Section 1 - Introduction and Resources

6A.1.1 Overview

OOP

This week we will introduce the most important concept in modern programming languages: ***Object-Oriented Programming (OOP)***. We start learning ***OOP*** by introducing ***classes***. Then we introduce methods that are defined inside classes. There are two kinds of methods and data in classes: instance and static. We'll see how they differ, and when to use each.

Without further ado, let's dive in.

Section 2 - Classes You Define

6A.2.1 Defining a Class

We have used ***classes*** that were defined for us by Java: **JOptionPane, String, Integer, Math,** etc. We used them to declare ***objects*** or to call ***methods***. Now we wish to define our own ***classes***. We have been doing this all along, of course, but we only defined one ***class,*** giving it a name like **Foothill** or **Sample,** and allowing it to stand alone in our program. Sometimes I referred to this single class as the "main class" and, not coincidentally, it has always contained a **main()** ***method.*** Every Java application needs one class that contains a **main()** method.

We are now going to accompany our main class with one or more new ***classes.*** We will define these new classes in the same **.java** file, below (or above) the main class.

When you think of defining ***classes***, you should think of defining your own custom-made data type. **Floats, ints, doubles,** even **Strings**: these are all boring. We want to define a new class that has a little flash and appeal. And what could be more entertaining than a pet? (I don't know, I don't have one, but all my neighbors do.)

So we will create a new data type, called **Pet**, and we will decide that a **Pet** will always have four things.

- A name
- An owner
- A weight
- A sound

And with that, we can create a class definition:

```
class Pet
{
   // member data
   String petsName;
   String ownersName;
   double weight;
   AudioClip soundOfPet;
}
```

This is the basic idea. We use the word **class** followed by the name we want to give this new data type. Then we open a brace and place the definition of the ***class*** between the opening brace and the closing brace.

We build our ***class*** from any old "spare parts" we have lying around: **Strings**, **doubles** and **AudioClips**. Wait, what's an **AudioClip**? We haven't seen any data of that type before. I'm sure you can guess that it is a pre-defined ***class*** in Java that represents a sound file. This will store the sound of our pet.

And, once we define this new ***class***, **Pet**, how do we ***use it***? Since it is a data type, like any other, we use it by declaring ***variables*** of that type. These ***variables*** behave more like **Strings** than like the ***primitive types***. You'll recall that the **String *class*** was a little more complex than the ***primitive*** **int** or **double** type. We follow the pattern established for **Strings** when we first introduced them, namely, we will first declare a ***reference*** of the new type, and then we ***instantiate a particular object*** using the keyword **new**. Here's an example of the program minus the details. Notice that we first present the ***main class***, **Foothill**, followed by our ***new class*** **Pet**. Inside **Foothill** we declare **Pet *objects***. Look:

```
import java.net.*;
import java.applet.*;
import java.util.*;
import javax.swing.*;

public class Foothill
{
   public static void main(String[] args)
   {
      // declare a couple Pet references.
      Pet mikesDog, noisyDog;

      // instantiate a new Pet
      mikesDog = new Pet();

      // ... a little later on, perhaps make an assignment statement
      noisyDog = mikesDog;
   }
}

class Pet
{
   // member data
   String petsName;
   String ownersName;
   double weight;
   AudioClip soundOfPet;
}
```

Things to observe:

