University of British Columbia
CPSC 111, Intro to Computation
Jan-Apr 2006

Tamara Munzner

Interfaces, Polymorphism

Lecture 20, Tue Mar 21 2006

based on slides by Kurt Eiselt

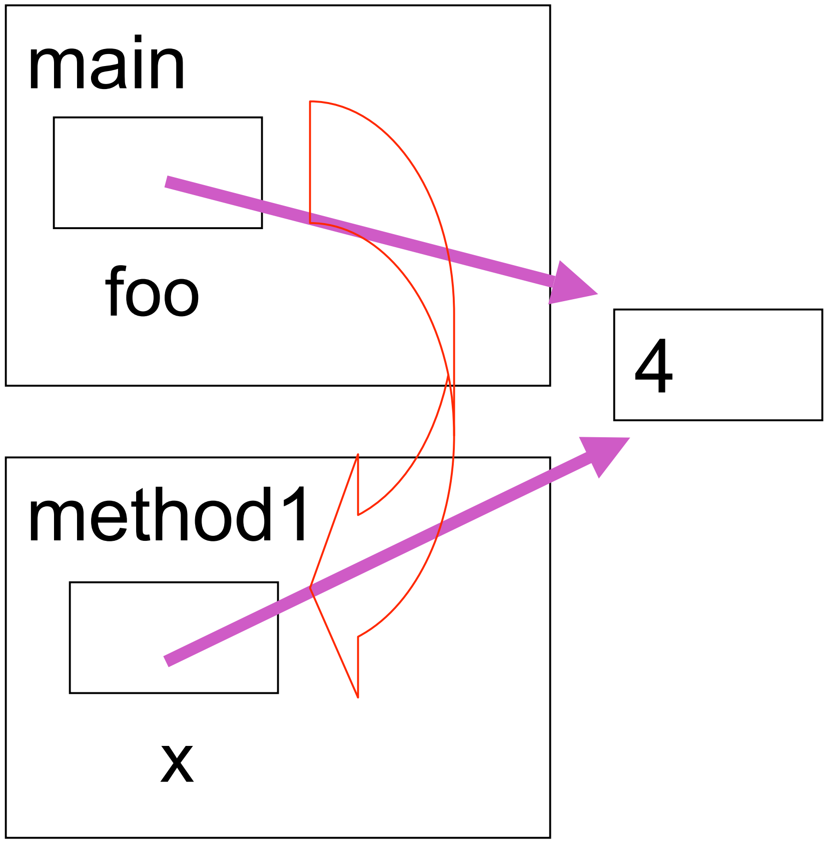
http://www.cs.ubc.ca/~tmm/courses/cpsc111-06-spr

News

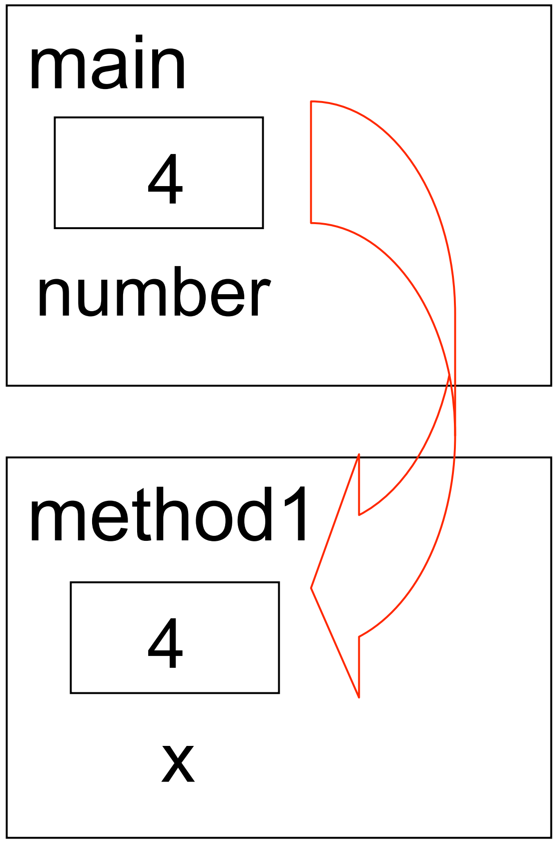
- labs this week
  - midterms returned
  - work through what you got wrong on midterm
  - can earn back up to 5 out of 70 points

