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

Tamara Munzner

Inheritance II

Lecture 34, Mon Apr 12 2010

borrowing from slides by Kurt Eiselt

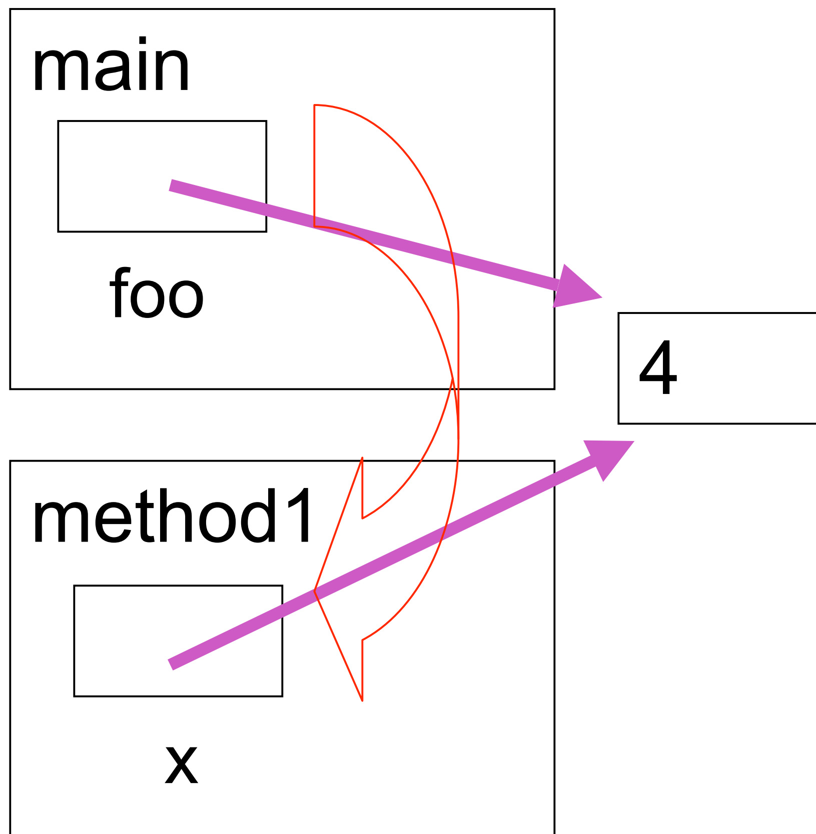
http://www.cs.ubc.ca/~tmm/courses/111-10

Reading

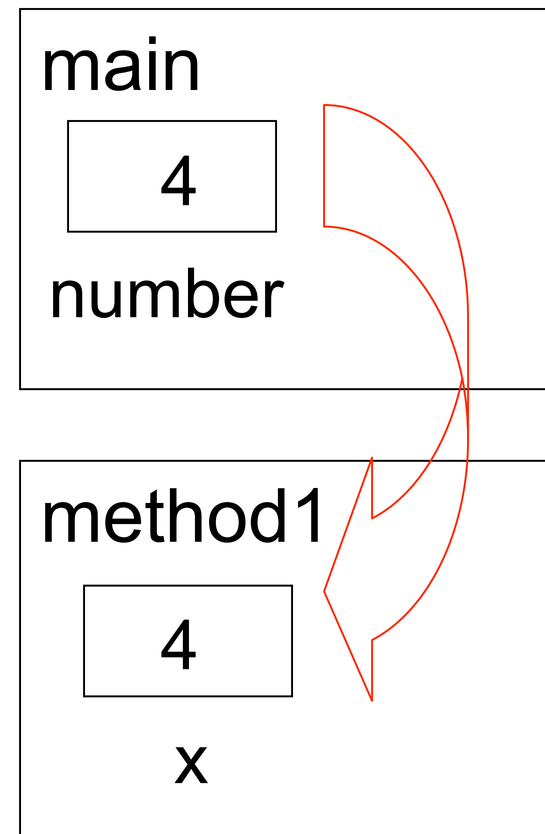
- This week reading: 2.11-2.12, 9.5-9.8,10.9-10.10
  - 5.1-5.2, 11.5, 12.2-12.3 (2nd edition)
  - we will not get through all this material in lecture
    - minimal/no coverage on final
    - weekly reading question still due last class Wed 4/14

Recap: Parameter Passing Pictures

- object as parameter
  - copy of pointer made
  - pass by **reference**
  - modifications **visible** outside method

