The Plan For Unit 13

- **Reading**: Ch 4 (125-135), Ch 11 (479-492), Ch 12
- **Console Applications**: writing text-mode Java programs
- **GUI Applications**: graphical stand-alone Java programs
- **Images**: how to load and paint images in Java
- **Threads & Animation**: making your images move
- **AudioClip**: adding simple sound to your programs
- **Lab**: SketchPad refines your event-handling skills
- **Homework**: SuperSketch adds images to your lab

Java Applications

- Along with applets, Java has two types of applications
  - Applications run as "stand-alone" programs
  - They run on your local machine
  - They don't use your Web browser
  - Instead, they use the Java interpreter or runtime
- **GUI applications**
  - Act like regular Windows/Mac programs
- **Console applications**
  - Use text input and output like a Pascal or C program
- Let's start by looking at console programs
I/O Basics

- Getting data into and out-of your program is called I/O
  - This particular view is called stream I/O

- Data can come into your program from various places
  - You can read data from a file, the keyboard, or the network
  - Each of these is called a source
  - The process of reading the data is called input

- Data is sent out of your program to different destinations:
  - You can write to a printer, the screen, or the network
  - Each of these destinations is called a sink
  - The process of writing the data is called output

The Glass Teletype

- The console is both an input and output device
  - Name given to the keyboard / monitor you are sitting at
- One early console was the teletype
  - Keyboard type input device
  - Fan-folded greenbar paper with line-printer for output
  - Output was an endless scroll of paper
- The VDT or CRT replaced output with scrolling monitor
  - Acts like a line-printer, except data scrolls off top of screen
  - In W2K/XP, you can scroll back and see previous pages

Java's Glass Teletype

- Java has three objects used for console I/O
  - System.out : the standard console output object
  - System.err : the standard console error output object
  - System.in : the standard console input object

- Each of these objects works with the O/S
  - Used to read and write data using standard streams
  - You can read more about these in Chapter 10
- For right now, let's look at the System.out object
Using System.out

- This predefined object has two interesting methods:
  ```java
  System.out.println("Hi there");
  System.out.print(42);
  System.out.println(" is the answer to everything");
  ```

- The print() and println() methods:
  - Can take arguments of any type (overloaded versions)
  - Arguments are converted to a String and printed
  - The println() method adds a newline to end of output
  - Output does not appear in applet window, but in console

Introducing ConsoleApp

- Applications are organized differently than applets
- Here's an example console-mode application

```java
public class ConsoleApp {
    public static void main(String[] args) {
        System.out.print("I am ");
        System.out.print(53);
        System.out.println(" years old.");
    }
}
```

Examining ConsoleApp

- Compiling your program is the same as with an applet
- Running is a little different than an applet
  - From the command line: `C:\> java ConsoleApp`
  - From JCreator, select Run File
    - Step-by-step instructions in Lesson 2D

- Differences in structure from an applet
  - No `java.awt` or `java.applet` import statement (not GUI)
  - No `extends Applet` (not an applet)
  - No `init()` method (part of applet lifecycle)
The main() Method

- Biggest difference is the main() method
- All applications (console and GUI) must have main()
  - Must be public (called from outside)
  - Must be static (called without creating an object)
  - Must be void (doesn't return a value)
- Must take a single String[] argument
  - This is an array of Strings
  - Contains command-line arguments
  - You can change the word args if you like

Applets versus Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Applet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile using javac</td>
<td>✓</td>
</tr>
<tr>
<td>Run in Web browser</td>
<td>✓</td>
</tr>
<tr>
<td>Run in appletviewer</td>
<td>x</td>
</tr>
<tr>
<td>Run using java interpreter (java)</td>
<td>x</td>
</tr>
<tr>
<td>Start at main() method</td>
<td>✓</td>
</tr>
<tr>
<td>Start with init(), start(), etc.</td>
<td>✓</td>
</tr>
</tbody>
</table>