Recap: Parameter Passing

object as parameter: copy of pointer made

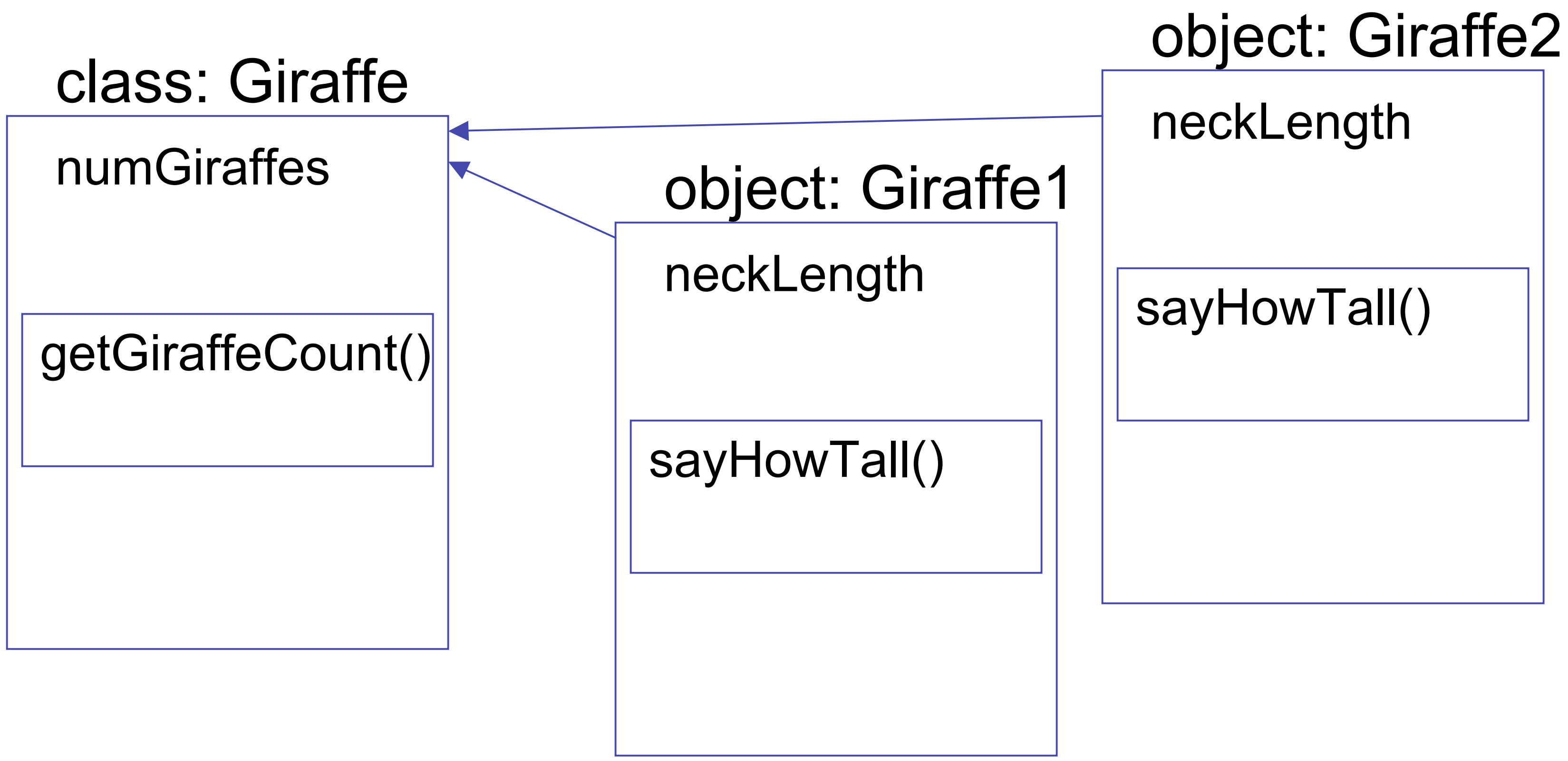


prim as parameter: copy of value



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

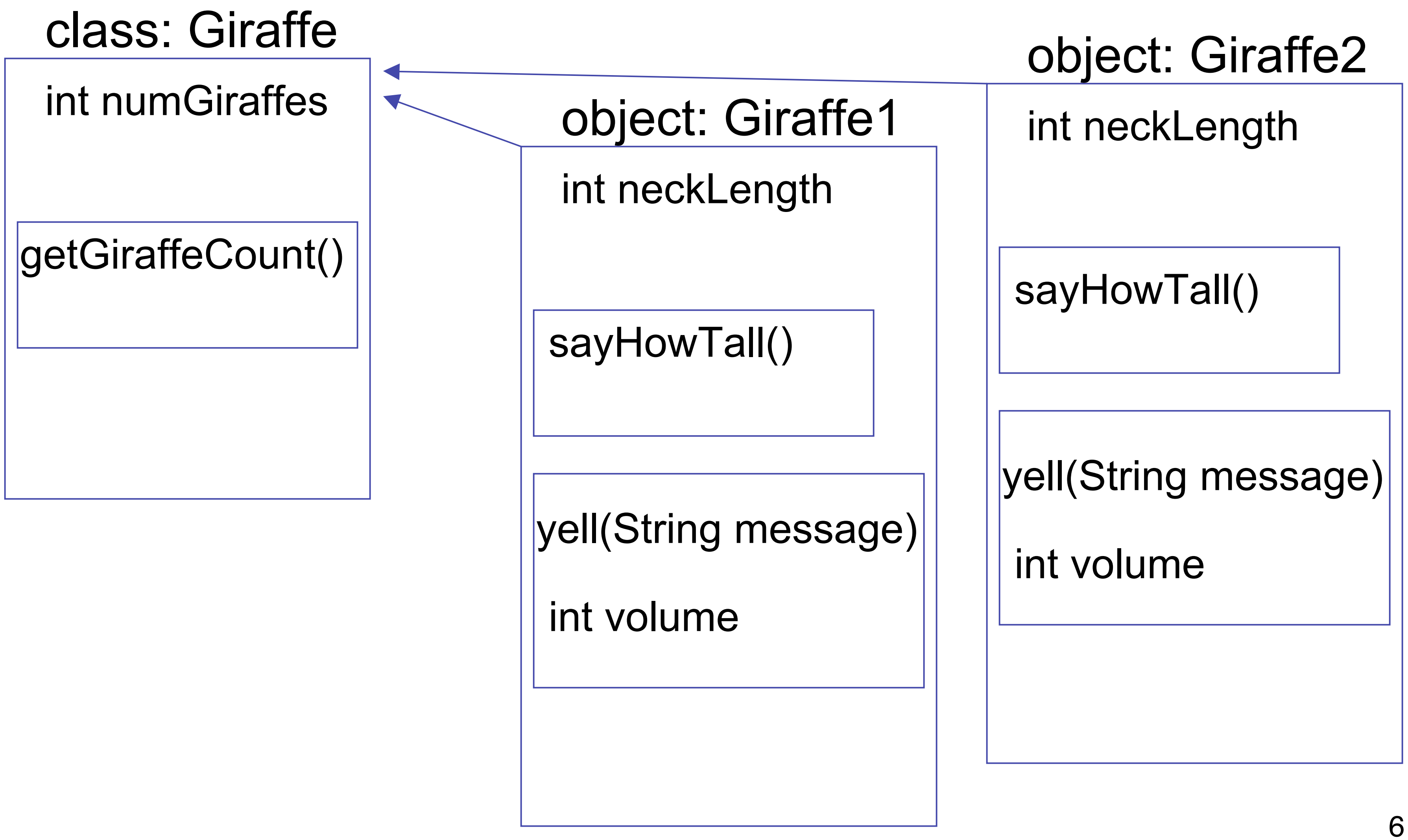


Recap: Variable Types and Scope

- 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
  - acessible throughout method, lifetime of method

Recap: Variable Types

- Static? Instance? Local? Parameters?



Here's a puzzler...

How does **`System.out.println()`** accept different data types as parameters?

```
public class PrintlnTest
{
  public static void main(String[] args)
  {
    int a = 7;
    double b = 3.14159;
    boolean c = false;
    String d = "woohoo!";
    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
    System.out.println(d);
  }
}
```

```
> java PrintlnTest
7
3.14159
false
woohoo!
```

Here's a puzzler...

