CS 4: Lecture 4
Monday, January 30, 2006

DEFINING CLASSES

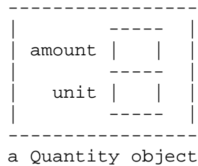
Recall that an object is a repository of data. _Fields_ are variables that hold the data stored in objects. Let's define a Quantity object.

```
class Quantity {
  public double amount;      // The numerical quantity.
  public String unit;        // The unit of measurement.

  public void contents() {
    System.out.println("I represent " + amount + unit + ".");
  }
}
```

This class is called a _class_definition_, because it defines the "Quantity" class. The three lines starting with "public void contents()" are called a _method_definition_, because they define a method called "contents". Last lecture, we saw that objects have methods associated with them. Methods perform actions on objects.

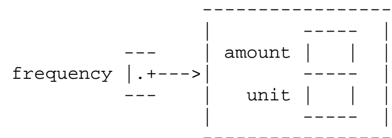
Now that we've defined the Quantity class, we can construct as many Quantity objects as we want. A Quantity object looks like this.



a Quantity object

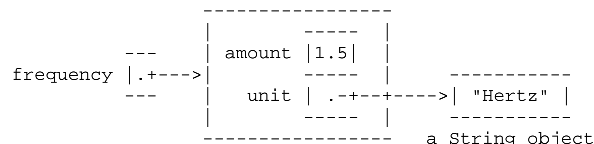
The small boxes in the object represent memory dedicated to storing two things: a floating-point number and a reference to a String. You can create a Quantity object the same way you create a String. (The following line of code might be in the "main" method.)

```
  Quantity frequency = new Quantity();
```



Now, "frequency.amount" and "frequency.unit" are both fields. We can assign values to them, just like normal variables.

```
  frequency.amount = 1.5;                // Set frequency's fields.
  frequency.unit = "Hertz";
```



a String object

Fields are also known as _instance_variables_.

A Java program can call the method "contents" that we defined. The following line of code is called a _method_call_.

```
  frequency.contents();          // Print what "frequency" represents.
  frequency.contents();          // Print it again.
```

The output is:

```
  I represent 1.5 Hertz.
  I represent 1.5 Hertz.
```

What just happened? When Java executes the first line, it _jumps_ to the method contents() in the Quantity class, and executes the code inside. When Java reaches the end of the code in the contents() method, it _returns_ to the method call and continues where it left off, executing the next line of code after the method call. Java leaves a sort of "bookmark" to remind it what line of code it was at when it made the method call, so that when it returns, it can continue reading where it left off.

When Java executes the second "frequency.contents()", the same thing happens over again.

Why is it that, inside the definition of contents(), we don't have to write "frequency.amount" and "frequency.unit"? When we invoke "frequency.contents()", Java remembers that we are calling the contents() method _on_ the object that "frequency" references. So we can just write "amount" inside the contents() method, and Java knows which object's amount to use. If you invoke "wavelength.contents()" instead, then inside the contents() method, "amount" refers to wavelength.amount instead.

Note that method definitions and method calls always have parentheses after them, whereas fields never do. That's how you tell them apart.

Formal and Actual Parameters

A method is not limited to manipulating just one object. Let’s look at a method called "triple" that manipulates two.

```
class Quantity {
  // Include all the stuff from the previous definition of Quantity here.

  public void triple(Quantity original) {
    amount = 3.0 * original.amount;
    unit = original.unit;
  }
}
```

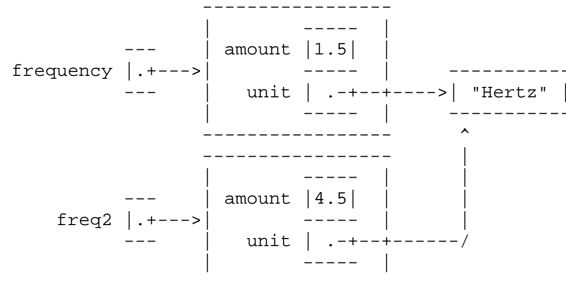
Now, suppose the Java program above that created "frequency" continues with the following code.

```
  Quantity freq2 = new Quantity();
  freq2.triple(frequency);
```

The definition of the "triple" method has a special variable called "original", which is called a _formal_parameter_ of triple. The code "freq2.triple(frequency)" calls the method "triple" with the _actual_parameter_ "frequency". During the method call, Java substitutes the actual parameter for the formal parameter. It’s just like in math: if you define

$$f(x) = x^3 ,$$

then f(2) = 8 because you substitute the actual parameter 2 for the formal parameter x. Analogously, Java substitutes "frequency" for "original" during the method call, so when the method call finishes, the objects look like this.



Suppose we triple "freq2".

```
  Quantity freq3 = new Quantity();
  freq3.triple(freq2);
  freq3.contents();
```

This creates a third Quantity object with thrice the value of freq2.
The output is:

```
  I represent 13.5 Hertz.
```

Return Values and Static Methods

Let’s write a method that acts like a mathematical function.

```
class Quantity {
  public static double cube(double x) {
    System.out.println("I’m cubing " + x);
    return x * x * x;
  }
}
```

Instead of "void", the "cube" method has "double" in its declaration, which means that it _returns_ a double. In other words, it passes a floating-point number back to the caller. The method ends with a line of code called a _return_statement_ that tells the method what value to return.

Now we can write

```
  double result;
  result = Quantity.cube(1.5);
```

What does the last line do? First, it calls the "cube" method, substituting 1.5 for x. The "cube" method prints "I’m cubing 1.5", then calculates the cube of 1.5, which is 3.375, and returns it. Java returns to the method call, and assigns the return value 3.375 to "result".

Notice the "static" keyword in the "cube" method definition. This means that "cube" does _not_ operate on a Quantity object. It doesn’t involve any object at all; it just operates on doubles. So when we call it, we don’t call it on an object; we call it on the class: "Quantity.cube(1.5)".

Java has many built-in functions. The Math class has particularly useful ones.

```
  double root = Math.sqrt(3.0);    // Compute the square root of 3.
```