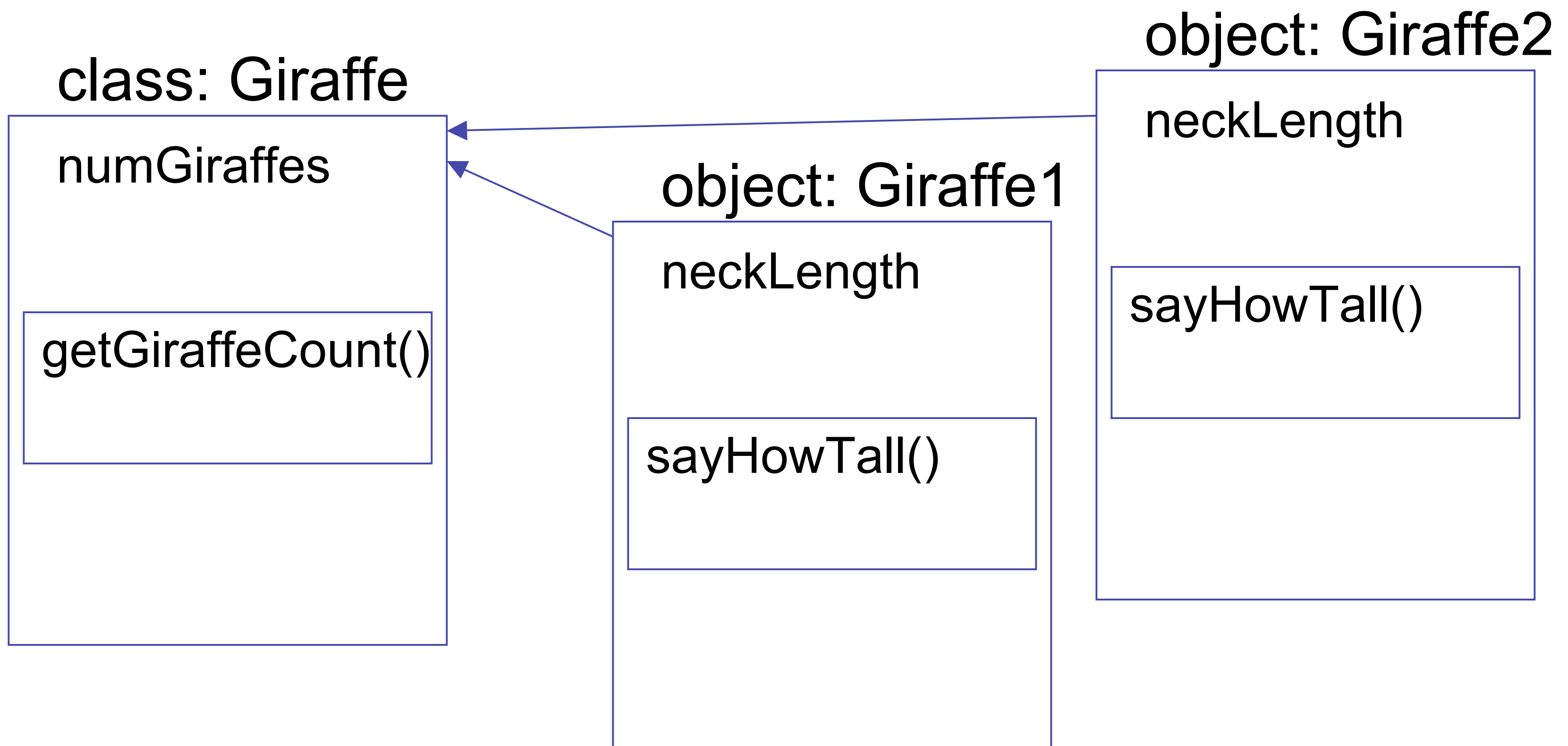


- primitive as parameter
  - copy of value
  - pass by **value**
  - modifications **not visible** outside method



Recap: 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



Recap: Variable Types and Scope

- 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

Recap: Inheritance

- Inheritance: process by which new class is derived from existing one
  - fundamental principle of object-oriented programming
- Create new child class (subclass) that `extends` existing parent one (superclass)
  - inherits all methods and variables
    - except constructor
  - can just add new variables and methods