- The **Pet** *class* definition comes after the **Foothill** *class* (which is our main *class*). It could come before the *class*, as well. It cannot be nested within the Foothill *class* (or at least not yet).
- We use the **Pet** *class* from inside the **Foothill** *class methods*. In this case, we are using the **Pet** *class* inside the **main()** *method* of **Foothill**.
- There is only one **main()** *method*. **Pet** should (normally) not have a **main()** of its own. Most ***classes*** do not have **main()**s in them.
- Although we define the **Pet** *class* outside the main **Foothill** *class*, we create **Pet** *objects* (variables) inside **Foothill's main()** method. In this regard ***classes*** are a lot like *methods*, because if you recall, with ***methods***, we also make the ***definition*** in one place and ***invoke*** the ***methods*** in another place.
- Creating **Pet** *objects* is a two step process. First you *declare the reference*. Then you *instantiate an object* and point to it with that reference:
- `Pet mikesDog;`
- `mikesDog = new Pet();`
- Sometimes the two statements are combined into one:
- `Pet mikesDog = new Pet();`
- We can assign one **Pet** *reference* to another:
- `noisyDog = mikesDog;`
- Unlike *primitive* data (but like **String** data) this assignment does not normally copy all of the individual elements of the **Pet** *objects*. The assignment merely points the **noisyDog** *reference* to **mikesDog** so that both *references* point to the same **Pet** *object*.
- The four pieces of data that comprise our **Pet** are called ***member data***. There are other common names for these items:
    - ***Member Data***
    - ***Fields***
    - ***Instance Variables***
    - ***Members***
- The ***fields (***or ***members*** or ***variables)*** of the ***class*** can be either **public** (default) or **private**. If there is no keyword in front of the data in the ***class*** definition, the data is assumed to be **public**, and can be modified throughout your program.
- The class **Pet** does not have the keyword **public** in front of it. This is because we are defining it inside the same **.java** file as our main class, **Foothill**. Since **Foothill** is the main class in the file **Foothill.java**, it has the unique privilege of being declared **public**, while all the other classes defined in that file must be declared without the **public** keyword. If you wanted to, you could create a new ***class*** in its own file, separate from **Foothill.java**. You would name the file **Pet.java**. In that case you would place the **public** keyword in front of **class Pet**.

This is a good start. Let's see an entire program using **Pet** objects.

Section 3 - A Simple Class: Pet

6A.3.1 Dereferencing the object

Now that we have some **Pet *objects***, how do we manage the individual data in those ***objects***? We do it through the ***reference***. If we want to manipulate, read or display the name of the **Pet** stored in **mikesDog**, we would access that datum by using the ***name of the object*** followed by a **period**, followed by the **name of the field** that we want to access (in this case the **petsName** field):

```
mikesDog.petsName = "Jerry";
```

or, accessing the same field to do something different:

```
System.out.println( mikesDog.petsName );
```

So you see, this is how we get at the primitive data that lies deep within each object of our class.

Another way to think of this is as follows. Whenever we instantiate a new **Pet** object, it is imbued with four data fields (as we described in the class definition: two **Strings**, a **double** and an **AudioClip**). If the object is called **garfield** because we declared the reference like so:

```
Pet garfield;
```

then we use the word **garfield** to get at the **weight** or **petsName** fields through the notation: **garfield.weight** or **garfield.petsName**. Of course, **garfield.petsName** is probably the string "**Garfield**" or "**Mr. Garfield**". Don't confuse this **String** stored inside the object (which can be anything we want and might change as the program progresses) with the object reference, **garfield**. This is a repeat, but I'll state it again here:

Terminology

Using the object name to get at the field (or member) data this way is called ***dereferencing*** the field (or member). We also say we are using the object to ***dereference*** the field or member. Sometimes we say we are ***dereferencing*** the object to get at the field. It all means the same thing. We are using the object with a period followed by a field name.

6A.3.2 The Noisy Dog

With that introduction we can look at a complete (silly) program. You will either have to create a project that calls its main class **Foothill**, or else you'll have to rename the **Foothill** class so that it matches your main class (the name of the .java file).

```
import java.net.*;
import java.applet.*;
import java.util.*;
import javax.swing.*;

public class Foothill
{
   public static void main(String[] args) throws Exception
   {
      Pet mikesDog, noisyDog;
      mikesDog = new Pet();

      // fill in the fields or members of mikesDog
      mikesDog.petsName = "Jerry of Westhampton";
      mikesDog.ownersName = "Michael Loceff";
      mikesDog.weight = 8.5;

      // setting the soundOfPet member of mikesDog
      URL url = new URL("http://www.doghause.com/sounds/dogbark4.wav");
      AudioClip bark = Applet.newAudioClip(url); // don't worry about this
      mikesDog.soundOfPet = bark;

      // do some things with mike's dog
      JOptionPane.showMessageDialog( null,
            mikesDog.petsName + " is owned by "
            + mikesDog.ownersName);
      mikesDog.soundOfPet.loop();   // start the barking
      JOptionPane.showMessageDialog( null,
            "Click OK when you're sick of hearing this.");
      mikesDog.soundOfPet.stop();   // stop the barking

      // assign one reference to another
      noisyDog = mikesDog;

      // the noisyDog reference, but it still points to the same object
      JOptionPane.showMessageDialog( null,
            noisyDog.petsName + " sounds like this ..."
            + "\n hit return when you want to hear some noise.");
      noisyDog.soundOfPet.loop();
      JOptionPane.showMessageDialog( null,
            "Click OK to end this madness.");
      noisyDog.soundOfPet.stop();
   }
}

class Pet
{
   // member data
   String petsName;
   String ownersName;
   double weight;
   AudioClip soundOfPet;
}
```

