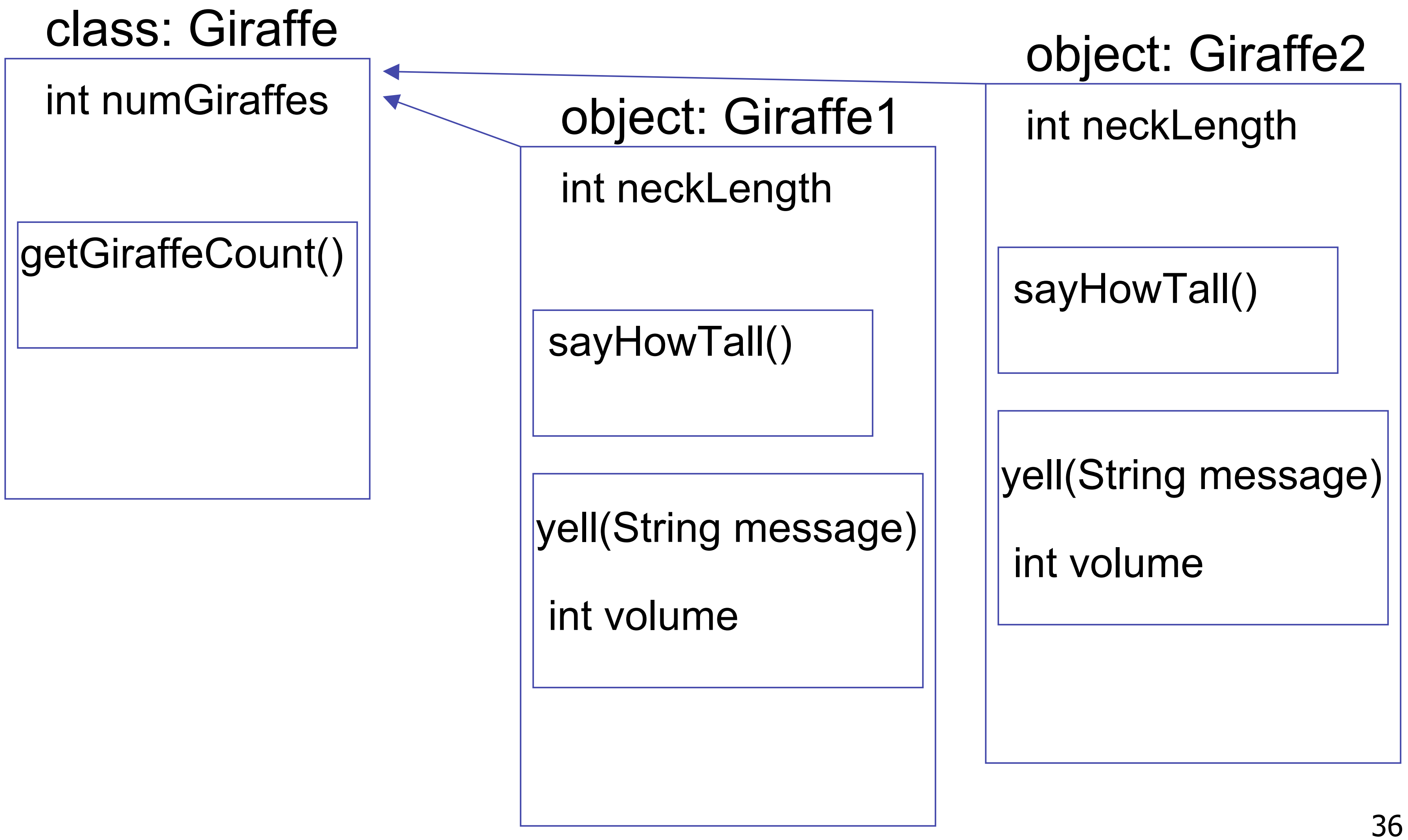
- loadedCans is method parameter
  - scope is method: also local scope
  - just like variable declared within parameter
  - accessed only within that method

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 throughout method, lifetime of method
- Parameters
  - declared in parameter list of method
  - accessible throughout method, lifetime of method

Variable Types

- Static? Instance? Local? Parameters?