Recap: Inheritance and Constructors

```
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

Recap: Inheritance and Scope

- Subclasses inherits but cannot directly access private fields or variables of superclass
- Protected variables can be directly accessed from declaring class and any classes derived from it

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

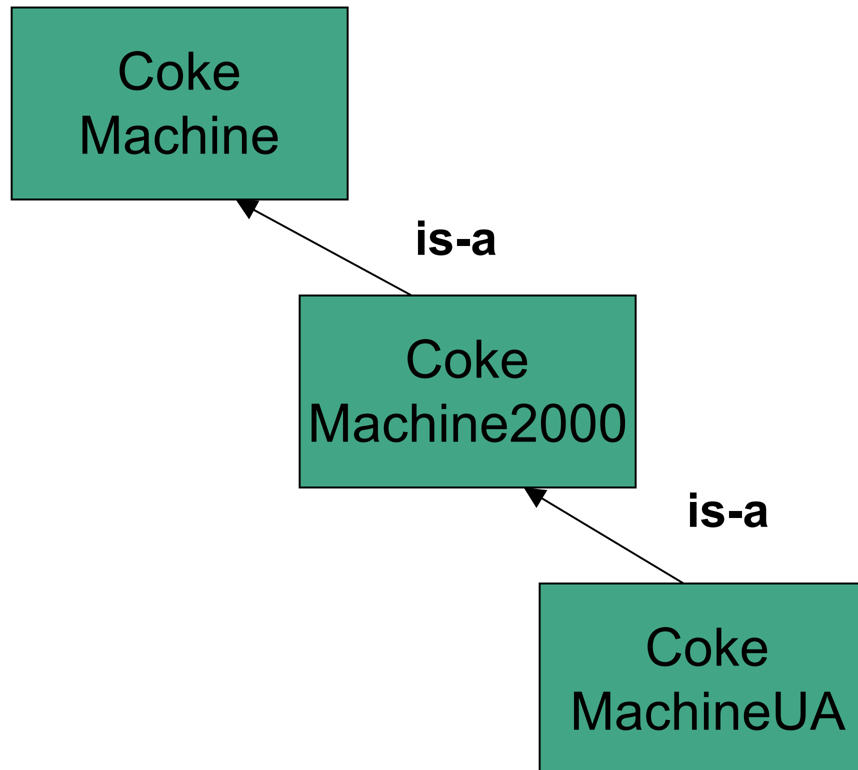
Objectives

- Understanding when and how to use abstract classes