Command Line Arguments

- The String[] args argument to main()
- Array of Strings representing the command line
- How arguments are passed to applications
- Let's create a console mode application and take a look:
  - Call the class Args, don't extend anything
  - Loop through the args array, using the length field
  - Print each element to the console
  - Try passing: java one two "three four"
- Introducing Exercise 14A
Lesson B: Graphical (GUI) Applications

- Applications don't have to use console-mode
- You can write applications using the same elements that you've used in your applets: Button, Label, etc.
- These GUI applications need a "top-level" window to house them
- This will be a Frame or Dialog

Meet the Frame

- Frame is a top-level window that hosts an application
  - You don't have to use Frame in your applets
  - The Web browser (or appletviewer) provides a host
- There are only two Frame constructors:
  - Frame()
  - Frame(String title)
- The first creates an untitled window, the second adds a title
- Often, for a GUI app, you'll extend the Frame class

Your Basic Graphic App I

- Let's use JCreator to create a "basic" GUI application
  - Create a file named GUIApp.java
  - Extend the Frame class instead of Applet
  - Import the awt and awt.event packages, not applet

```java
import java.awt.*;
import java.awt.event.*;
public class GUIApp extends Frame {
    // Code goes here
}
```
Your Basic Graphic App II

- Add a constructor that takes a String argument
  - Inside the constructor, call the Frame(String) constructor
  - To do that, you use the method named super()
  - super() can only appear as first line in a constructor

```java
public GUIApp(String title)
{
    // Call the Frame(String) constructor
    super(title);
}
```

Your Basic Graphic App III

- Now write the main() method
  - Inside the method, create a new GUIApp variable
  - Name your variable app
  - Call the GUIApp constructor to initialize your object

```java
public static void main(String[] args)
{
    GUIApp app = new GUIApp("The GUIApp");
}
```

What's the Problem?

- When you run GUIApp, nothing happens at all. Why?
  - The default size for a Frame is 0 x 0 (kinda small, huh?)
  - Add this to the constructor to fix:
    ```java
    setSize(320, 240);
    ```
  - Frames are not visible by default
  - Use the Window show() method in main() to fix
    ```java
    app.show();
    ```
  - Compile and run. What happens?
Uh Oh!
- Everything works fine, except the window doesn't close
  - Because Frame doesn't listen for "WindowEvents"
  - Use WindowAdapter to make the application end

```java
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent we)
    {
        dispose();
        System.exit(0);
    }
});
```

Using GUIApp I
- Let's create a new application using GUIApp
  - Has a text area to type in, and a "Clear Text" button
  - Place GUIApp.java or GUIApp.class in same folder
  - extends GUIApp, implement ActionListener

```java
import java.awt.*;
import java.awt.event.*;

public class FirstGuiApp extends GUIApp
    implements ActionListener
{
    // Code goes here
}
```

Using GUIApp II
- Add a TextArea instance variable named ta
  - Other objects will be local variables
- Add a constructor
  - Must call the superclass constructor
  - Create and hook up a Button named clear
  - Add clear to the bottom of the application
    - Default Frame layout is BorderLayout()
  - Add the TextArea to the center of the application
Using GUIApp III

- Your added code should look like this:

```java
private TextArea ta = new TextArea();
public FirstGuiApp(String title)
{
    super(title);
    Button clear = new Button("Clear Text");
    clear.addActionListener(this);
    add(clear, BorderLayout.SOUTH);
    add(ta, BorderLayout.CENTER);
}
```

Using GUIApp IV

- Respond to events triggered by the clear Button
  - For this you have to write an `actionPerformed()` method
  - In the method, erase the text in the `TextArea`

```java
public void actionPerformed(ActionEvent ae)
{
    ta.setText(""");
}
```

- Add a `main()` method that creates and show the app
- Introducing Exercise 14B

Lesson C: Java and Images

- How can I use images in my Java programs?
  - Java will display GIF or JPEG images
  - Other formats will have to be translated
  - Applets can load images only from machine where Web-page was loaded
  - Applications can use both local and remote images
- To use an image you must do two things:
  - Load the image
  - Display the image
Loading Images I

To load an image in an applet, you:

1. Create an `Image` instance variable
   ```java
   private Image img = null;
   ```

2. Load the image from disk into memory
   ```java
   im = getImage(getCodeBase(), imgName);
   ```

Load an image in an application requires the use of the `Toolkit` class and its `getImage()` method.