Two things about this program:

1. There is no way to convey the run of the program by pictures. You'll have to copy and run this yourself. Be sure to have your computer volume turned down, and have some aspirin ready.
2. This is not really my dog. I don't even own a dog.

This program is unrealistic in a couple ways. First, all of the **Pet** data is **public** (by default). This enables the ***main class*** to access the data directly. While that sounds harmless, actually it is quite dangerous. One of the advantages of surrounding data with a ***class*** is that the ***class*** can protect the data from unskilled users or clients who might inadvertently damage the data. A second unrealistic aspect of this example is that there are no ***methods*** (i.e., member functions) defined inside **Pet**. This is related to the first issue in that, if we had ***methods***, we could declare all the ***fields*** to be **private** and then design the ***methods*** so they could be used by the client to modify the data. By forcing the client to access the ***instance data*** through ***public methods*** (rather than giving it access to ***instance data*** directly as we did here) we could have protected the class data against a malicious client.

6A.3.3 From Dogs to Ducks

Of course we could have many different kinds of pets. How about a duck? Just ***instantiate*** another **Pet** and this time give it the characteristics of a duck. You really want to turn your volume down for this one, especially, if you're at work.

```
import java.net.*;
import java.applet.*;
import java.util.*;
import javax.swing.*;

public class Foothill
{
   public static void main(String[] args) throws Exception
   {
      Pet mikesDuck;
      mikesDuck = new Pet();

      // fill in the fields or members of mikesDuck
      mikesDuck.petsName = "Daffy";
      mikesDuck.ownersName = "Michael Loceff";
      mikesDuck.weight = 1.88;

      // setting the soundOfPet member of mikesDuck:
      URL url = new URL("http://www.doghause.com/sounds/duck.wav");
      AudioClip quack = Applet.newAudioClip(url);
      mikesDuck.soundOfPet = quack;

      // do some things with mike's duck
      JOptionPane.showMessageDialog( null,
            mikesDuck.petsName + " is owned by "
            + mikesDuck.ownersName);
      mikesDuck.soundOfPet.loop();
      JOptionPane.showMessageDialog( null,
            "Click OK when you're sick of hearing this.");
      mikesDuck.soundOfPet.stop();
   }
}

class Pet
{
   // member data
   String petsName;
   String ownersName;
   double weight;
   AudioClip soundOfPet;
}
```

6A.3.4 Instance Variables

The four members inside the Pet class are called ***instance variables*** for the class.

Every time a **Pet** object is ***instantiated*** in **main()** or some other client, *that instance gets **its own copy** of all these fields:* **petsName**, **ownersName, weight** and **soundOfPet**. That's why they are called ***instance variables*** - a new set is created every time a new ***instance*** of **Pet** is created.

6A.3.5 Static Class Variables

If the keyword **static** had been placed in front of any of the fields, then that field would be a ***static class member***. The result would be that new **Pet** ***instances*** would not get their own copy of those **static members.** All the objects of the class **Pet** would share one copy of a ***static class variable***. We don't have any examples in class **Pet** of ***static class variables*** since all the ***data members*** are ***instance variables***.

Section 4 - Classes Redefined

6A.4.1 Another Perspective

Since this is all new, I'm going to say the same thing again differently and with a new example. Hopefully, if you were hazy on any of what I just presented, this will help solidify things for you.

We have seen and used ***classes*** before. Remember this excerpt from a couple weeks ago:

```
String thxMom;                          // declare the reference
thxMom = new String("Thanks, Mom!");  // create the object
```

This is an example of our using a pre-defined ***class***, **String**, in **main().** We declared a **String** ***reference***, **thxMom**, and then created an ***object*** for that ***reference*** to point to. We also stored the ***literal*** **"Thanks, Mom!"** in the ***object***.