- Understanding tradeoffs between interfaces and inheritance

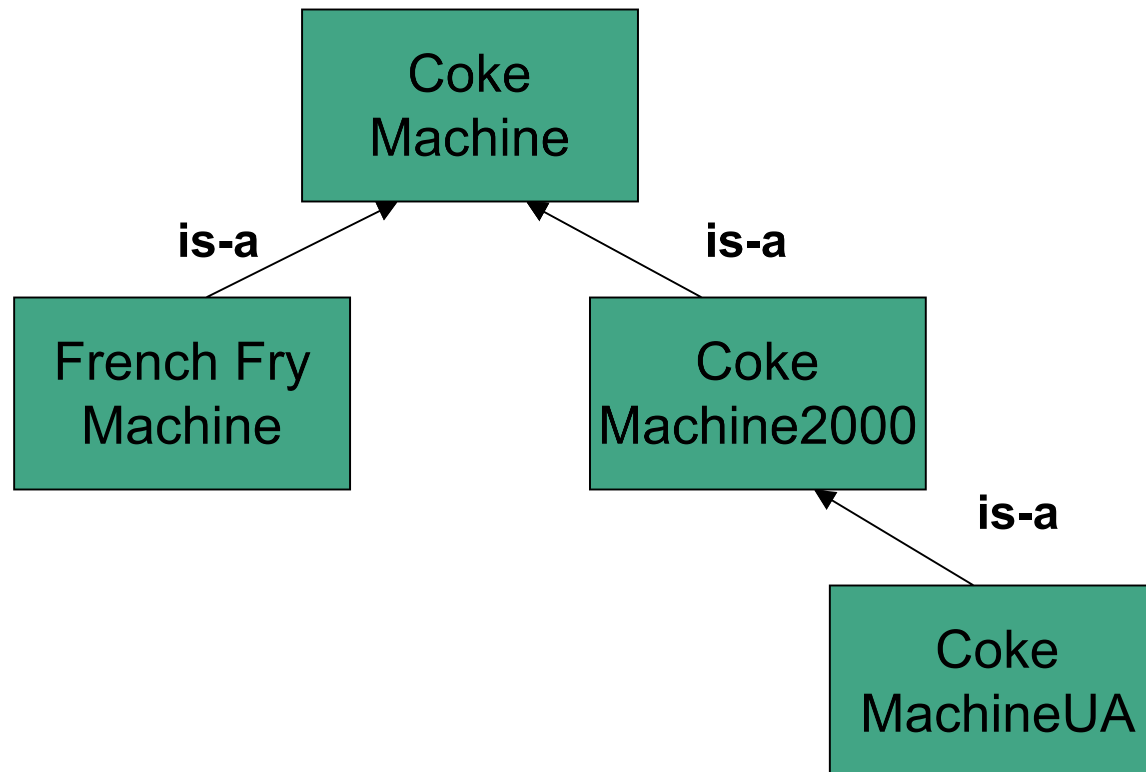
A New Wrinkle

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

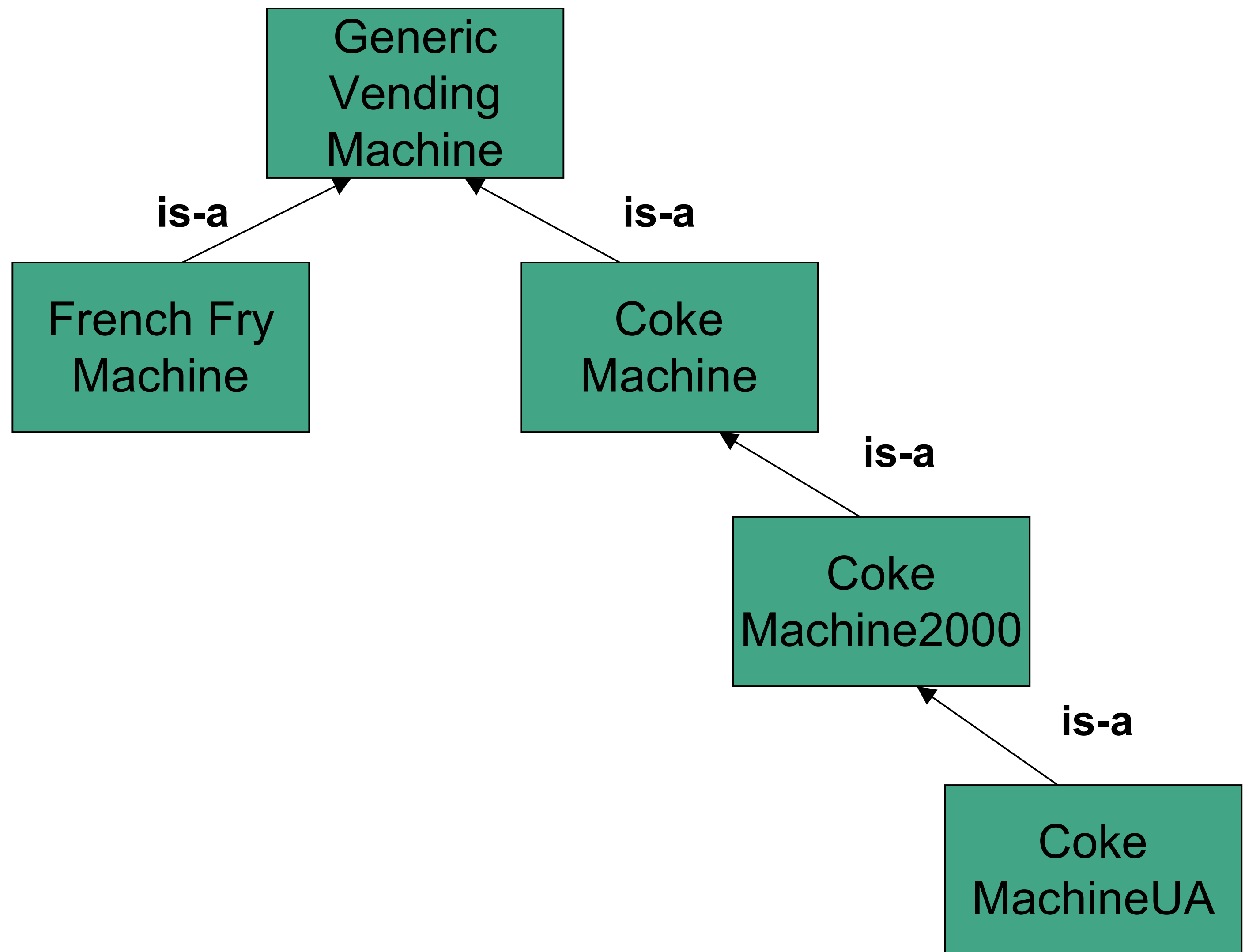
If We Have This Class Hierarchy...



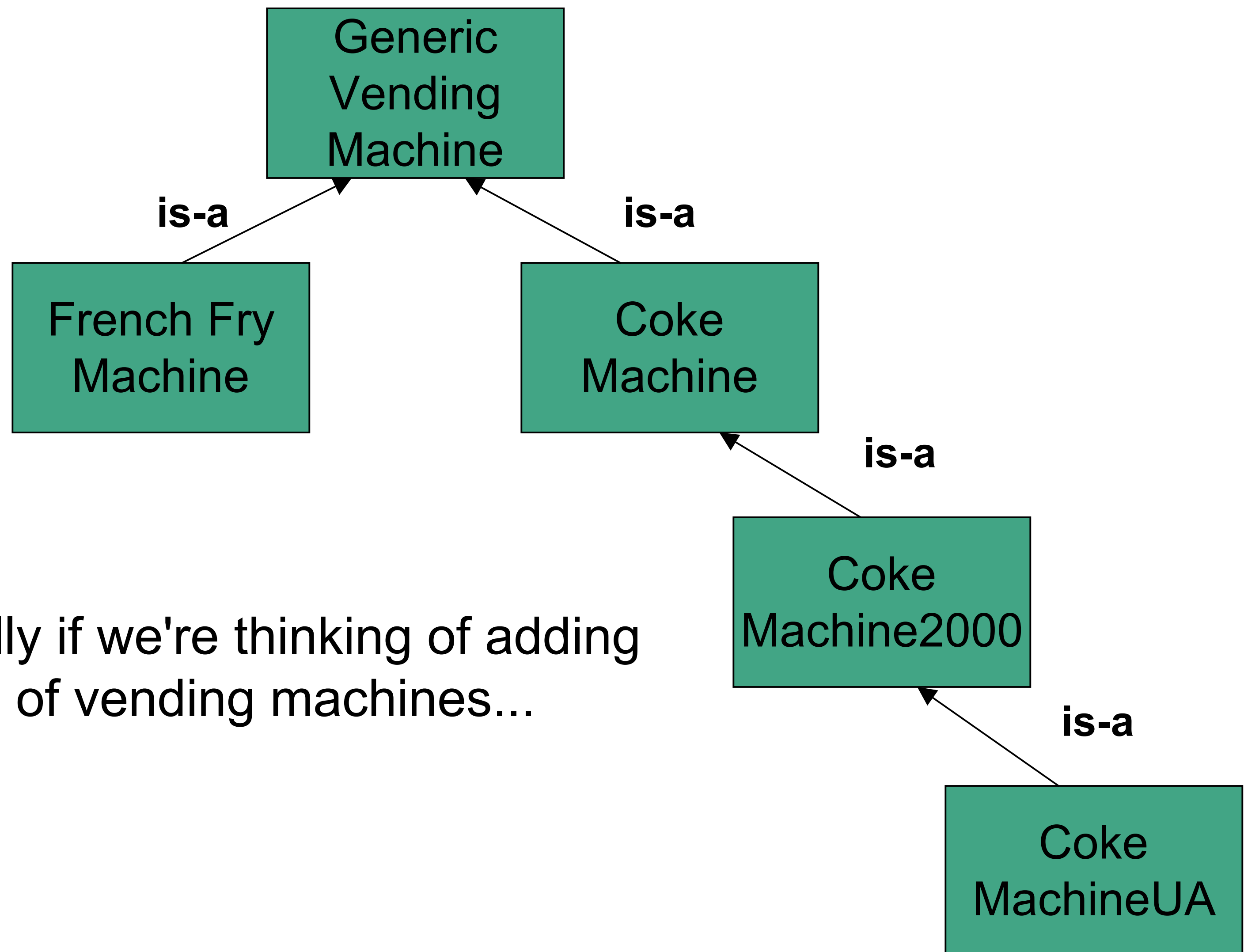
...Does This Make Sense?



Does This Make More Sense?

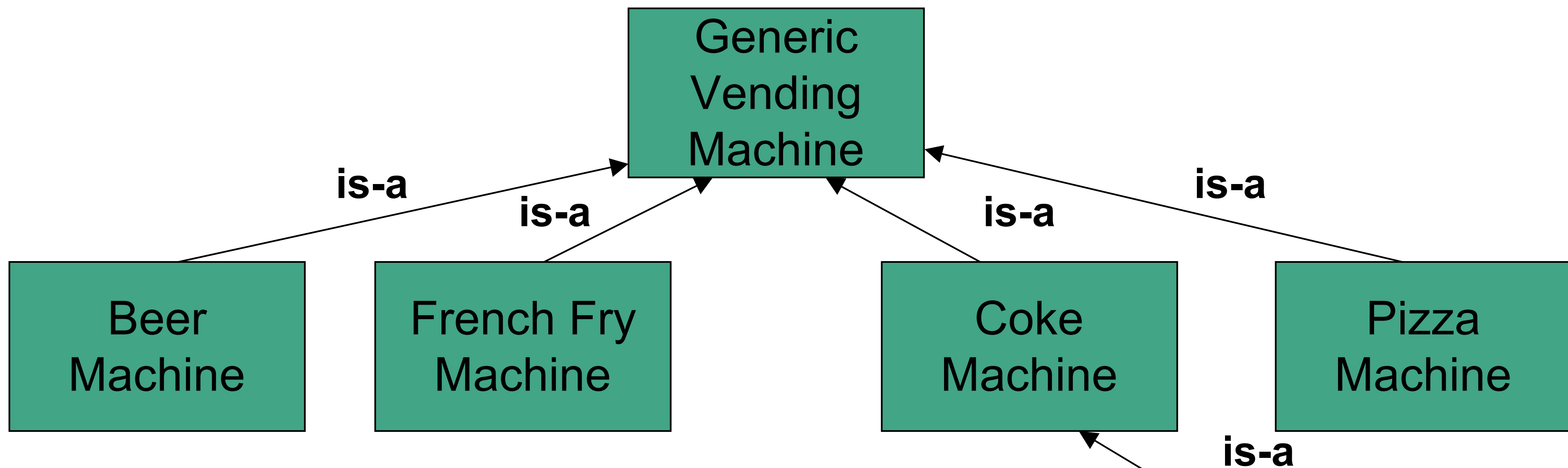


Does This Make More Sense?

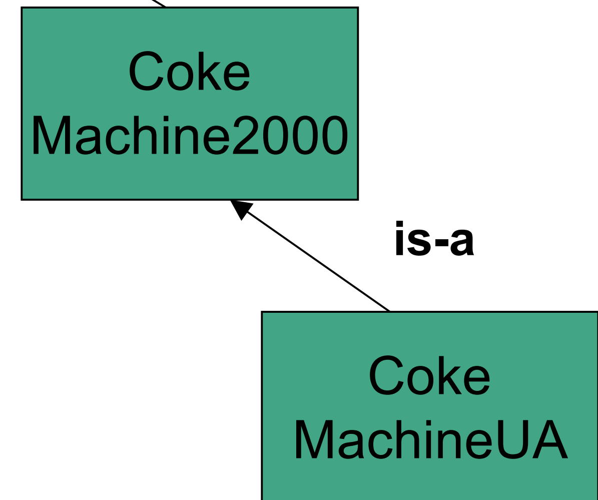


- Yes
  - especially if we're thinking of adding all kinds of vending machines...