How does **`System.out.println()`** accept different data types as parameters?

```
public class PrintlnTest
{
  public static void main(String[] args)
  {
    int a = 7;
    double b = 3.14159;
    boolean c = false;
    String d = "woohoo!";
    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
    System.out.println(d);
  }
}
```

```
> java PrintlnTest
7
3.14159
false
woohoo!
```

Here's a puzzler...

How does **`System.out.println()`** accept different data types as parameters?

```
public class PrintlnTest
{
  public static void main(String[] args)
  {
    int a = 7;
    double b = 3.14159;
    boolean c = false;
    String d = "woohoo!";
    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
    System.out.println(d);
  }
}
```

In other words, why doesn't this blow up? Can you construct a method that will accept different data types?

Method overloading

Java allows us to create methods with the same name but different parameter lists. This is useful when you want to perform similar operations on different types of data as well as different numbers of parameters. This is called method overloading.

Method overloading - different types

```
public class OverloadTest
{
  public static void main(String[] args)
  {
    int a = 7;
    boolean c = false;
    String d = "woohoo!";
    test(a);
    test(c);
    test(d);
  }

  public static void test(int x)
  {
    System.out.println("I am an integer.");
  }

  public static void test(boolean x)
  {
    System.out.println("Am I a boolean?  Yes?  No?");
  }

  public static void test(String x)
  {
    System.out.println("Aye, I'm a String and proud of it!");
  }
}
```

Method overloading - param list length

```
public class AvgTest
{
  public static void main(String[] args)
  {
    System.out.println(avg (10, 30, 20));
    System.out.println(avg(30,20));
  }

  public static double avg(double a, double b)
  {
    return ((a + b) / 2);
  }

  public static double avg(double a, double b, double c)
  {
    return ((a + b + c) / 3);
  }
}
```

Method overloading

When two or more methods have the same name, Java uses the number of parameters, the types of the parameters, and/or the order of the types of parameters to distinguish between the methods.

The method's name, type, and order of its parameters is called its signature. If you try to create two methods with the same signature, the compiler will let you know.

Method overloading

```
public class AvgTest2
{
  public static void main(String[] args)
  {
    System.out.println(avg(30,20));
  }

  public static double avg(double a, double b)
  {
    return ((a + b) / 2);
  }

  public static double avg(double a, double b)  // same signature
  {
    return ((a + b) / 2.0);                     // different logic
  }
}
```

Method overloading

```
public class AvgTest2
{
  public static void main(String[] args)
  {
    System.out.println(avg(30,20));
  }

  public static double avg(double a, double b)
  {
    return ((a + b) / 2);
  }

  public static double avg(double a, double b)  // same signature
  {
    return ((a + b) / 2.0);                     // different logic
  }
}
```

```
1 error found:
File: AvgTest2.java  [line: 13]
Error: avg(double,double) is already defined in AvgTest2
```

Method overloading

When two or more methods have the same name, Java uses the number of parameters, the types of the parameters, and/or the order of the types of parameters to distinguish between the methods.

The method's name, type, and order of its parameters is called its signature. If you try to create two methods with the same signature, the compiler will let you know.

The return type is not part of the signature. That is, you can't have two overloaded methods whose signatures differ only by the return type. Why? There's no way for Java to know from the method invocation which method was intended to be used, and it's not going to choose one at random, is it?

Method overloading

```
public class AvgTest3
{
  public static void main(String[] args)
  {
    System.out.println(avg(30,20));
  }

  public static double avg(double a, double b)
  {
    return ((a + b) / 2);
  }

  public static float avg(double a, double b)   // same signature
  {                                             // different return type
    return ((a + b) / 2);
  }
}
```

```
2 errors found:
File: AvgTest3.java  [line: 13]
Error: avg(double,double) is already defined in AvgTest3
File: AvgTest3.java  [line: 15]
Error: possible loss of precision
found   : double
required: float
```

Constructor overloading

Can we overload constructor methods? Of course!
Here's our favourite program, the CokeMachine...

```
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++;
  }
```

Constructor overloading

```
  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");
    }
  }
}
```

Constructor overloading

```
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++;
  }
```