Questions?

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

Reminder: CokeMachine2

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

  public CokeMachine2() {
    numberOfCans = 10;
    System.out.println("Adding another machine to your empire with "
                         + numberOfCans + " cans of Coke");
    totalMachines++;
  }
  public CokeMachine2(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 = numberOfCans - 1;
      System.out.println("Have a Coke");
      System.out.print(numberOfCans);
      System.out.println(" cans remaining");
    } else {
      System.out.println("Sold Out");
    }
  }
}
```

One Way: Copy CM2, Change Name, ...

```
public class CokeMachine2000 {
  private static int totalMachines = 0;
  private int numberOfCans;

  public CokeMachine2000() {
    numberOfCans = 10;
    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 = numberOfCans - 1;
      System.out.println("Have a Coke");
      System.out.print(numberOfCans);
      System.out.println(" cans remaining");
    } else {
      System.out.println("Sold Out");
    }
  }
```

...Then Add New Method

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

}
```

Update The SimCoke Program

```
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();
  }
}
```

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 machine to your empire with 1 cans of Coke
Have a Coke
9 cans remaining
Have a Coke
236 cans remaining
Have a Coke
0 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?

Is There An Easier Way...

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

No.

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

}
```

- Create new class called CokeMachine2000
  - inherits all methods and variables from CokeMachine2
    - mostly true...we'll see some exceptions later
  - can just add new variables and methods
- Inheritance: process by which new class is derived from existing one
  - fundamental principle of object-oriented programming

Easier Way (First Pass)

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

}
```

- 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 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();
  }
}
```

```
1 error found:
File: SimCoke2000.java  [line: 8]
Error: cannot resolve symbol
symbol   : constructor CokeMachine2000 (int)
location: class CokeMachine2000
```

OOPS! What happened?

Easier Way (Second Pass)

```
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");
  }
}
```

- 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

Testing Second Pass

```
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");
  }
}

2 errors found:
File: CokeMachine2000.java  [line: 15]
Error: numberOfCans has private access in CokeMachine2
File: CokeMachine2000.java  [line: 15]
Error: numberOfCans has private access in CokeMachine2
```

Easier Way (Third Pass)

```
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");
  }
}
```

- Subclass inherits all variables of superclass
- But private variables cannot be directly accessed, even from subclass

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

Easier Way (Third Pass)

```
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");
  }
}
```

- 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

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

Testing With SimCoke

```
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();
  }
}
```

Testing With SimCoke