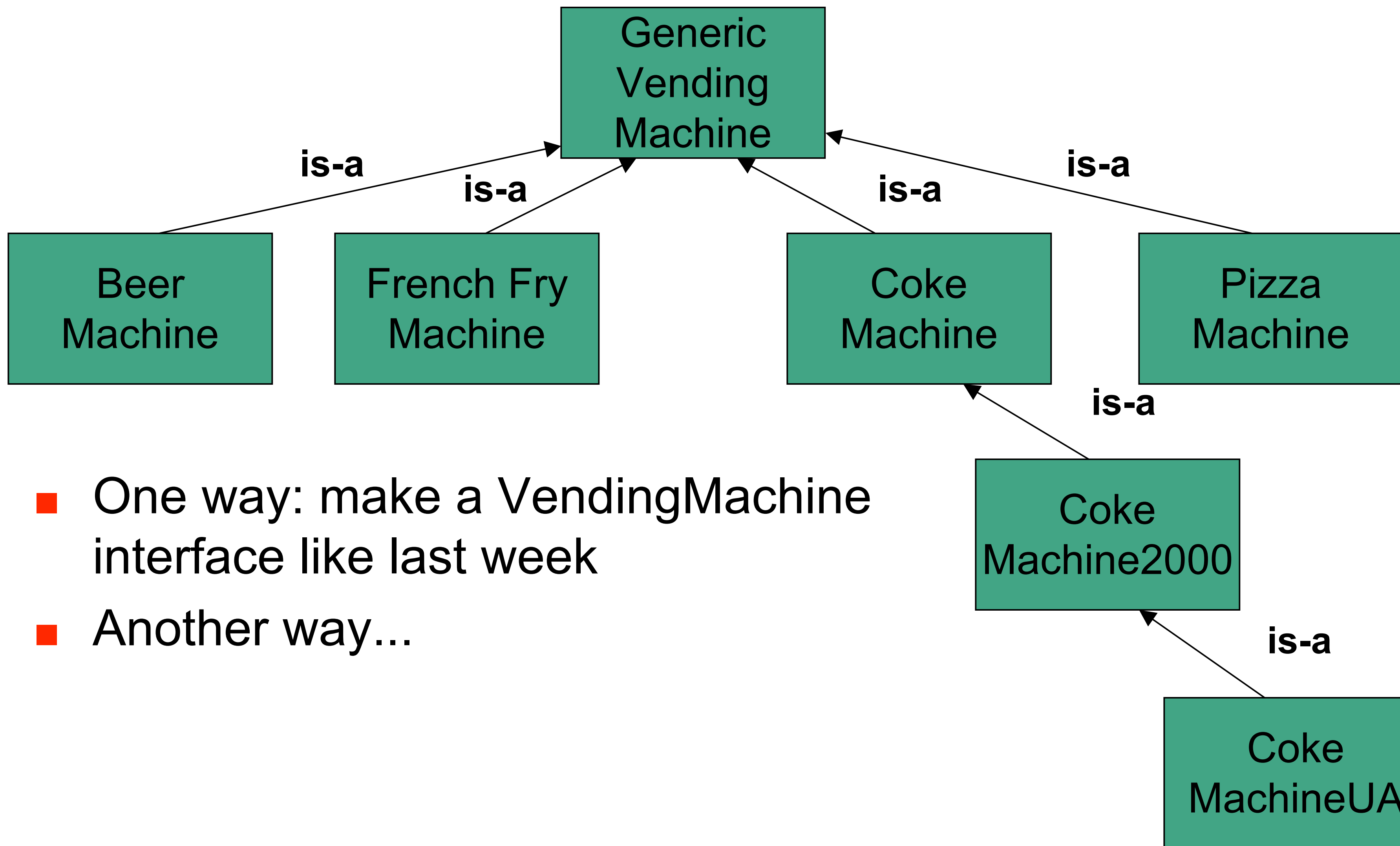
Does This Make More Sense?



- Yes
  - especially if we're thinking of adding all kinds of vending machines...
  - want our classes to be more specific as we go down class hierarchy
    - is French Fry Machine more or less specific than Coke Machine?
    - neither, both specific versions of generic Vending Machine class



Does This Make More Sense?



- One way: make a VendingMachine interface like last week
- Another way...

Inheritance Solution

