CS 4: Lecture 3
Wednesday, January 25, 2006

OBJECTS
=======
An _object_ is a repository of data. For example, if MyList is a ShoppingList object, MyList might record...your shopping list.

A class is a type of object. Many objects of the same class can exist; for instance, MyList and YourList may both be ShoppingList objects. In other words, they're both of the class ShoppingList.

There are two ways to get some classes to play with:
(1) Use one defined by somebody else. Java has tons of pre-defined classes you can use. Many come in the "Java standard library" provided with every Java compiler.
(2) Define one yourself.

For example, Java has a built-in class called String. You can declare a String variable like this.

```
  String myString;
```

However, this does _not_ create a String object. Instead, it declares a variable--a space in memory--that can store a _reference_ to a String object. I draw it as a box.

```
          ---
  myString | |                  <-- This box is a variable (not an object).
          ---
```

Initially, myString doesn't reference anything. You can make it reference a String object by writing an assignment statement. But how do we get ahold of an actual String object? You can create one.

```
  myString = new String();
```

This line performs two distinct steps. First, the phrase "new String()" is called a _constructor_. It constructs a brand new String object.

```
          ---     ------
  myString | |     |    | a String object
          ---     ------
```

Second, the assignment "=" causes myString to _reference_ the object. You can think of this as myString pointing to the object.

```
          ---     ------
  myString |.+---->|    | a String object
          ---     ------
```

Strings are designed to store sequences of characters, like the letters and symbols you type on your keyboard. Strings are a special class of object in Java, because you can construct one just by writing something in quotation marks. The following code declares a String variable, creates a String, and assigns the latter to the former.

```
  String s1 = "cs 4";

             ---     ----------
        s1 |.+---->|  cs 4  |
             ---     ----------
```

```
  String s2 = s1;

             ---     ----------     ---
        s1 |.+---->|  cs 4  |<----+.| s2
             ---     ----------     ---
```

Now s1 and s2 reference the same object. What if we'd prefer to have a copy of the object?

```
  s2 = new String(s1);

             ---     ----------     ---     ----------
        s1 |.+---->|  cs 4  |  s2 |.+---->|  cs 4  |
             ---     ----------     ---     ----------
```

Now we've seen three String constructors:
(1) new String() constructs an _empty_string_--it's a string, but it contains no characters.
(2) "cs 4" constructs a string containing the characters cs 4.
(3) new String(s1) takes a _parameter_ s1. Then it makes a copy of the object that s1 references.

Think about that last one. When Java sees String(s1), it does the following things, in the following order.
- Java looks inside the variable s1 to see where it's pointing.
- Java follows the pointer to the String object.
- Java reads at the characters stored in that String object.
- Java creates a new String object that stores a copy of those characters.

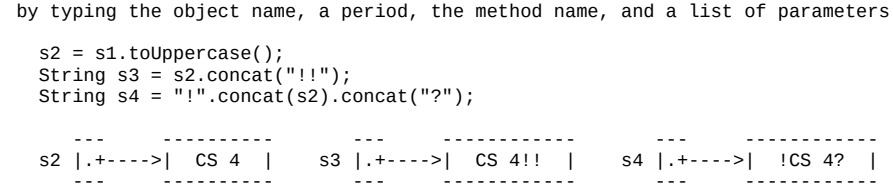
Constructors always have the same name as their class, except the special constructor "stuffinquotes". That's the only exception.

Observe that "new String()" can take no parameters, or one parameter. These are two different constructors--one that is called by "new String()", and one that is called by "new String(s1)". (Actually, there are many more than two--check out the online Java API to see all the possibilities.)

METHODS
=======
A _method_ is a procedure that operates on an object or a class. A method is associated with a particular class. For instance, addItem might be a method that adds an item to any ShoppingList object. You've already seen a method called println that prints things to the screen.

concat is a method that concatenates two Strings together. A method is invoked by typing the object name, a period, the method name, and a list of parameters.

```
  s2 = s1.toUppercase();
  String s3 = s2.concat("!!");
  String s4 = "!".concat(s2).concat("?");
```



Now, here's an important fact: when Java executes the line

```
  s2 = s1.toUppercase();
```

the object that s2 referenced did _not_ change. Instead, s2 itself changed to reference a new object. Java wrote a new "pointer" into the variable s2, so now s2 points to a different object than it did before.

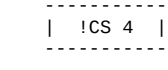
In fact, String objects are _immutable_--once they've been constructed, their contents never change. If you want to change a String object, you've got to create a brand new String object that reflects the changes you want. This is not true of all objects; most Java objects let you change their contents.

Let's look at the method call s2.concat("!!").
- The concat method takes one parameter, the String you want to concatenate on the end. Some methods take two or more parameters; some take none.
- It also operates _on_ an object, namely the String that s2 references.
- It also _returns_ a value. The value it returns is a reference to a new String.

After the method toUppercase executed, we assigned its return value to s2, which makes s2 point at the new String. But that's not the only thing we can do with a return value. Take another look at this:

```
  "!".concat(s2).concat("?")
```

First, "!".concat(s2) is a method call that returns a String object.



But we don't assign it to a variable at all. Instead, we immediately call a method on it! That method is concat("?"). This creates another new object, which we assign to s4.