Constructor overloading

```
public class SimCoke2
{
  public static void main (String[] args)
  {
    System.out.println("Coke machine simulator");
    CokeMachine2 cs = new CokeMachine2();
    CokeMachine2 engr = new CokeMachine2(237);
    CokeMachine2 chan = new CokeMachine2(42);
    CokeMachine2 library = new CokeMachine2(9000);
    cs.buyCoke();
    engr.buyCoke();
  }
}
```

```
> java SimCoke2
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 42 cans of Coke
Adding another machine to your empire with 9000 cans of Coke
Have a Coke
9 cans remaining
Have a Coke
236 cans remaining
```

Another vending-related opportunity

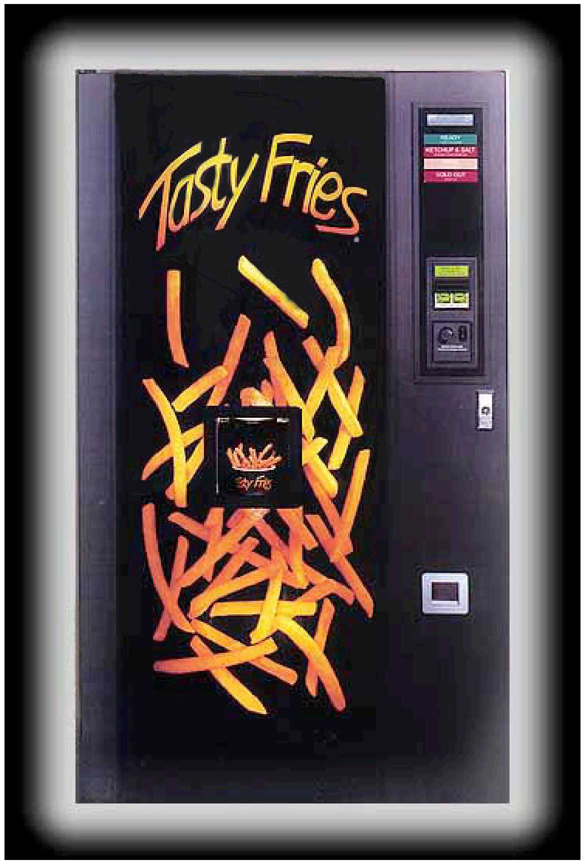
Let's say that you've been inspired by CPSC 111 and decide to create commercial vending-machine simulation software. To make this work, you'll need to accommodate vending machines beyond those that sell only Coca-Cola products.

For example, you may want to include...

Pizza machines...

Beer machines...

...and even French fry machines!



Another vending-related opportunity

Furthermore, while recognizing that a pizza machine is not the same as a beer machine is not the same as a Coke machine, you'll want to take advantage of the fact these two distinct types of vending machines have much in common. How can you do this? Here's one way...

Interfaces

Informally, we've used the word "interface" to refer to the set of public methods (for example, getters and setters) through which we interact with an object.

There's also a more formal use of the word interface in Java. A Java interface is a collection of constants and abstract methods.

Interfaces

An abstract method has no implementation...no body. It's just a method header followed by a semicolon. It specifies how one communicates with a method, but not what the method does.

Interfaces

```
public interface VendingMachine
{

  public void vendItem();

  public int getItemsRemaining();

  public int getItemsSold();

  public double getCashReceived();

  public void loadItems(int n);

}
```

We create an interface by using the reserved word `interface` in what would be the class header, if an interface were a class (which it's not...think of it more as the syntactic specification for a class).

Interfaces

```
public interface VendingMachine
{

  public abstract void vendItem();

  public abstract int getItemsRemaining();

  public abstract int getItemsSold();

  public abstract double getCashReceived();

  public abstract void loadItems(int n);

}
```

We could use the reserved word `abstract` in the abstract method headers, but we don't have to because the methods in an interface must be abstract.

Interfaces

An interface is not a class. An interface only prescribes what methods some other class or classes must have. That is, an interface specifies the headings for methods that must be defined in any class that implements the interface, but it doesn't say what happens inside the methods.

from Absolute Java by Walter Savitch.

Implementing an interface

A class implements an interface by providing method implementations for each of the abstract methods defined in the interface. A class that implements an interface uses the reserved word `implements` followed by the interface name in the class header.