```
public 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;
  }
```

Inheritance Solution

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

  public int getNumberOfItems()
  {
    return numberOfItems;
  }

}
```

Inheritance Solution

```
public class CokeMachine3 extends GenericVendingMachine
{
  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");
    }
  }
```

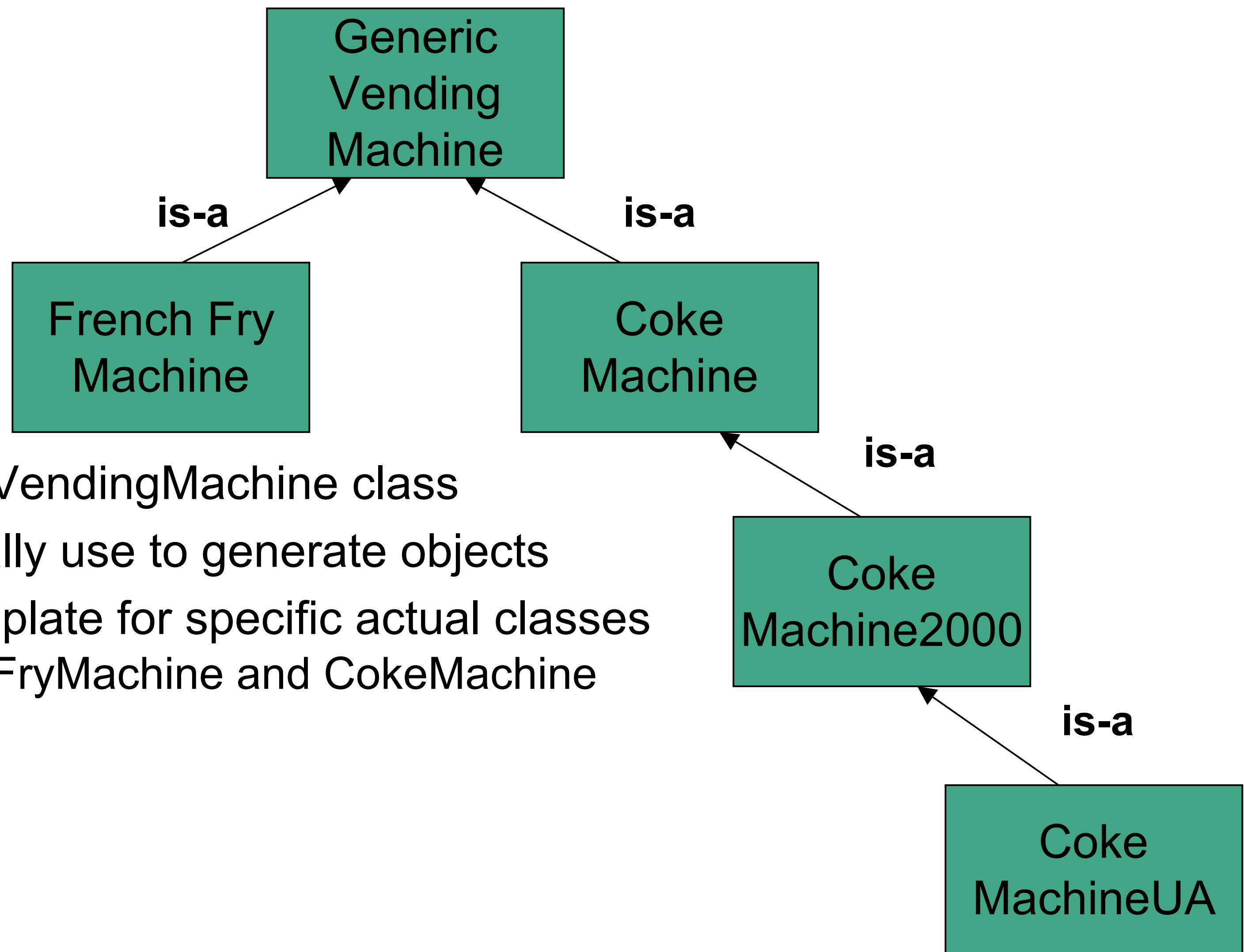
Inheritance Solution

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

Inheritance Solution

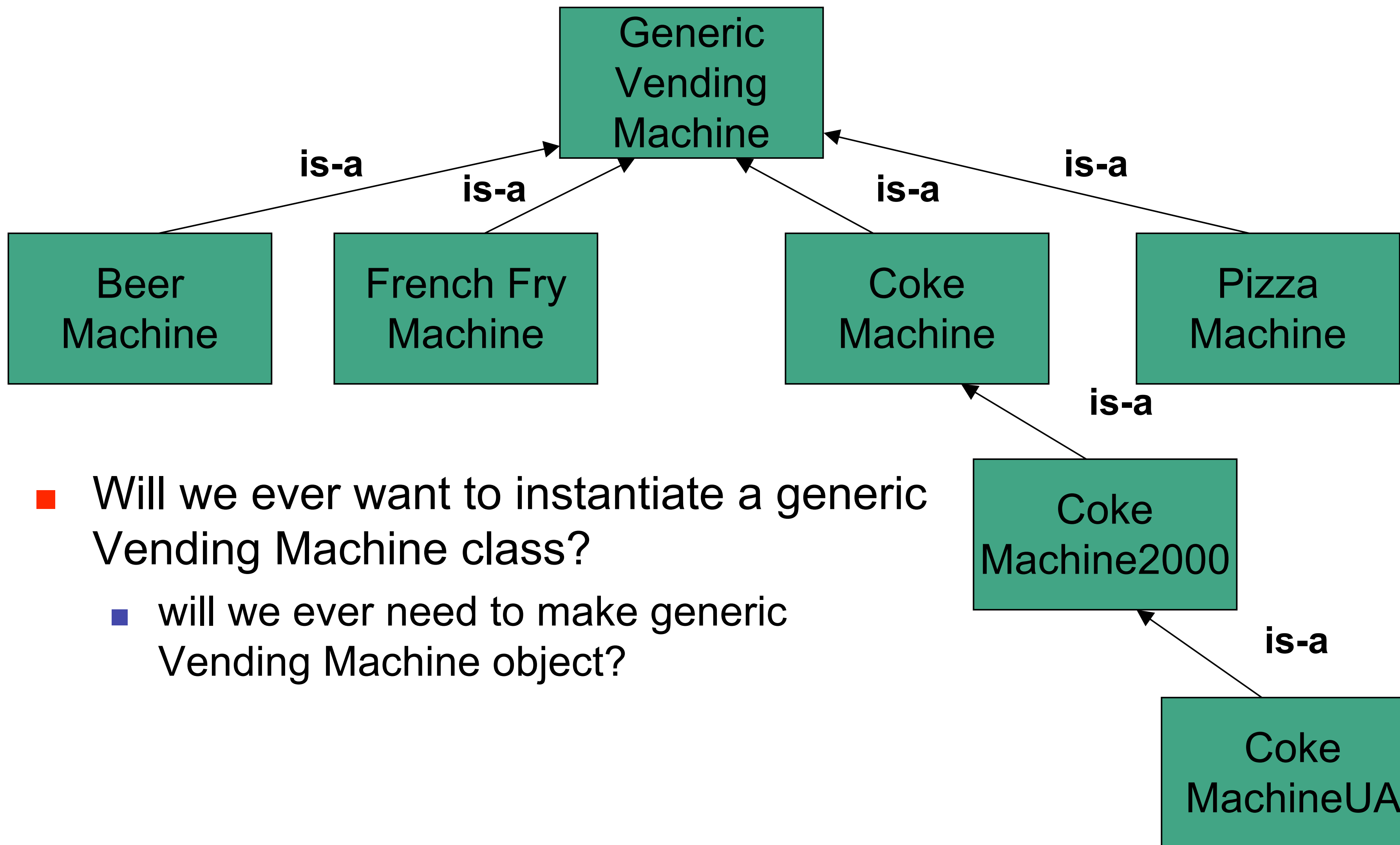
```
public class CokeMachine2000 extends CokeMachine3
{
  public CokeMachine2000()
  {
    super();
  }

  public CokeMachine2000(int n)
  {
    super();
    this.loadItems(n);
  }

  public void loadCoke(int n)
  {
    super.loadCoke(n);
    System.out.println("Loading in the new millennium!");
  }
}
```

Inheritance From Generic Objects



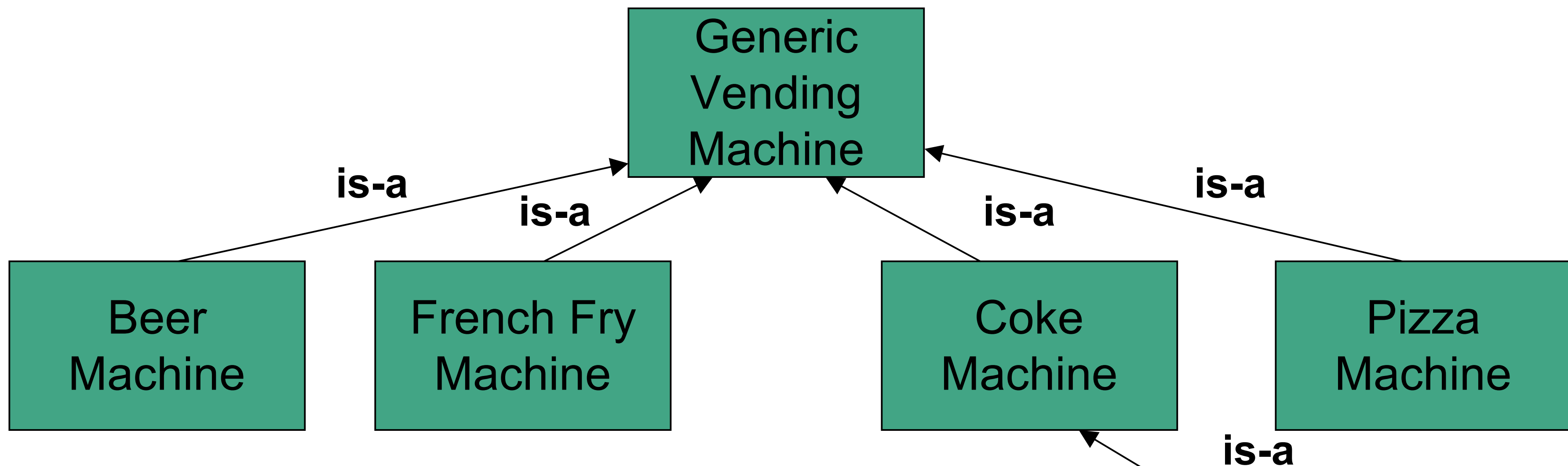
- Want generic VendingMachine class
  - don’t actually use to generate objects
  - use as template for specific actual classes like FrenchFryMachine and CokeMachine

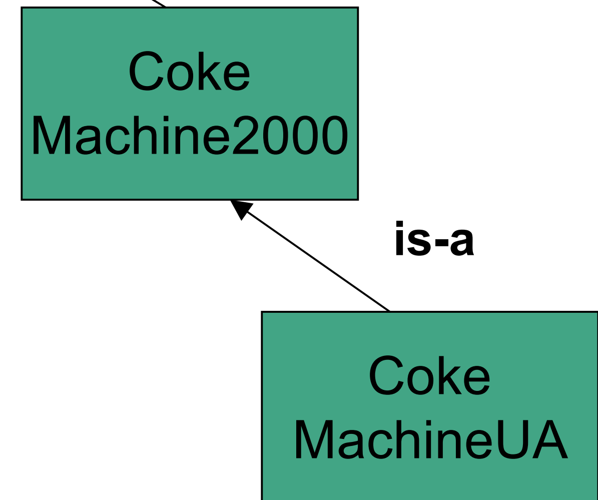
Inheritance From Generic Objects



- Will we ever want to instantiate a generic Vending Machine class?
  - will we ever need to make generic Vending Machine object?