```
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();
  }
}
```

```
> 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 machine to your empire with 1 cans of Coke
Have a Coke
9 cans remaining
Have a Coke
236 cans remaining
Have a Coke
0 cans remaining
Adding 150 cans to this machine
Have a Coke
149 cans remaining
>
```

Some Coke Machine History

early Coke Machine

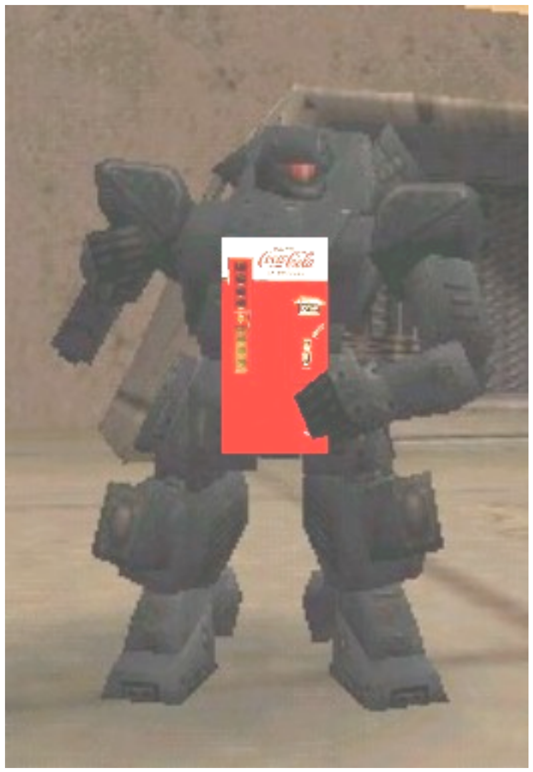
- mechanical
- sealed unit, must be reloaded at factory
- no protection against vandalism

Some Coke Machine History

Coke Machine 2000

- electro-mechanical
- can be reloaded on site
- little protection against vandalism

Some Coke Machine History

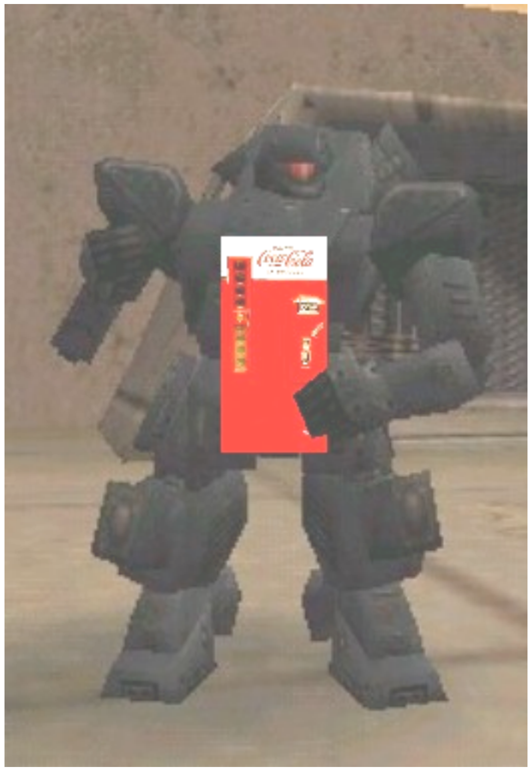


Coke Machine UA*

- 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

Method Overriding

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

  public CokeMachine2()
  {
    numberOfCans = 10;
    System.out.println("Adding another machine to your empire with "
                     + numberOfCans + " cans of Coke");
    totalMachines++;
  }

  public CokeMachine2(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;
  }
```

Method Overriding

```
  public int getNumberOfCans()
  {
    return numberOfCans;
  }

  public void buyCoke()
  {
    if (numberOfCans > 0)
    {
      numberOfCans = numberOfCans - 1;
      System.out.println("Have a Coke");
      System.out.print(numberOfCans);
      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");
  }
}
```

Method Overriding

```
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("loading " + n + " cans");
  }

  public void vandalize()  // this overrides the vandalize method from parent
  {
    System.out.println("Stop it!  Never mind, here's my money");
  }

}
```

Method Overriding

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

  public CokeMachineUA(int n)
  {
    super(n);
  }

  public void vandalize()  // this overrides the vandalize method from parent
  {
    System.out.println("Eat lead and die, you slimy Pepsi drinker!!");
  }
}
```

Method Overriding

```
public class SimVend
{
  public static void main (String[] args)
  {
    CokeMachine2[] mymachines = new CokeMachine2[5];
    mymachines[0] = new CokeMachine2();
    mymachines[1] = new CokeMachine2000();
    mymachines[2] = new CokeMachineUA();

    for (int i = 0; i < mymachines.length; i++)
    {
      if (mymachines[i] != null)
      {
        mymachines[i].vandalize();
      }
    }
  }
}
```

```
> 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
Please don't hurt me...take all my money
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