Implementing an interface

```
public class CokeMachine2005 implements VendingMachine
{
  private int itemsRemaining;
  private int itemsSold;
  private double cashReceived;

  public CokeMachine2005()
  {
    itemsRemaining = 10;
    itemsSold = 0;
    cashReceived = 0;
    System.out.println("Adding another CokeMachine to your empire");
  }

  public int getItemsRemaining()
  {
    return itemsRemaining;
  }

  public int getItemsSold()
  {
    return itemsSold;
  }
```

Implementing an interface

```
  public double getCashReceived()
  {
    return cashReceived;
  }

  public void loadItems(int n)
  {
    itemsRemaining += n;
  }

  public void vendItem()
  {
    if (itemsRemaining > 0)
    {
      itemsRemaining = itemsRemaining - 1;
      itemsSold++;
      cashReceived = cashReceived + 1.25;
      System.out.println("Have a Coke");
      System.out.print(itemsRemaining);
      System.out.println(" cans remaining");
    }
    else
    {
      System.out.println("Sold out.");
    }
  }
}
```

Implementing an interface

```
public class FrenchFryMachine2005 implements VendingMachine
{
  private int itemsRemaining;
  private int itemsSold;
  private double cashReceived;

  public FrenchFryMachine2005()
  {
    itemsRemaining = 10;
    itemsSold = 0;
    cashReceived = 0;
    System.out.println("Adding another FrenchFryMachine to your empire");
  }

  public int getItemsRemaining()
  {
    return itemsRemaining;
  }

  public int getItemsSold()
  {
    return itemsSold;
  }
```

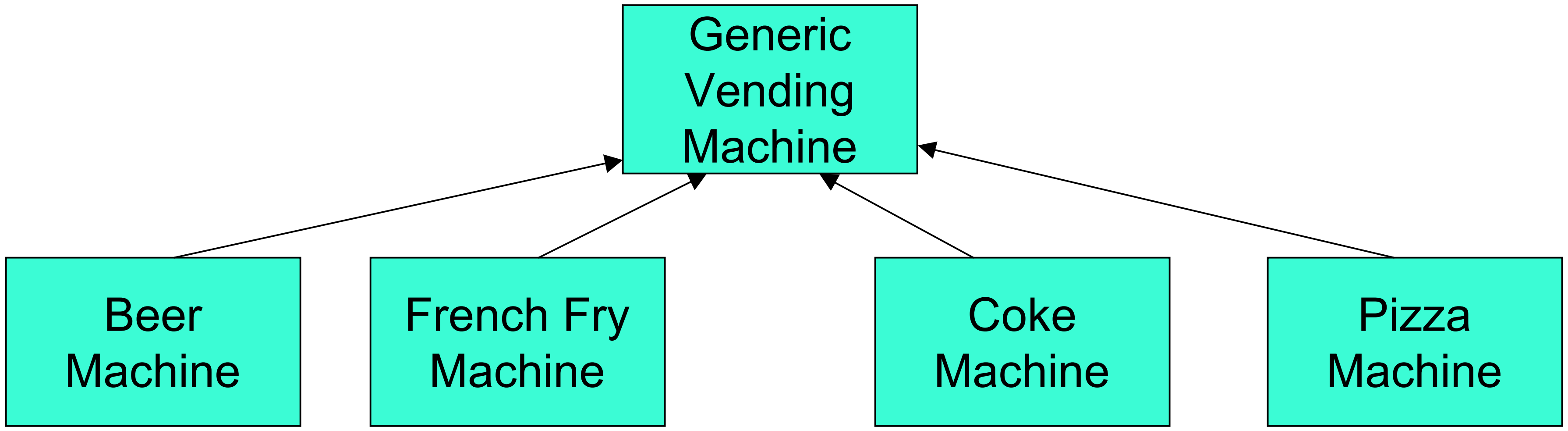
Implementing an interface

```
  public double getCashReceived()
  {
    return cashReceived;
  }

  public void loadItems(int n)
  {
    itemsRemaining += n;
  }

  public void vendItem()
  {
    if (itemsRemaining > 0)
    {
      itemsRemaining = itemsRemaining - 1;
      itemsSold++;
      cashReceived = cashReceived + 1.00;
      System.out.println("Have a nice hot cup of french fries");
      System.out.print(itemsRemaining);
      System.out.println(" cups of french fries remaining");
    }
    else
    {
      System.out.println("Sold out.");
    }
  }
}
```

Implementing an interface

For a class to implement an interface, it must provide a definition for all methods in the interface. If there are unimplemented methods, the class must be declared to be an abstract class, but that's a topic for another time.

