CS360 Lecture 5
Object-Oriented Concepts

Tuesday, February 17, 2004

Last Time

On Thursday I introduced the basics of object-oriented programming. We covered:

- Creating public classes (each public class must be in its own file
- Each class must have a constructor. The constructor is what is called whenever an instance of an object is created. If no constructor is explicitly supplied, then the compiler creates a default constructor that takes no arguments.
- A class can have many constructors (called overloaded constructors)
- Accessor and mutator methods. These are the get and set methods. These are required to manipulate and query private instance variables.

Creating instances of classes (or objects) and using the objects to access the class methods.

Today we will look at other object-oriented programming concepts, and memory management in Java.

Constructing Objects

How are objects constructed in C++?

```
Time3 t;
Time3 t( 3, 34, 0 );
Time3* pTime = new Time3( 4, 23, 34 );
```

What is the difference between these objects?

How are objects constructed in Java?
Time3 t = new Time3(3, 54, 32);

What is t? Is it a reference to Time3 or is it Time3?

In the Java case, t is a disguised pointer. It is disguised because you cannot dereference it. What is dereferencing?

How is the object reference in Java different from a pointer in C++?

Passing Class Objects as Method Arguments

Compare between passing class objects by value in C++ and Java. In C++ the object is passed as a copy, in Java a reference to the object is passed. This means that, in Java, any change to the object in the method will be reflected outside of the method.

Composition

A class can have instance variables that are objects of other classes. This capability is called composition.

Assume that we have two classes:

- Time3 is the same class that we used last week. It contains three instance variables for the hour, minute and second. It also contains methods for converting the time to a string.
- Class Meeting is defined below.

```
public class Meeting {
   private Time3 meetingTime;
   private String meetingPurpose;

   public Meeting( Time3 time, String purpose )
   {
      meetingTime = time;
      meetingPurpose = purpose;
   }

   public String toMeetingString()
   {
      return meetingPurpose + " at " +
               meetingTime.toStandardString();
   }
}
```

Let us now test both classes. We need to create a new class for the Java application (it will contain the `main` method). In this class we want to create an instance (object) of the Meeting class to hold the data for the PUCC meeting at noon.

How would we go about doing this?

```
public class MeetingTest {

   public static void main( String args[] )
   {
      Time3 time = new Time3( 12, 05, 00 );
      Meeting pucc = new Meeting( time,
                              "PUCC Meeting" );

       time.setTime( 16, 00, 00 );
      Meeting boxerRom = new Meeting( time,
                                "BoxerRom Meeting" );

      String output = "The meetings you have for
                          today are: \n" +
```

```
                        pucc.toMeetingString() +
                           "\n" +
                           boxerRom.toMeetingString();

      JTextArea out = new JTextArea();
      out.setText( output );

      JOptionPane.showMessageDialog( null, out,
                    "Testing Class Time1",
                    JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   } // end main
}
```

Question: What is wrong with the above code? What is the output?

Passing Objects to Methods

I have already mentioned that in Java all arguments are passed value. There are three ways of passing arguments in C++, what are they?

Although the arguments are passed by value in Java, in actual fact they are passed by reference.

What does that mean?

```
public class MeetingTest2 {

   public static void main( String args[] )
   {
      Time3 time = new Time3( 12, 05, 00 );
      Meeting pucc = new Meeting( time,
                                      "PUCC Meeting" );

      String output = "The meetings you have for
                          today are: \n" +
                     pucc.toMeetingString();

      change( pucc );

      output += "\n\n\n After changing the name and
                    time of the meeting: \n" +
                pucc.toMeetingString();

      JTextArea out = new JTextArea();
      out.setText( output );
```

```
      JOptionPane.showMessageDialog( null, out,
               "Testing Class Time1",
               JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   }

   public static void change( Meeting m1 )
   {
      m1.setMeetingTime( 15, 30, 00 );
      m1.setMeetingPurpose( "Campus Life" );
   }
}
```

What is the output of the above program?