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

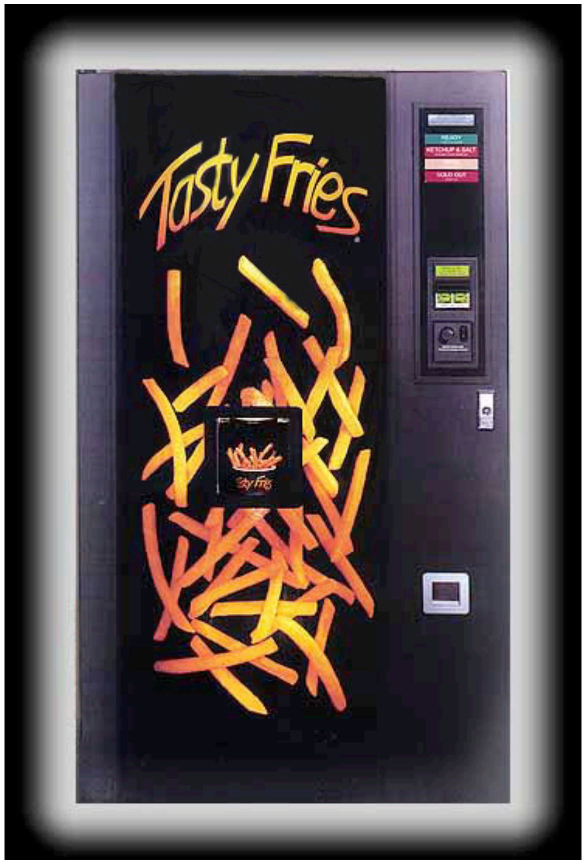
- From Just Java 2 by Peter van der Linden:
  - Polymorphism is a complicated name for a straightforward concept. It merely means using the same one name to refer to different methods. "Name reuse" would be a better term.

- Polymorphism made possible in Java through method overloading and method overriding
  - remember method overloading?

Method Overloading and Overriding

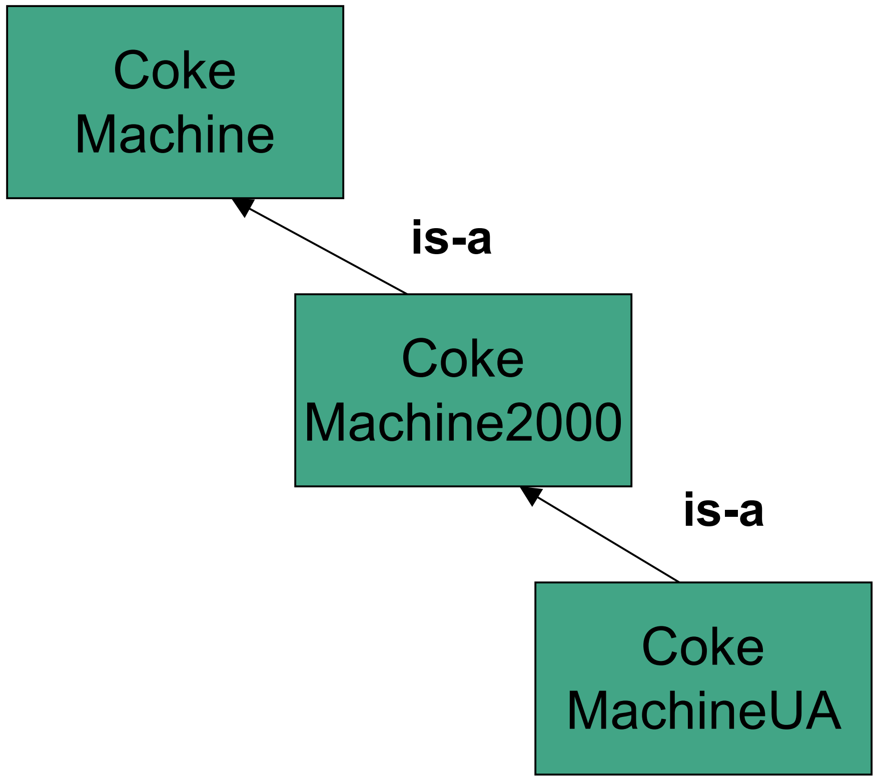
- Method overloading: "easy" polymorphism
  - in any class can use same name for several different (but hopefully related) methods
  - methods must have different signatures so that compiler can tell which one is intended
- Method overriding: "complicated“ polymorphism
  - subclass has method with same 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