What do these relationships look like?



Why this stuff is very very cool*

Because an object of a class that implements an interface is also an object of that interface type. That concept is the basis of an important object-oriented programming principle called polymorphism.

Polymorphism is derived from the word fragment *poly* and the word *morpho* in Greek, and it literally means "multiple forms".

*assuming you come from the planet Nerdtron

Why this stuff is very very cool

Polymorphism simplifies the processing of various objects in the same class hierarchy by using the same method call for any object in the hierarchy. We make the method call using an object reference of the interface. At run time, the Java Virtual Machine determines which class in the hierarchy the object actually belongs to and invokes the version of the method implemented for that class.

from Java 5 Illuminated by Anderson and Franceschi

Why this stuff is very very cool

```
public class SimCoke2005
{
  public static void main (String[] args)
  {
    VendingMachine foo1 = new CokeMachine2005();
    VendingMachine foo2 = new FrenchFryMachine2005();

    foo1.vendItem();
    foo2.vendItem();
  }
}
```

```
Adding another CokeMachine to your empire
Adding another FrenchFryMachine to your empire
Have a Coke
9 cans remaining
Have a nice hot cup of french fries
9 cups of french fries remaining
```

Why this stuff is very very cool

```
public class SimCoke2005
{
  public static void main (String[] args)
  {
    VendingMachine foo1 = new CokeMachine2005();
    VendingMachine foo2 = new FrenchFryMachine2005();

    foo1.vendItem();
    foo2.vendItem();
  }
}
```

The little foos may look like VendingMachine objects to you and me, but Java knows the difference and finds the appropriate method for each foo. That makes our programming job a lot easier to do. Why?

Why this stuff is very very cool

Because the alternative is to write lots of chunks of code that look like sort of like this (if they were written in English):

```
if we want to vend an item from foo1 and foo1 is a CokeMachine2005
then print "have a Coke" else
if we want to vend an item from foo1 and foo1 is a FrenchFryMachine2005
then print "have a cup of french fries" else
if we want to vend an item from foo1 and foo1 is a PizzaMachine2005
then....
```

As the number of classes within the same hierarchy grows, so does the size of the chunks of code represented above. Eeyow!

A simple bunny example

```
public interface Bunnies
{
  public void moveBunny(int direction);

}
```

A simple bunny example

```
public class BigBunny implements Bunnies
{
  private int x, y;
  private int carrots;

  public BigBunny()
  {
    x = 5;
    y = 5;
    carrots = 10;
  }

  public void moveBunny(int direction)
  {
    if (direction == 12)
    {
      y = y + 3;
      carrots = carrots - 2;
    }
```

A simple bunny example

```
      else if (direction == 3)
      {
        x = x + 3;
        carrots = carrots - 2;
      }
      else if (direction == 6)
      {
        y = y - 3;
        carrots = carrots - 2;
      }
      else if (direction == 9)
      {
        x = x - 3;
        carrots = carrots - 2;
      }
      else
      {
        System.out.println("Invalid direction");
      }
    }
  }
```

A simple bunny example

```
public class LittleBunny implements Bunnies
{
  private int x, y;
  private int carrots;

  public LittleBunny()
  {
    x = 5;
    y = 5;
    carrots = 10;
  }

  public void moveBunny(int direction)
  {
    if (direction == 12)
    {
      y = y + 1;
      carrots = carrots - 1;
    }
```

A simple bunny example

```
      else if (direction == 3)
      {
        x = x + 1;
        carrots = carrots - 1;
      }
      else if (direction == 6)
      {
        y = y - 1;
        carrots = carrots - 1;
      }
      else if (direction == 9)
      {
        x = x - 1;
        carrots = carrots - 1;
      }
      else
      {
        System.out.println("Invalid direction");
      }
    }
  }
```

Interface caution

Can’t construct interface

Can only construct objects that belong to some class