Using Classes

Most books teach how to use classes with the String class. However, the String class is designed in a deceiving manner, so that the user is hidden from many details that are essential to understand with respect to using classes. Thus, this lecture will use a separate example (the GiftCard class, which is NOT a pre-written class in Java) to illustrate how to use classes.

Before we get into the specifics of a class, let's quickly lay out the philosophical reason for object oriented programming and classes.

As programs get large, it's impossible for any single person to understand all of the complexity behind them. In order for programmers to be able to do very complicated things, such as write video games, it's impossible for them to deal with everything on a very low level. Instead, it's advantageous for them to be able to use "objects" that are flexible but EASY to use.

A natural object that comes to mind is a cell phone. It's a complicated device, but once you have one, you can VERY EASILY use it to do lots of different things without understanding how IT works, OR how the buttons on it really work. You just have to understand how to "create" one (go to the store and buy one =)) and then you have to understand how to implement tasks that are specific to the cell phone (such as looking up a number in your directory).

Note that you can't do anything you want with your cell phone. For example, you can't search all of your text messages for a specific string. (Well, at least I can't on my phone =)) If you knew how to mess with the inside of the phone, you could probably make the changes to do that, but since you don't have that access, you can ONLY carry out the different "methods" that the cell phone maker has provided for you.

Furthermore, note that each of these methods can be carried out on ANY cell phone object and that it's different to call a number from your cell phone than my cell phone.

The reason it's advantageous to define a class like the CellPhone class is that once it is written, others can create objects of the class and USE those objects (with the methods specific to the CellPhone class) without spending a whole lot of time to understand HOW a CellPhone is organized and stored.

In C, whenever you call a function, you didn't need to know HOW the function worked either, BUT you did need to know how all relevant data was stored.

In an object oriented language, when you use a class, there's NO NEED for you to know HOW the data in an object of that class is stored. Instead, you can just create an object and USE it without knowing how the data for the object is stored. This is known as DATA ABSTRACTION, and is the primary feature of object oriented languages that is lacking from imperative languages (like C).

CellPhone example with Pseudocode

Here is an example of some pseudocode that will look similar to Java code that utilizes CellPhone objects:

```
CellPhone one = new CellPhone(Sprint, "John", "Doe");
CellPhone two = new CellPhone(Verizon, "Jenny", "Doe");

one.addDirectory(Jenny, 8675309);
one.call(Jenny);
two.ring();
two.answer();
```

In this example, we first "create" two cell phone objects, which are referred to by one and two, respectively. (These are references to the actual objects in Java.)

Then, Jenny's number is added to the phone object one. Following this, the phone object one (John) calls Jenny.

Jenny's phone (two) rings. Then, her phone is answered.

Hopefully it should be clear that each of these actions: adding to a directory, calling, ringing and answering, can ONLY be done on CellPhone objects.

addDirector(Jenny, 8675309) would make no sense because we would have no idea WHICH phone directory to add that entry to.

Similarly, we must call from a CellPhone object, a CellPhone object (and not a coffee table) rings, and CellPhone objects can be answered, which once again, coffee tables can NOT be.

Using a Class in Java: Calling the Constructor

A typical class defines the characteristics of an object. In order to make use of that class, first one has to create an object of that class.

The only way to create an object in Java is to call a constructor for a class.

In the GiftCard class, the following constructors exist:

```
// Creates a GiftCard object owned by first last with no
// initial balance.
public GiftCard(String first, String last);

// Creates a GiftCard object owned by first last with an
// initial balance of amt dollars.
public GiftCard(String first, String last, double amt);
```

Notice that the name of a constructor is ALWAYS the same name as the class. Furthermore, constructors can take in zero or more parameters like other methods. These parameters are used to create the object.

In order to *call* a constructor, the key word new must be used. Here is a typical call to the GiftCard constructor:

```
Giftcard macys = new GiftCard("Arup", "Guha", 200);
```

This call creates a GiftCard object whose owner is Arup Guha that has an initial balance of 200 dollars.

Using a Class in Java: Calling Instance Methods

Once an object is created, then that object can be manipulated or acted upon by the instance methods in the class. In particular, instance methods define the possible behaviors of objects of a class. Here is a listing of the instance methods in the GiftCard class:

```
// Spends amt dollars from the current object if the current
// object has adequate funds and returns true. If not, false is
// returned and no money is spent.
public boolean spend(double amt);

// Transfers ownership of the current object to first last.
public void transfer(String first, String last);

// Returns a String representation of the current object.
public String toString();

// Returns a negative integer if the current object has a smaller
// balance than g, 0 if they have the same balance, and a
// positive integer if it has a greater balance than g.
public int compareTo(GiftCard g);

// Adds amt dollars to the current object.
public void addAmount(double amt);
```

Here is how we can add $10.99 to the GiftCard object we just created:

```
macys.addAmount(10.99);
```

addAmount is void, so we want to call it on a line by itself.