Let's look at another program.

```
public class MeetingTest {

   public static void main( String args[] )
   {
      Time3 time = new Time3( 12, 05, 00 );
      Meeting pucc = new Meeting( time,
                              "PUCC Meeting" );
      Time3 time2 = new Time3( 16, 00, 00 );
      Meeting boxerRom = new Meeting( time2,
                              "BoxerRom Meeting" );

      String output = "The meetings you have for
                           today are: \n" +
                     pucc.toMeetingString() +
                           "\n" +
                           boxerRom.toMeetingString();

      swap( pucc, boxerRom );

      output += "\n\n\n After swapping, the
                meetings are: \n" +
                pucc.toMeetingString() + "\n" +
                 boxerRom.toMeetingString();

      JTextArea out = new JTextArea();
      out.setText( output );
```

```
      JOptionPane.showMessageDialog( null, out,
               "Testing Class Time1",
               JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   }
   public static void swap(Meeting m1, Meeting m2)
   {
      Meeting temp = m1;
      m1 = m2;
      m2 = temp;
   }
}
```

Static Class Variables and Methods

Static class variables are variables that are shared by all objects of a class. Only one copy of the variable is ever created (during the compilation of the class) and is shared by all the objects.

Why would we need to do this?

Public static methods can be invoked by an object name or by the class name.

```
class Robot
{
   public int idNum;
   public static int nextIdNum = 1;
   public String owner;

   public int getIdNum() { return idNum; }
   public String getOwnder() { return owner; }

   public Robot() { idNum = nextIdNum++; }
   public Robot( String  name ) { this();
                                  owner = name; }

   public void print() { System.out.println(
                          idNum + " " + owner ); }
}

class RobotTest
{
   public static void main( String args[] )
   {
      System.out.println( Robot.nextIdNum );
      Robot r1 = new Robot( "Shereen" );
```

```
        r1.print();
        Robot r2 = new Robot( "Doug" );
        r2.print();
        Robot r3 = new Robot( "Michelle" );
        r3.print();
    }
}
```

Memory Management in Java

All objects in Java are created using the `new` operator. The operator constructs an object of a given class and returns a reference to it. Memory allocated for new objects comes from the heap.

Java does not need a delete operator to free up memory occupied by objects that are no longer referenced. The garbage collector in Java automatically does this. You can indicate that an object is no longer referenced, by setting it to null.

```
Time3 time =  new Time3( 23, 45, 00 );
time = null;
```

Garbage Collection

Whenever an object is created, its constructor is called. Even if you do not explicitly create a constructor in a class, then the default constructor in the object class (which every class inherits from) is called. The constructor acquires memory in order to store the instance variables.

For example, in the Meeting class we saw earlier, the constructor needs to allocate space for the object of class Time and the String variable. What happens to this space once the object has been destroyed (it is not referenced any more)?

Destructors

How are destructors created in C++?

In Java, the destructor is the `finalize` method. This method is declared in the Object class and it is possible for any class to overload this method, though you should never need to do this.

Java performs automatic garbage collection to return the memory occupied by objects that are no longer referenced back to the system.

The finalize method is always called finalize and has return type void. Java does not provide a way to destroy an object at will. Instead, you can recommend to Java to run the garbage collector, but there is no guarantee that the garbage collector will run at that time (or at all).

```
class X
{
   int id;
   static int nextId = 1;

   public X()
   {
      id = nextId ++;
      if( id % 1000 == 0 )
         System.out.println( "Construction of X
                              object, id = " + id );
   }

   protected void finalize() throws Throwable
   {
      if( id % 1000 == 0 )
         System.out.println( "Finalization of X
                              object, id = " + id );
      super.finalize();
   }
}

class Test
{
   public static void main( String args[] )
   {
      X[] xArray = new X[ 10000 ];
      for( int i=0; i< 10000; i++ )
         xArray[i] = new X();
      xArray = null;
      System.gc();
   }
}
```