In the same way we ***declared*** and ***instantiated*** a **JFrame** object some time back. We used the class **JFrame** to declare a ***reference***, and then ***instantiated*** an object for that ***reference***:

```
JFrame frmMyWindow = new JFrame("Transporter Room");
```

which is really a combination of the two separate statements:

```
JFrame frmMyWindow;                           // declare JFrame reference
frmMyWindow = new JFrame("Transporter Room"); // instantiate JFrame object
```

Here's another example of a class: **Scanner**. We used this when we needed to do some console input:

```
Scanner inputStream;                     // declare a Scanner reference
inputStream = new Scanner(System.in);    // instantiate a Scanner object

// then later use the object to "dereference" the nextDouble() method:
x = inputStream.nextDouble();
```

The only difference between these ***classes*** and the **Pet *class*** of the last example is that *we* (you and I) defined the **Pet *class***, whereas some geniuses at Sun Microsystems defined the **Scanner, JFrame** and **String *classes*.** We don't see the definitions of those ***classes***, we just use them. With **Pet**, we see the definition because we created it.

Compare the above to our use of the **Pet** class in **main()**:

```
Pet mikesDuck;
mikesDuck = new Pet();
```

Do you see? Same deal!

The word ***class*** refers to the blueprint, or data type.

The word ***object*** refers to a particular ***instance*** of the class.

In our second example, we'll again define a class that contains only data, no methods, and make it clear that the data has some unifying purpose.

Our class will be all basic data that is needed to define an **Employee**.

6A.4.2 An Employee Class

Class **Employee** (capital **E**) will reflect certain aspects of a real ***employee*** (lowercase e). Consider, for the sake of our example, that an **Employee** consisted of three numbers, bundled together. Each number represented something about the ***employee***. The first number is the social security number. The second is the hourly wage. The third is the employee's age. Now we construct a new data type as follows:

```
class Employee
{
   // member data for the class
   public long socSec;
   public double wage;
   public short age;
}
```

This describes our new data type. You can see that this definition begins with the keywords **class**. That means that we are defining a new type - not an object yet. The class is the new data type.

The three data inside the new data type are called the ***data members*** or ***members*** or ***variables*** or ***fields of the class***. As we hinted at in earlier lectures, because the keyword **static** is missing from their definition, they are called ***instance variables.*** If the word **static** appeared before them they become ***static class variables***.

I'm going to repeat this important concept from the last section:

Instance Variable vs. Static Class Variable

Every time an **Employee** object is ***instantiated*** in **main()** or some other client, that instance gets **its own copy** of all these fields: **socSec**, **wage** and **age**. That's why they are called ***instance variables*** - a new set is created every time a new ***instance*** of **Employee** is created.

If the keyword **static** were placed in front of any of the fields, then that field would be a ***static class member***. The result would be that new **Employee *instances*** *would not get their own copy* of those **static members.** All the objects of the class **Employee** would share one copy of the ***static class variables***. We don't have any examples in class **Employee** of ***static class variables*** since all the ***data members*** are ***instance variables***.

This class demonstrates the creation of a user-defined data type made up of exactly the kinds of primitive data that we need. The idea that we can bundle different data into a user-defined type is called ***encapsulation***.

Now that we have described a new type, we can declare a ***reference*** of this type. We do this inside a method, just like a local variable, or at the top of a different class, making it a member of the other class**.**

```
Employee birkoff, walter;
```

This defines two **Employee** variables. Variables for objects like **Employee** are not called objects -- yet. They are called ***references***. We still don't have an object yet! All we have is something capable of pointing to an object.

So we have to go one step further. We must ***allocate*** an actual object and point to it with this ***reference***:

```
walter = new Employee();
birkoff = new Employee();
```

Now we finally have an **Employee** object.

1. We defined the ***class***.
2. Then we used the class name to declare a ***reference*** for the class,
3. Then we used the keyword **new** to instantiate an ***object*** of the class and point to it using the ***reference***.

Understand, please, that before we had used the **new** operator, there were no objects of the type **Employee**. We only had the notion (blueprint) of what this type was and we had a couple ***references*** capable of pointing to such objects if they ever got created.