Since addAmount is an instance method, we NEED to precede a call to it with the name of an object followed by a dot. This is how ALL instance methods are called.

Finally, we must follow the rules and pass in a double to this method. The effect of the method is to add 10.99 to the balance of the GiftCard object referenced by macys.

It's also important to note that we can declare multiple objects of the same class. Let's define another GiftCard object:

```
GiftCard sears = new GiftCard("John", "Doe", 50);
```

Now, our picture roughly looks like this:

```
macys ----------------→ [ Arup Guha 210.99 ]
sears -----------------→  [ John Doe 50.00 ]
```

If we execute the line,

```
sears.spend(19.99)
```

then our picture would be

```
macys ----------------→ [ Arup Guha 210.99 ]
sears -----------------→  [ John Doe 30.01 ]
```

We must specify WHICH GiftCard object to run spend on.

Using the methods that we have, we could determine which GiftCard object had more money in it in the following fashion:

```
if (macys.compareTo(sears) > 0)
  System.out.println("Macys has more money!");
else if (macys.compareTo(sears) == 0)
  System.out.println("Both have the same money!");
else
  System.out.println("Sears has more money!");
```

Given the current state of the objects in our trace, the following would be printed:

```
Macys has more money!
```

Now, consider executing the line:

```
macys.transfer("Sarah","Pierce");
```

Now, our picture looks like:

```
macys ----------------→ [ Sarah Pierce 210.99 ]
sears -----------------→  [ John Doe 30.01 ]
```

In this manner, we can continue to utilize any number of GiftCard objects we want. We are limited by the different public methods that are given to us in the class, as was mentioned in the cell phone analogy before.

Use of references in Java

It was alluded that the names "macys" and "sears" are really *references* to the two GiftCard objects in the previous examples. The pictures on the previous page indicate that as well. Now, let's examine the user of references.

Consider the following line of code:

GiftCard mycopy = macys;

Because mycopy (just like macys and sears) are *references* the picture ACTUALLY looks like this now:

```
macys ---------------→ [ Sarah Pierce 210.99 ] ←------mycopy
sears ----------------→  [ John Doe 30.01 ]
```

Thus, there are STILL only two objects, but both macys and mycopy refer to the SAME object.

In general, the line of code

a = b;

where a and b are non-primitives makes a reference the SAME object that b is referencing. Although this seems a bit weird, it's the same thing that happens with pointers in C, and it comes in amazingly handy and makes many things work very smoothly.

Now, consider running the line:

mycopy.spend(199.50);

Here is the new picture of what has happened:

```
macys ---------------→ [ Sarah Pierce 11.49 ] ←------mycopy
sears ----------------→  [ John Doe 30.01 ]
```

Thus, even though macys isn't explicitly shown on the last line of code, the object to which macys is referring HAS CHANGED!!!

Now consider the following line:

macys = new GiftCard("Donald", "Trump", 100000000);

Here is the new picture:

```
macys ---------------→ [ Donald Trump 100000000.00 ]
                       [ Sarah Pierce 11.49 ] ←------mycopy
sears ----------------→  [ John Doe 30.01 ]
```

All that line of code does is create a new object (the constructor does this), then macys is set to refer to the newly created object.

In essence, ANY time a non-primitive is on the left-hand side of an assignment statement, that means that it will reference WHATEVER the reference on the right-hand side of the assignment statement is referencing.

The key is that references are separate from objects. Constructor calls create objects. References point to them.

Second Example of Using a Class: Time

The following is the listing of the public methods in a class called Time. This class describes an object that stores a unit of time. We will use it in an example where the user enters how long his/her tasks for the day took and see if that exceeds a particular threshold. The listing below only contains the method calls needed to complete the given task.

```
// Creates a time object that is h hours
// and m minutes long.
public Time(int h, int m);

// Returns a negative integer if the current
// object is shorter than time2, 0 if it is
// equal to time2 and a positive integer if
// it is longer than time 2.
public int compareTo(Time time2);

// Returns a time object that is the as long
// as the sum of the current object and
// time2.
public Time addtime(Time time2);
```

Time Class Example Program

```
public class Activities {

  public static void main(String[] args) {

    Scanner stdin = new Scanner(System.in);
    System.out.println("Enter # hours.");
    int hours_worked = stdin.nextInt();

    Time min = new Time(hours_worked,0);

    Time worked = new Time(0,0);

    int ans = 1;
    while (ans == 1) {

      System.out.println("Enter task time");
      int h = stdin.nextInt();
      int m = stdin.nextInt();
      Time temp = new Time(h,m);

      worked = worked.addtime(temp);

      System.out.println("Again?(1/0)");
      ans = stdin.nextInt();
    }

    if (worked.compareTo(min) >= 0)
      System.out.println("Did enough!");
    else
      System.out.println("Work more!");
  }
}
```