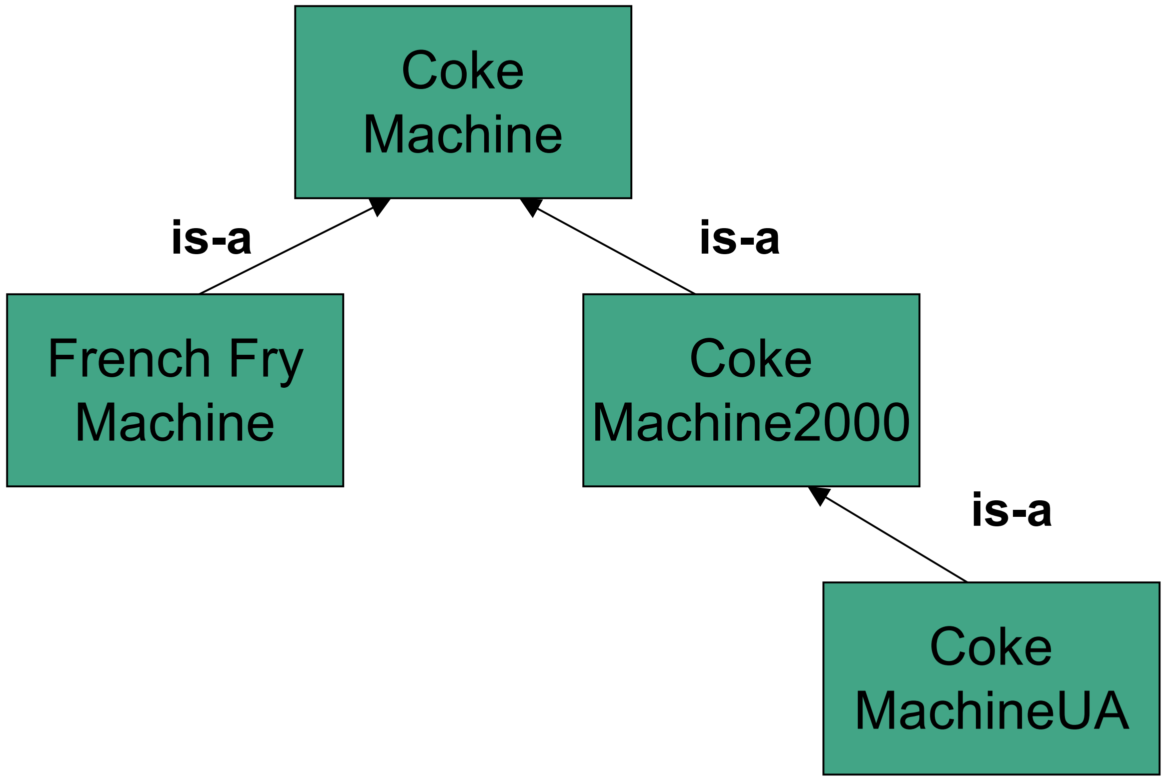


- Expand vending machine empire to include French fry machines
  - is a French fry machine a subclass of Coke Machine?

