CSE 11: Lecture 16

- Generic sorting and the java.util.Comparator interface
- Applets and HTML
- Javadoc

(Reading: Savitch, Ch. 13 for Applets, Appendix 10 for Javadoc)

More on sorting and searching

- Last time, we looked at some sorting algorithms, for example bubble sort:

```
void sort(double A[])  {
   int len = A.length;
   for (int N=0; N<len; N++)
      for (int M=len-1; M>N; M--)
         if (A[M] < A[M-1]) swap(A,M,M-1);
}
```

- ... and at a search algorithm, namely linear search:

```
   boolean find(Object arr[], Object key) {
      for(int i=0; i<arr.length; i++)
         if (arr[i].equals(key)) return true;
      return false;
   }
```

Toward generic sorting

- Note one important difference between those two algorithms' implementations...
- The search method is *generic*: it will work with any Objects
- The sort method is not generic: it will work only with one type, double
- The sort method could be made generic like the search method, if every Object had something like a "lessThan" method...
  - but there is no such method defined in the Object class...
  - and it would not really make sense to have one, because "saying if I'm less than another" is not a behavior *every* object should have
- So, sorting in Java cannot be made completely generic... but we can get close

Comparing objects in Java

- Some algorithms (for example, linear search) require being able to compare items to each other for equality
- In Java, this is straightforward to do: every object has an `equals()` method
- Some algorithms (for example, sorting) require being able to compare items to each other for *ordering*, such that for any two items A,B exactly one of these must be true:
  - A is less than B, or
  - A is greater than B, or
  - A is equal to B
- In JDK 1.2 ("Java 2"), there are two ways to meet this requirement:
  - Make sure items to be sorted are instances of a class that implements the interface `java.lang.Comparable`.
  - Make sure items to be sorted are instances of a class that can be compared using the `compare()` instance method of an object that implements the interface `java.util.Comparator`, and let the sort algorithm use that Comparator

interface java.lang.Comparable

- The Comparable interface imposes a total ordering on the objects of any class that implements it
- This ordering is referred to as the class's "natural ordering", and the class's **compareTo** method is referred to as its "natural comparison method"
- An instance of such a class "knows how" to compare itself to other objects for ordering

```
public interface Comparable {
/**
* Compares this object with the specified object for order. Returns
* a negative integer, zero, or a positive integer as this object is
* less than, equal to, or greater than the specified object.
*
* @param   o the Object to be compared.
* @return a negative integer, zero, or a positive integer as this
* object is less than, equal to, or greater than the param object.
* @throws ClassCastException if the param object's type
* prevents it from being compared to this Object.
*/
    public int compareTo(Object o);
}
```

The Comparable interface, cont'd

- The implementor of the compareTo method should follow these guidelines:
  - For all x and y, the sign of y.compareTo(x) must be -1 times the sign of x.compareTo(y)
  - For all x and y, y.compareTo(x) must throw an exception if x.compareTo(y) throws
  - For all x, y, and z, if x.compareTo(y) > 0 and y.compareTo(z) > 0, then x.compareTo(z)>0
  - For all x, y, and z, if x.compareTo(y) == 0, then the sign of x.compare(z) must be the same as the sign of y.compareTo(z)
  - For all x and y, (x.compareTo(y) == 0) == (x.equals(y)).
    - If this holds, the natural ordering is "consistent with equals"
    - If this does not hold, the fact should be clearly indicated in documentation!

- In Java 2, the following standard Java classes implement the **`Comparable`** interface, providing a compareTo method that is consistent with equals:

`Byte, Character, Double, File, Float, Long, ObjectStreamField, Short, String, Integer, BigInteger, BigDecimal, Date, CollationKey`

interface java.util.Comparator

- The Comparator interface specifies a comparison function, which imposes a total ordering on some kind of objects.
- An instance of a class that implements Comparator “knows how” to compare pairs of objects for ordering
- A Comparator can be passed to a sort method to allow “non-natural” control over the sort order, and to control the order of certain data structures
- (The same guidelines apply as for the compareTo method of Comparable)

```
public interface Comparator {
/**
* Compares its two arguments for order. Returns
* a negative integer, zero, or a positive integer as its first
* argument is less than, equal to, or greater than the second.
* @return a negative integer, zero, or a positive integer as the
* first argument is less than, equal to, or greater than the
* second argument.
* @throws ClassCastException if the arguments' types prevent them
* from being compared by this Comparator.
*/
     int compare(Object o1, Object o2);
}
```

