

LCC 6310 The Computer as an Expressive Medium

Lecture 18

Overview

Project 4 questions?

Assignment 5

Java for real!

- Java application basics

- Java applet basics

- Graphical Java - double buffering!

- Mouse listeners

Project 4

Due: Friday November 2

Making sense of the world is not just a matter of structure, but of process, of the dynamic construction of meaning. And as we've been discovering together, computation is fundamentally a process medium. What would you do to the web? Create an applet that dynamically does something to one or more web pages (e.g. collage, systematic distortion, re-layout, ironic superposition, etc.).

Assignment 5

Posted online, **not graded**

- A5-01: Modify image collage to, instead of grabbing images, grab headlines from several news sources and display them. This gives you practice in looking at the html source for multiple sites (e.g. New York Times, CNN), determining how a piece of information is represented, and writing the parse code to grab that piece of information.
- A5-02: Write an html parser that looks for keywords (you pick the keywords) in the text (not within a tag) of a page and counts how many times different keywords appear. You can imagine that this might be the beginning of an information visualizer that visualizes pages as a function of different keywords that appear.

JDK

Download the JDK (J2SE Development Kit) from java.sun.com

The JDK provides command line tools to compile and run java applications and applets, located in the `<jdk-install>/bin` directory:

`javac.exe` - compile java application or applet

`jar.exe` - archive files into a .jar file

`java.exe` - run java application

(We'll see the usage of these later...)

Choose your favorite **text editor** to write Java code (e.g. I like Notepad2 or emacs so that's what you'll see me using, but you can choose any simple text editor you like - e.g. notepad, vi, bbedit, etc.)

We'll build a simple java application and then a java applet...

Java application basics

The main class

Your application needs to have a main class with a main method

This is where the program execution thread will start

```
public static void main(String[] args) {  
    ...  
}
```

This method needs to be **public** since it is called from outside and **static** because it is called before any object is instantiated

The array of strings are the **command line arguments** (if any) for the application

Let's try to build a simple application...

Hello world!

```
public class HelloConsole {  
  
    // Constructor  
    // -----  
    public HelloConsole() {  
        System.out.println("Hello world!");  
    }  
  
    // Main Entry Point  
    // -----  
    public static void main(String[] args) {  
        new HelloConsole();  
    }  
}
```

Let's look at the [HelloConsole.java](#) file...

Now we just need to compile and run this!

Compile

The first step is to compile our java class...

We do this by opening a command prompt (e.g. DOS shell on Windows), navigating to the directory where the source code is stored, and typing the following command:

```
> javac *.java
```

This means "compile all the java classes in this directory" (the `**` is a wildcard) and generates compiled `.class` files

If there are any errors, they will show up in the console window

If the `<jdk-install>/bin` directory is not in your PATH environment variable, you might need to type the entire path for your java command, e.g. `c:\java\jdk1.6.0_03\bin\javac.exe`

Let's try compiling our `HelloConsole.java` file at the command line...

Jar manifest

Now that we've compiled, we want to run our application. To do this, we can package our `.class` files into a `.jar` file...

This isn't strictly necessary, but it makes things a lot easier when you start to have lots of `.class` files!

But first we need to create a **Manifest**...

The jar manifest is used to define extension and package-related data for a java application. You can read the details here:

<http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html>

We need to have a **Main-Class** attribute in our manifest. This will define the relative path of the main application class which the launcher will load at startup. Without one, we wouldn't be able to run the jar file from the command-line. **Let's create a manifest file...**

HelloConsole Manifest file

Create a new file in a text editor and type the following line:

```
Main-Class: HelloConsole
```

Save this file (you can use whatever name you like, e.g. `Manifest.txt`)

Next we'll see how to use the **jar command** to add it to our jar file when we package our java classes...

Jar command

Now we're ready to package our `.class` files into a `.jar` file!