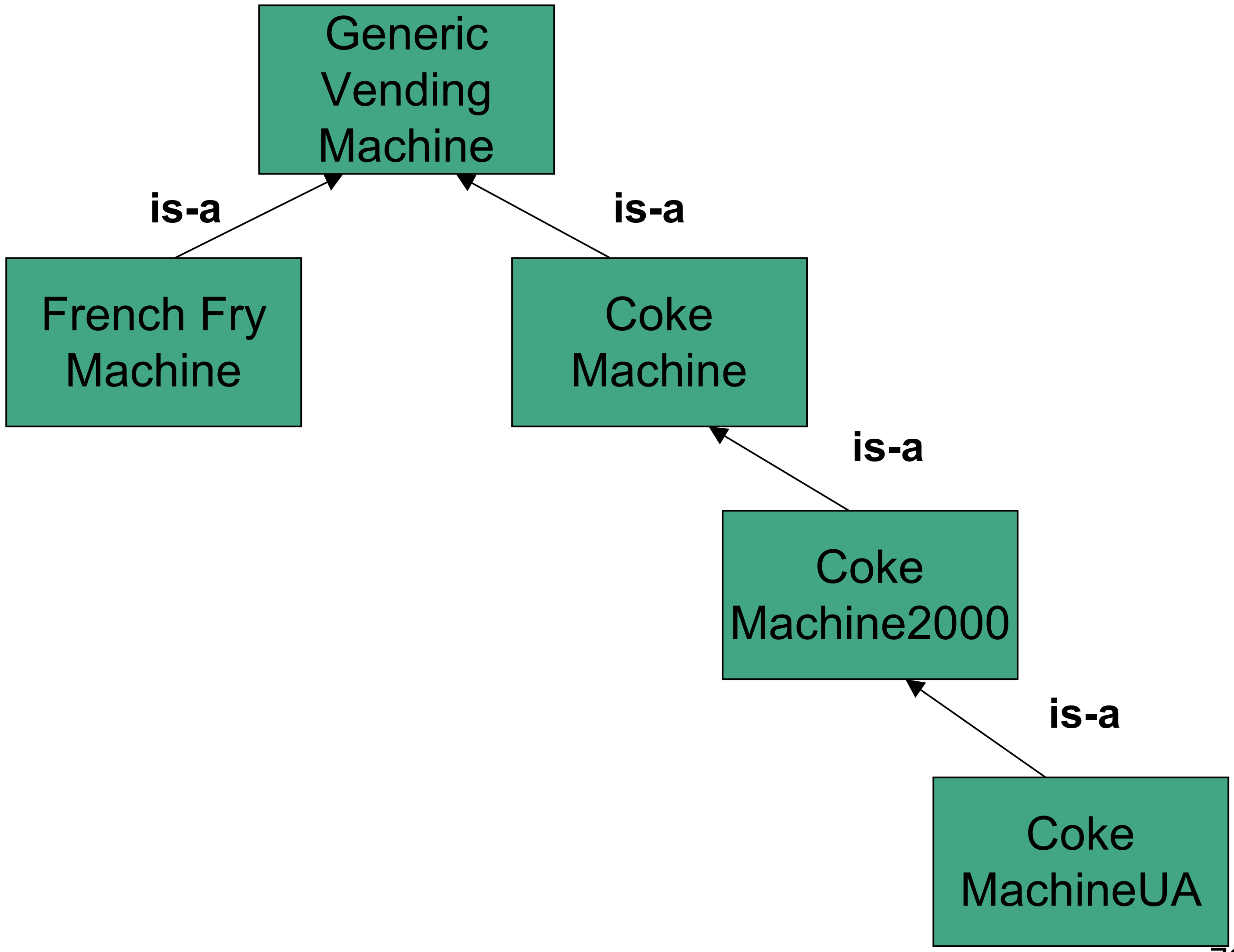
If We Have This Class Hierarchy...