Generic sorting

- Bubble sort can now be defined like this, to work on Comparable objects:

```
void sort(Comparable a[])  {
   int len = a.length;
   for (int N=0; N<len; N++)
      for (int M=len-1; M>N; M--)
         if (a[M].compareTo(a[M-1]) < 0) swap(a,M,M-1);
}
```

- ... or like this, to use a Comparator:

```
void sort(Object a[], java.util.Comparator c)  {
   int len = a.length;
   for (int N=0; N<len; N++)
      for (int M=len-1; M>N; M--)
         if (c.compare(a[M],a[M-1]) < 0) swap(a,M,M-1);
}
```

- These implementations are now quite generic

An example implementation of Comparator

- Suppose Time is a class that represents a time in terms of hours, minutes, and seconds after midnight; and we wanted to sort Time objects according to their time after midnight

- We could define a class like this:

```
public class TimeComparator implements java.util.Comparator {

   public int compare(Object o1, Object o2) {
      Time t1 = (Time) o1;
      Time t2 = (Time) o2;
      int h1 = t1.getHours(), h2 = t2.getHours();
      int m1 = t1.getMinutes(), m2 = t2.getMinutes();
      int s1 = t1.getSeconds(), s2 = t2.getSeconds();
      if(h1 != h2) return h1 - h2;
      if(m1 != m2) return m1 - m2;
      return s1 - s2;
   }
}
```

- An instance of this class could be passed as second arg to the sort() method, to sort an array of Time[] objects

HTML

- HTML stands for “Hyper-Text Markup Language”
- HTML is a simple set of commands (sometimes called tags) that when included in a document instruct a web browser how to display the document as a web page
- Some simple HTML tags:

  Start a new paragraph: **<P>**

  Start a new line: **
**

  Start bold-faced text: **<B>**
  End bold-faced text: **</B>**

  Start italic text: **<EM>**
  End italic text: **</EM>**

  Start a comment: **<!--**
  End a comment: **-->**

  Start a hypertext anchor: **<A HREF= ... >**
  End a hypertext anchor: **</A>**

Applets

- An *applet* is a GUI Java program that runs as a window embedded in a web page viewed in a browser

- You get an applet to appear in the browser window viewing the web page by
  - writing the applet source code (we'll see how to do that in a moment)
  - compiling the applet to produce a .class file
  - referring to the .class file in the web page's HTML document using a applet tag pair:

```
<APPLET CODE="name-of-class-file" WIDTH=integer HEIGHT=integer>
</APPLET>
```

- When the web page is loaded into the browser, the .class files for the applet are loaded too, and the applet runs on the same machine as the browser

- The applet displays itself in a region of the browser window with the given width and height

The <APPLET> tag

- Q: What goes between the opening <APPLET ...> and closing </APPLET> tags?

  A: Two things...

- 1. HTML-formatted text you place between the tags will appear in the browser window, if the browser is not Java-enabled.

- 2. PARAM tags can be placed here to pass arguments to the applet. A PARAM tag has the form

  <PARAM NAME=*param-name* VALUE=*param-value*>

  - There can be any number of PARAM tag
  - The applet can get the value of a parameter as a String by calling its own getParameter() instance method
  - This is similar to a Java application getting command-line Strings through main's String[] array

Writing an applet

- A Java applet is very similar to a Java Swing GUI application
- Some important differences:
  - In Java 1.2 or later, instead of extending JFrame, your applet class extends JApplet
  - In Java 1.1 or earlier, instead of extending Frame, your applet class extends Applet
    - Applet or JApplet are very similar to Frame or JFrame
  - Instead of writing a main() method or a constructor for your applet class, you override the `public void init()` method to create components and add them to the applet
    - ... the init() method is called when the browser finishes loading your applet .class files; this starts the execution of the applet
  - Because of the "applet security model", an applet cannot access files on the browser's machine, and is restricted in some other ways as well
- Otherwise, writing an applet is pretty much like writing an application

A hello-world applet

- This is a simple “Hello World!” applet

```
import java.awt.*;
import java.applet.*;

public class Hello extends Applet{
   public void init() {

      Label lab = new Label();  // create a Label to hold text
      lab.setText("Hello World!"); // set its text

      add(lab);      // add the Label to this applet

   }
}
```

A hello-world applet, cont'd

- To view the applet:
  - compile it to create a .class file
  - create a HTML document with an applet tag that references the .class file:

```
<B>
Applet will appear below this line!
</B>
<applet code="Hello" height=480 width=640>
</applet>
```

  - view the HTML document in a browser with Java enabled, or use **`appletviewer`**