Inheritance From Generic Objects





- Will we ever want to instantiate a generic Vending Machine class?
  - will we ever need to make generic Vending Machine object?
  - No, not in our simulated vending world!
  - How would we use one? What would be a real-life equivalent?

Inheritance From Generic Objects

- Introduced CokeMachineUA to combat vandalism and theft
- Could just add vandalize() methods to CM, CM2000, CMUA
  - but we want to ensure that all Vending Machines have vandalize() methods
  - want all of them to be different
    - if put into base class at top, easy to have them identical
    - no way to force method overriding

Abstract Classes

- Abstract class: not completely implemented
- Usually contains one or more abstract methods
  - has no definition: specifies method that should be implemented by subclasses
  - just has header, does not provide actual implementation for that method
- Abstract class uses abstract methods to specify what interface to descendant classes must look like
  - without providing implementation details for methods that make up interface
- Example: require that all subclasses of VendingMachine class implement **`vandalize()`** method
  - method might differ greatly between one subclass and another
  - use an abstract method

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

- Use abstract class for generic template
  - can use abstract methods

- Making abstract method
  - Use restricted word **`abstract`** in method header
  - do not provide a method body
  - just end method header with semicolon

Vending Machine Class Revisited

```
public abstract class VendingMachine
{
  private int numberOfItems;

  public VendingMachine()
  {
    numberOfItems = 0;
  }

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

  public abstract void vandalize();

}
```

Abstract Methods and Abstract Classes

- What happens when we try to compile it all now?
  - Java tells us that there's an abstract class we have to implement

Abstract Methods and Abstract Classes

- What happens when we try to compile it all now?
  - Java tells us that there's an abstract class we have to implement
  - Could put this CokeMachine class:

```
public void vandalize()
 {
   System.out.println("Take all my money, and have a Coke too");
 }
```

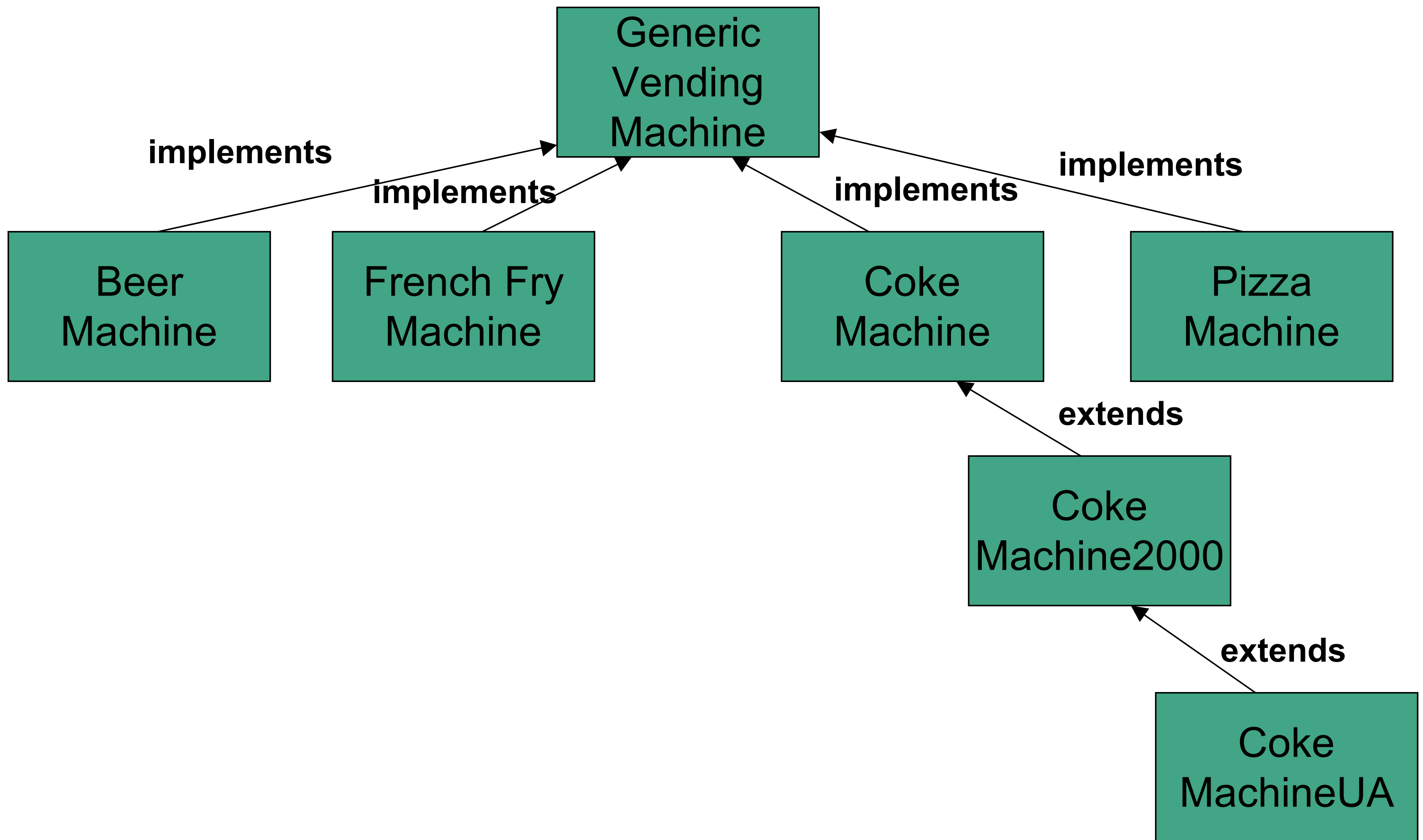
Abstract Methods and Abstract Classes

- What happens when we try to compile it all now?
  - Java tells us that there's an abstract class we have to implement
  - Could put this CokeMachine class:

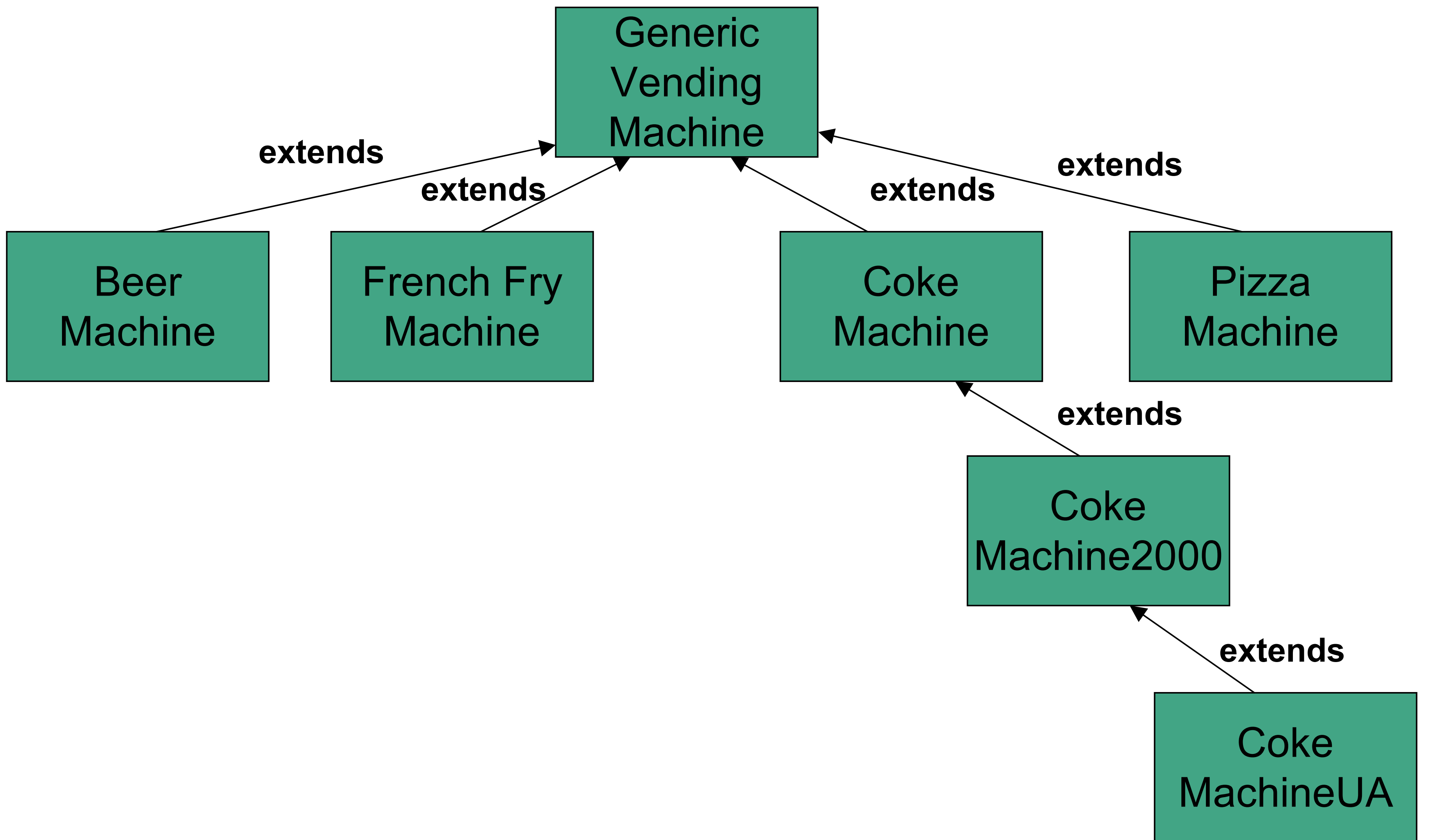
```
public void vandalize()
{
  System.out.println("Take all my money, and have a Coke too");
}
```

- Do we have to implement method in CokeMachine2000 and CokeMachineUA classes too?
  - Yes, if we want them to behave differently when they're vandalized
    - original intent

Which Organization?



Which Organization?



Interfaces vs. Abstract Classes

- If we can have abstract class that contains only abstract methods, why do we need interfaces?

Interfaces vs. Abstract Classes

- If we can have abstract class that contains only abstract methods, why do we need interfaces?
  - Java does not support multiple inheritance: child classes inheriting attributes from multiple parent classes
    - other object-oriented languages do
  - multiple inheritance can be good, but causes problems
    - what if child class inherits two different methods with same signature from two different parents?
      - which one should be used?

Interfaces vs. Abstract Classes

- Java's formal interface provides some of the utility of multiple inheritance without the problems
  - class can implement more than one interface
  - can do this at same time it extends class
- Interface allows us to create classes that "inherit" features from multiple places

Interfaces vs. Abstract Classes

- Java's formal interface provides some of the utility of multiple inheritance without the problems
  - class can implement more than one interface
  - can do this at same time it extends class
- Interface allows us to create classes that "inherit" features from multiple places
- Why is problem from previous slide solved?
  - might have multiple method headers with same signature

Interfaces vs. Abstract Classes

- Java's formal interface provides some of the utility of multiple inheritance without the problems
  - class can implement more than one interface
  - can do this at same time it extends class
- Interface allows us to create classes that "inherit" features from multiple places
- Why is problem from previous slide solved?
  - might have multiple method headers with same signature
  - but only one will have an actual definition
    - no ambiguity on which will be used
    - but still could be problem with different return types

Interfaces vs. Abstract Classes

- Another useful feature provided by interfaces:
  - inheritance happens between classes that are related
  - But classes can implement completely unrelated interfaces
    - and that can be useful

Interfaces vs. Abstract Classes

- Another useful feature provided by interfaces:
  - inheritance happens between classes that are related
  - But classes can implement completely unrelated interfaces
    - and that can be useful
- Example: implement interfaces for
  - computer, printer, cell phone, vending machine
  - create class for new interactive vending machines that:
    - vend Cokes, show annoying music videos, phone their owner when they're running low on product, and spit out coupons for free prizes

How Interfaces Differ From Abstract Classes

- Abstract class is incomplete class that requires further specialization
  - interface is just specification or prescription for behavior

from Just Java 2 by Peter van der Linden

How Interfaces Differ From Abstract Classes

- Abstract class is incomplete class that requires further specialization
  - interface is just specification or prescription for behavior
- Inheritance implies specialization, interface does not
  - interface just implies "We need something that does 'foo' and here are ways that users should be able to call it.“

from Just Java 2 by Peter van der Linden

How Interfaces Differ From Abstract Classes

- Abstract class is incomplete class that requires further specialization
  - interface is just specification or prescription for behavior
- Inheritance implies specialization, interface does not
  - interface just implies "We need something that does 'foo' and here are ways that users should be able to call it.“
- Class can implement several interfaces at once
  - but class can extend only one parent class

from Just Java 2 by Peter van der Linden

Interfaces vs. Abstract Classes: Bottom Line

- Use abstract class to initiate a hierarchy of more specialized classes
- Use interface to say, "I need to be able to call methods with these signatures in your class."
- Use an interface for some semblance of multiple inheritance

from Just Java 2 by Peter van der Linden

Interfaces vs. Abstract Classes

- Interface can only extend another interface
  - cannot extend abstract class or "concrete" class
- Class can legally implement only some methods of interface if it's abstract class
  - then must be further extended through inheritance before can be instantiated

from Just Java 2 by Peter van der Linden

Who Can Do What?

- Interface can be implemented only by class or abstract class
- Interface can be extended only by another interface
- Class can be extended only by class or abstract class
- Abstract class can be extended only by class or abstract class
- Only classes can be instantiated as objects
  - Interfaces are not classes and cannot be instantiated
  - Abstract classes may have undefined methods and cannot be instantiated