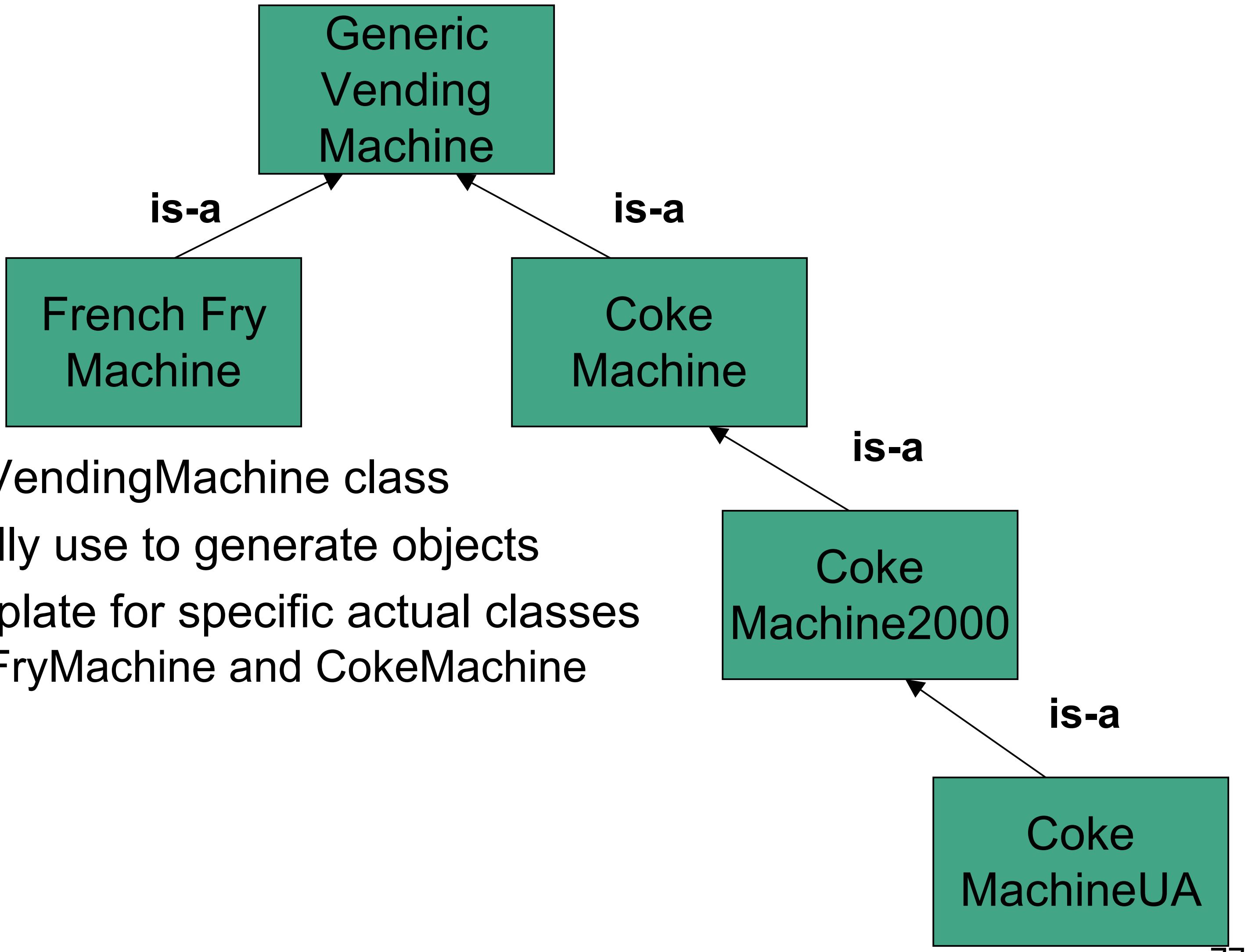
...Does This Make Sense?



Does This Make More Sense?