A parameter-passing applet

- This is an applet that displays the value of a parameter

```
import java.awt.*;
import java.applet.*;

public class Win extends Applet{
   public void init() {

      Label lab = new Label();  // create a Label to hold text

      String msg = getParameter("message");

      lab.setText(msg); // set its text

      add(lab);      // add the Label to this applet

   }
}
```

A parameter-passing applet, cont'd

- To view the applet:
  - compile it to create a .class file
  - create a HTML document with an applet tag that references the .class file and that uses a PARAM tag to pass an argument:

```
<B>
Applet will appear below this line!
</B>
<applet code="Win" height=480 width=640>
<param name=message value="How about that!">
Alternate text can go here.
</applet>
```

  - view the document in a browser with Java enabled, or use **`appletviewer`**

Applets: pro and con

- ✔ One of the original ideas behind the design of Java was that computers and computer-based devices should be able to download software from the network, as needed
- ✔ This network could be wireless, so that as a device moves around the world, it is continually downloading software and running it
- ✔ This idea requires a flexible, portable, secure language that can be compiled into a compact form... and Java was designed Java to be pretty good on these points
- ✔ Applets are an early and only partially successful expression of this idea:
  - ✗ Not all browsers support Java applets (and users can disable Java in their browser if they want, in any case)
  - ✗ Different browsers support different event models (if you use the "JDK 1.1" event model as we have discussed in class, Netscape 4.0 or earlier will not work with it; Internet Explorer does not support JFC/Swing without a plugin)
  - ✗ Because of security concerns, applets are by default limited in what they can do (it is hard to access files on the user's machine, or to open general network connections)
- ✔ A more complete and more useful implementation of the idea will appear with JINI and other Java technologies, and other web services frameworks

Interface, implementation, and “doc comments”

- It is a principle of good ADT design that these should be separate:
  - the *interface* (what the user of the ADT sees and gets to interact with)
  - the *implementation* (gory details that the programmer of the ADT had to deal with, but a user does not need to know about to use the ADT)

- Some programming languages adopt the convention that interface code (e.g. method headers) and implementation code (full ethod definitions) should be kept in separate files

- Java does not permit that convention: everything about a class goes into one .java file

- Instead, the Java Development Kit (JDK), besides the Java compiler **`javac`** and the Java interpreter **`java`**, has a program **`javadoc`** which can automatically produce a separate HTML interface specification document from your .java source code files.

- This is a very nice feature! To make use of it, you have to write “doc comments” in your source code

“doc comments”

- Comments in Java are of 3 kinds:
  - double-slash comments `// like this one`
  - “C-style” comments `/* like this one */`
  - “ doc comments” `/** like C-style, except start with /**  */`
- When doc comments immediately precede class, data member, or method definitions, and you run **`javadoc`** on the source code files, hyperlinked HTML files are created for you that document these definitions

doc comments, cont'd

- doc comments are the appropriate place to put documentation about the user interface to a class: about the class as a whole, and about its public methods and variables
- A doc comment must immediately precede the declaration of the class, method, or variable that it documents
- The first sentence of the doc comment (up to the first period '.') should be a brief summary of what the class, method, or variable does. Javadoc will use this as a summary comment
- Any spaces and asterisks '*' are eliminated from the beginning of each line
- **`javadoc`** will create HTML documents, so you can include simple HTML tags in your doc comments to improve formatting if you want
- There are also special doc tags, such as **`@author`**, **`@version`**, **`@param`**, **`@returns`**, etc.

doc comments: an example

```
/** Time implements a 24-hour time in hours, minutes, and seconds.
 * Solution for Programming Assignment CSE 11
 * @author Paul Kube kube@cs.ucsd.edu copyright (c) 1999-2000
 * @version 1.1
 */
public class Time {
   /** Initialize a Time object to 12:00:00. */
  public Time() {
    this(12,0,0);
  }
   /** Initialize a Time object to given hours, minutes, and
    *  seconds.
    * @param h The initial hours value
    * @param m The initial minutes value
    * @param s The initial seconds value
    */
  public Time(int h, int m, int s) {
    this.h=h; this.m=m; this.s=s;
  }
```

Using javadoc

- Running `javadoc *.java` in a directory will create a collection of hyperlinked .html files which include information from doc comments and other facts about your classes that javadoc can determine

- One file created is `index.html`. This is an index to the other files javadoc creates

- For an example of javadoc generated documentation of all the Java standard libraries, follow links from the Textbooks and other Resources class page to the Java API documentation

Next time

- Packages
- Package naming, and CLASSPATH
- Multithreaded programming
- Synchronized statements and synchronized methods