We do this by invoking the jar command at the command line:

```
> jar -cfm HelloConsole.jar Manifest.txt *.class
```

Extensions invoked with our call to the jar command:

c - create a new archive

f - specify the name of the archive (e.g. `HelloConsole.jar`)

m - include manifest info from the specified file (e.g. `Manifest.txt`)

(Note: the order of arguments in your call should correspond to the order in which you typed in the extensions, e.g. `cfm` vs. `cmf`)

You can look at the usage of the jar command as follows:

```
> jar
```

This will output all the possible extensions you can use...

Run

Now that we have our HelloConsole.jar file, we can run our Java application using the following command:

```
> java -jar HelloConsole.jar
```

In this case, we see the following output:

```
Hello world!
```

The `java` command launches our Java application by starting a Java Runtime Environment (JRE), loading the specified main class and invoking its main method. The `-jar` extension launches the application from the packaged `.jar` file.

You can also run the application by double-clicking the `.jar` file.

Summary of steps

How to compile, package and run a Java application:

Step 0: Write source code for your java classes in text editor and create a manifest file which describes the main class

Step 1: Compile your `.java` files

```
> javac *.java
```

Step 2: Package your `.class` files into a `.jar`

```
> jar -cfm MyJar.jar MyManifest.txt *.class
```

Step 3: Run your java application using the `.jar`

```
> java -jar MyJar.jar
```

Java applet basics

Creating applets

Unlike applications, applets do not have a main method. Instead, to create an applet, we start by extending the Java Applet class

Look at the API docs for Applet:

<http://java.sun.com/j2se/1.4.2/docs/api/>

Some methods we can override to give the applet desired functionality:

`init()` - called by the browser to inform that the applet is loaded

`start()` - called by the browser to tell the applet to start its execution

`stop()` - called by the browser to tell the applet to stop its execution

`destroy()` - called by the browser to tell the applet that it is being reclaimed and should destroy resources that it has allocated

Let's try to build a simple application...

Hello world!

```
import java.awt.*; // needed in order to use the Color class
import java.applet.*; // needed in order to extend Applet

public class HelloConsoleApplet extends Applet {

    // Applet Init
    // -----
    public void init() {
        setBackground(Color.black);
        System.out.println("Hello World!"); // for an applet, where do you
                                           // think this will be displayed?
    }
}
```

Let's open the [HelloConsoleApplet.java](#) file...

Let's compile and package this

Compile and jar

Compiling an applet is just like compiling an application:

```
> javac *.java
```

And we again use the jar command to package our .class files:

```
> jar -cf HelloConsoleApplet.jar *.class
```

Note that we do not need to provide a manifest file, since applets do not have a main method! Instead, let's look at how to run our applet on a webpage...

HTML applet tag

Let's put our applet on an web page.

To do this, create a .html file in a text editor that contains the following: (e.g. HelloConsoleApplet.html)

```
<html>
<head>
<title>Hello Applet</title>
</head>
<body bgcolor="333333" text="ffffff">
<applet archive="HelloConsoleApplet.jar" code="HelloConsoleApplet"
width=300 height=100>
</applet>
</body>
</html>
```

Applet tag attributes

The **applet** tag with parameters tells the browser to put an applet on the page based on the specified attributes:

```
<applet archive="HelloConsoleApplet.jar" code="HelloConsoleApplet"
width=300 height=100>
```

archive - .jar where to find the code (required if you're using a jar)

code - name of the Applet subclass that should be launched (required)

(both these should be relative to the base URL of the applet)

width and **height** - required attributes to specify initial size of the applet

You can read about these and other attributes here:

<http://java.sun.com/j2se/1.4.2/docs/guide/misc/applet.html>

Let's see our applet run... all we need to do is open the [HelloConsoleApplet.html](#) file!

Graphical Java

The AWT

The **AWT** is the Java Abstract Windowing Toolkit - a package that provides all sorts of classes for creating graphical applications

Windows, panels, canvases, menus, fonts, images etc.

Swing is another windowing toolkit that has even more functionality. For simplicity, we'll use the AWT.

You can look into Swing on your own if you like.

The AWT and Swing classes are in the following packages:

`java.awt.*`

`javax.swing.*`

Graphical hello applet

Printing text in the console in our HelloConsoleApplet example was really not the right use for an applet - applets are meant to be graphical!