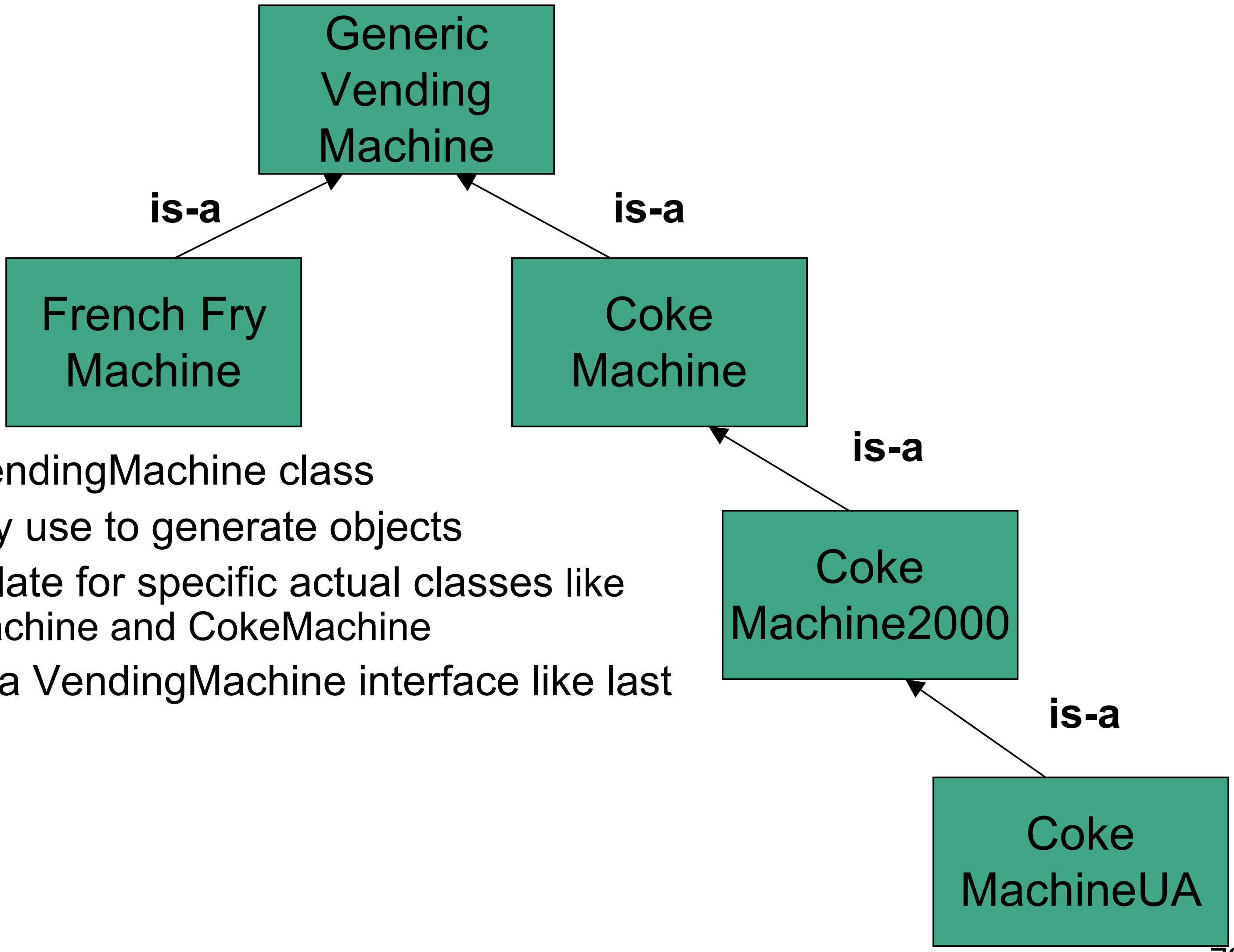


Does This Make More Sense?



- 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

Abstract Classes

```
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;
  }
```

Abstract Classes

```
  public void loadItems(int n)
  {
    numberOfItems = n;
  }

  public int getNumberOfItems()
  {
    return numberOfItems;
  }

}
```

Abstract Classes

```
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");
    }
  }
```

Abstract Classes

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