Lecture 5

We discuss, approvingly, the idea of reusing existing code rather than building 'from scratch'.

But, since we often have to write our own class definitions, we begin to look at how to do this.

For the class of gibbering wrecks, we write a class definition which, to begin with, has three instance methods.

Reuse

- **Reuse** is using pre-existing artifacts to construct a new artifact
- In object-oriented software development, we might reuse:
  - methods and other chunks of code (code salvaging, the cut-and-paste approach)
  - class definitions
  - collections of class definitions that collaborate
  - designs (see *design patterns*)
  - other, e.g. documentation, tests, etc.

Advantages & Problems

- Reuse increases productivity (it can cut analysis, design & implementation time); this can cut costs, improve morale and give market advantage
- Reuse increases software quality (e.g. software may be more reliable due to 'cumulative debugging' and you might feel compelled to write it in a more more maintainable/extensible way)
- But, generally, you get high levels of reuse only if it is planned for
  - e.g. make the artifact more general than is needed for present purposes
  - e.g. place the artifact in a library

Reusing class definitions

- Object-oriented programming encourages reuse: class definitions especially can be the building blocks of reuse
- Our programs so far exemplify this:
  - We've reused existing classes from the standard class library
  - With little extra code, we get interesting programs
- But, while there are about 1500 classes in the standard Java 2 library and there are other libraries available for free or for a fee, we will often have to define new classes
- (If we make a good job of them, they might go into a library for someone else to reuse)

Writing our own class definitions: Gibbering Wrecks

- Example client code:

```
Wreck w1 = new Wreck();
w1.displayGreeting();
w1.displayGibberish();
System.out.println(
   w1.getRepetition());
Wreck w2 = new Wreck();
System.out.println(
   w2.getMimicry("nahnah"));
```

- Begin by deciding on the interface:

```
public void displayGreeting()
public String getGreeting()
public void displayGibberish()
public String getMimicry(
  String aMessage)
public String getRepetition()
```


The `displayGreeting` instance method

```
/**
 * A class that represents highly emotional
 * objects.
 * @author Derek Bridge 666 d.bridge@cs.ucc.
 */
public class Wreck
{
   /**
    * Displays a greeting on the standard
    * output.
    */
   public void displayGreeting()
   {


   }
}
```

A client

```
/**
 * A program that demonstrates the behaviour
 * of two Wreck objects.
 * @author Derek Bridge 666 d.bridge@cs.ucc.
 */
public class Wreckers
{
   public static void main(String[] args)
   {  Wreck w1 = new Wreck();
      w1.displayGreeting();
      Wreck w2 = new Wreck();
      w2.displayGreeting();
   }
}
```

The lecture will make clear exactly how memory gets populated with two `Wreck` objects at run-time and why the method is called an instance method.

The `getGreeting` instance method

Add to the class definition:

```
/**
 * Returns a greeting from this wreck.
 *
 * @return a string containing a greeting.
 */
public String getGreeting()
{


}
```

Add to the client:

```
w1.getGreeting();
String s = w2.getGreeting();
System.out.println(s.toUpperCase());
System.out.println(w1.getGreeting());
```

What will happen when we run the client code?

The `displayGibberish` instance method

```
/**
 * Displays two random syllables on the
 * standard output. Each syllable comprises
 * a random consonant-vowel pair.
 */
public void displayGibberish()
{  /* Set up vowel & consonant lists.
    */



   /* Generate a random number and use it
      to select a consonant.
    */
```


(`displayGibberish`, cont'd)

```
   /* Generate a random number and use it
      to select a vowel.
    */


   /* Form the syllable from the consonant
      and the vowel and display it twice.
    */



}
```

Add some code to the client to try this out.

Implementation notes for `displayGibberish`

- To generate a random number (a `double`) between 0.0 (inclusive) and 1.0 (exclusive):

  `Math.random();`

  (This uses a *class method*, see later.)

- To generate a random `int` between, e.g., 0 (inclusive) and 10 (exclusive):

  `(int) (Math.random() * 10);`

  (Scale it, then 'cast' it to an `int`)

- How would you simulate a dice roll (i.e. generate a number between 1 and 6 inclusive)?

Implementation notes for `displayGibberish`, cont'd

- We will use `length()` to compute string lengths instead of hard-coding numbers. Why?

- Three types of comments in Java:
  - // Use for short comments *inside* methods
  - /*...*/ Use for longer comments *inside* methods
  - /**...*/ Use at the *start* of the class definition and *before* each method (not inside methods)

- We need to store temporary results during the execution of the method body: **local variables**