So let's see how we can print out the text in the applet panel that shows up in the browser...

The paint method

If we look at the Applet class hierarchy, we notice that Applets are actually AWT Panels (which are Containers, which are Components, which are Objects)

A **component** is a graphical object that can be displayed on-screen and that can interact with the user

Component is an abstract superclass that can be directly extended to create lightweight components (components that are not associated with a native opaque window)

Component contains a **paint** method that is called by the System whenever the component needs to be painted (e.g. if the component has been partially obscured and revealed again on screen). You can also call **repaint** to trigger a call to the paint method from your own code.

You can override the paint method in a Component subclass (or Panel, Applet etc. subclass) to say how your own class should be painted

Let's try this out in our Hello World applet...

```

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

public class HelloGraphicsApplet extends Applet {
    int w,h; // applet size
    String hellostr = "Hello world!"; // hello string
    int sw, sh, sx, sy; // hello string size and position

    public void init() {
        setBackground(Color.black);
        w = Integer.parseInt(getParameter("width"));
        h = Integer.parseInt(getParameter("height"));
    }

    public void paint(Graphics g) {
        FontMetrics fm = g.getFontMetrics(); // get our string size info
        sw = fm.stringWidth(hellostr);
        sh = fm.getHeight();
        g.setColor(Color.red);
        g.drawString(hellostr, (w-sw)/2, h/2);
    }
}

```

Let's look at the [HelloGraphicsApplet.java](#) file...

Let's turn this into an application...

To do this we need to do the following:

Instead of extending Applet, we can extend Panel

Decide how big our Panel should be

We can override the `getMinimumSize` and `getPreferredSize` methods to make sure it never gets displayed at any other size...

Override the paint method so that we can paint our text

Put our Panel in a Frame (top-level window to be displayed on screen)

Before trying this, let's look at one new concept - **double buffering!**

Double buffering

We can store off-screen image and graphics objects that mirror the on-screen graphics object that is associated with our Panel

Then we can do all our painting into the off-screen graphics object first, and when we're done simply copy this to the on-screen graphics object that corresponds with our Panel

This will reduce the flickering that is caused when the onscreen graphics object is redrawn frequently while the application is simultaneously trying to paint new stuff into it

Double-buffering is not really needed in our Hello World example because what we're painting is simple and doesn't change, but becomes important when the drawing changes, e.g. with user interaction

Let's look at the [HelloGraphics.java](#) class...

Mouse Listeners

Moving the text

Now that we've drawn some text, let's see if we can use the mouse to move it around on-screen...

We do this by making our class implement the `MouseListener` and `MouseMotionListener` interfaces, which means we need to provide the following methods (and fill in whichever ones we want to use):

```
public void mouseEntered(MouseEvent e) {}
public void mouseExited (MouseEvent e) {}
public void mouseMoved  (MouseEvent e) {}
public void mouseClicked(MouseEvent e) {}
public void mousePressed(MouseEvent e) {}
public void mouseDragged(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}
```

Add ourselves as listeners...

We also need to add ourselves as listeners for `MouseEvents`.

This can be done in the `init()` method of our Applet subclass or in the constructor of our Panel subclass:

```
addMouseListener(this);
addMouseMotionListener(this);
```

Let's look at the implementations of this...

[MoveHelloGraphics.java](#)

[MoveHelloGraphicsApplet.java](#)

Remember...

This **Thursday** is GUVU demo day!

Class cancelled – go listen to Andy van Dam & Genevieve Bell!

Readings moved to the following week

So for next **Thursday**...

Four students: present one reading each

Everyone else: prepare one discussion question for each reading

Process Intensity - Chris Crawford & *Interactivity, Process Intensity, and Instantial Assets* - Greg Costikyan (linked from class page, for Costikyan scroll down to Tues, May 20, 2003)

Semiotic Considerations - Michael Mateas (linked from class page)

Computing Machinery and Human Intelligence - Alan Turing (NMR pp.49-64)

From *Computing Power and Human Reason* - Joseph Weizenbaum (NMR pp.367-375)