It is like describing what the 2083 Nissan Sentra will look like in the year 2083 without actually manufacturing the car yet. After we created the object with **new**, however, we have made a car (or, in this case, **Employee**).

Terminology

Declaring and **new**-ing an object of a given class is often termed ***instantiating the class*** because we are creating a particular ***instance*** of the data type described by the class. Some authors use the term ***instancing*** or ***creating an instance.***

After the objects are **new**-ed, we can call the references ***objects*** without confusion.

Finally we want to assign data to the ***objects***, **birkoff** and **walter**. In the case of a primitive data type, we would do this:

```
k = 4;
```

But in the **Employee** case, there is more than one elementary data type that comprises it. So we have to explicitly say which one of the three we are interested in. We do this with the ***dot operator***:

```
birkoff.age = 26;
walter.socSec = 777443333;
```

or

```
System.out.println( birkoff.age + " "
   + walter.socSec );
```

We will complete this example in the next section.

Section 5 - The Employee Class

6A.5.1 The Program Listing

Here's a program that defines the **Employee *class***, then creates some **Employee *objects*** in the **main()** method of the main class. I have declared and used some **int** data along the way so you could compare the operation of ***declaring*** and ***using class variables*** with that of ***declaring*** and ***using* int *variables***.

```
class Employee
{
   // member data for the class
   public long socSec;
   public double wage;
   public short age;
}

// main class Foothill --------------------
public class Foothill
{
   public static void main (String[] args)
   {
      Employee walter, birkoff;
      int a, b;

      // allocate and initialize walter
      walter = new Employee();
      walter.socSec = 123456789;
      walter.wage = 12.95;
      walter.age = 61;

      // initialize a;
      a = 89;

      // assign to birkoff
      birkoff = walter;

      // assign to b
      b = a;

      // show both a and  b:
      System.out.println( "a: " + a + "\nb: "+ b);

      // show both walter and  birkoff:
      System.out.println( "walter: ss#->" + walter.socSec
         + " wage->"+ walter.wage
         + " age->"+ walter.age );
      System.out.println( "birkoff: ss#->" + birkoff.socSec
         + " wage->"+ birkoff.wage
         + " age->"+ birkoff.age );
```

```
      // modify a and walter
      a++;
      walter.age = (short)(walter.age - 40);
      walter.socSec ++ ;

      // show both a and  b after changing a:
      System.out.println( "a: " + a + "\nb: "+ b);

      // show both walter and  birkoff after changing walter:
      System.out.println( "walter: ss#->" + walter.socSec
         + " wage->"+ walter.wage
         + " age->"+ walter.age );
      System.out.println( "birkoff: ss#->" + birkoff.socSec
         + " wage->"+ birkoff.wage
         + " age->"+ birkoff.age );
   }
}

/* ------ a sample run for this program --------
a: 89
b: 89
walter: ss#->123456789 wage->12.95 age->61
birkoff: ss#->123456789 wage->12.95 age->61
a: 90
b: 89
walter: ss#->123456790 wage->12.95 age->21
birkoff: ss#->123456790 wage->12.95 age->21
   ---------------------------------------------  */
```

6A.5.2 The Meaning of the Assignment Operation with Objects

As before with the **Pets**, we see that we can set one **Employee** object equal to another,

```
   birkoff = walter;
```

This is analogous to assigning one **int** to another, except that instead of the objects being **ints**, they are **Employees** -- our home grown data type. But the analogy isn't complete. When we later modify one of the objects, **walter**, and then print out **birkoff**, we see that **birkoff** has changed! This is different from what occurs when we assign one **int** to another. With primitive data types, the two variables' data have different destinies after the assignment. With user-defined objects, after the assignment, what happens to one affects the other

The reason for this is that object variables (**walter**, **birkoff**) are not themselves objects, but *pointers to objects*, or, as they are more commonly called, ***references***. After the assignment, we have simply pointed **birkoff** at the same object to which ***walter*** was pointing; the two ***references*** then refer to the same object. So while it appears that changing **walter** also changes **birkoff**, what really is happening is that there is only one object, and any changes to it are seen by both references, **birkoff** and **walter**. See?