---

Loading Images II

The `getImage()` method takes 2 arguments

- A URL where the image can be found
  - For applets, must be on same server as Java applet
  - A URL is not a String, but a Java class
  - Easiest way to get one is to use one of the methods `getCodeBase()` or `getDocumentBase()`
    - Works best when applets and HTML are in same folder
- File name passed as a `String`
  - The filename may contain path info, just like a URL

---

Drawing Images

To draw an image, you use `drawImage()`

Four arguments to simplest version of `drawImage()`:

- `Image`: variable containing loaded picture to display
- `x, y`: upper-left corner of image to be drawn
- `ImageListener`: just pass this or null

```java
g.drawImage(img, 0, 0, this);
```

Returns `true` if image could be drawn successfully
A Quick Example

- Let's create simplest possible image drawing program
- Open JCreator, copy a GIF or JPG file into project folder
- Create Java applet skeleton named Image1.java
- Add an Image instance variable named img
- Initialize it inside your init() method, using getImage()
- Use drawImage() inside paint() to display the image
  - If drawImage() returns false, repaint() after 100 ms

Other Versions of drawImage()

- Version 1: Coloring transparent regions [Image2]
- Version 2: Resize the image [Image3]
  ```java
  g.drawImage(img, 0, 0, width, height, this);
  ```
- Version 3: Transformation [Image4]
  ```java
  g.drawImage(img,
              dx1, dy1, dx2, dy2, // destination
              sx1, sy1, sy2, sy2, // source
              this);
  ```

Other Image Methods

- Width and height of image
  ```java
  int width = im.getWidth(listener);
  int height = im.getHeight(listener);
  ```
  - Listener can be this or null
  - Both methods return -1 if Image has not yet “arrived”
- copyArea() faster than drawImage() [CopyArea]
  ```java
  g.copyArea(x1, y1, width, height, x2, y2);
  ```
Waiting For Images

- Images are loaded asynchronously
  - Programs rely on return value of `drawImage()`
  - Can result in a "busy wait" that acts like an endless loop
- Better solution is to wait until image arrives before painting
- Java API has two ways to do this:
  - Override the `imageUpdate()` method in class that loads
  - Very fine grain control: read about it on page 548-550
  - Use `MediaTracker` class to force your program to wait

MediaTracker I

- Class that monitors image loading: a 3-step process
  - Create a `MediaTracker` object
    ```java
    MediaTracker mt = new MediaTracker(this);
    ```
  - Add your image to the `MediaTracker`'s "watch list"
    ```java
    mt.addImage(img, 0);
    ```
  - The second argument is an ID that identifies the image
  - You can use the same ID for several images

MediaTracker II

- Use the `MediaTracker` monitoring methods
  - Put the code inside a try-catch block
  ```java
  boolean loadOK = false;
  try {
    mt.waitForID(0, 1000); // allow 1 second
    loadOK = mt.isErrorAny();
  } catch (InterruptedException ie) { }
  ```
- Introducing Exercise 14C
Lesson D: Threads & Animation

- Simple animation is possible by:
  1. Drawing your image in `paint()`
  2. Waiting a short period using `Thread.sleep()`
  3. Updating the x, y location for the next “frame”
  4. Calling `repaint()`

- Let’s walk through a “bouncing ball” applet that does this:
  - Create a new file in JCreator named `FlyingTiger.java`
  - Copy the code from `Image5.java` and change the name
  - Create an HTML file and make sure you can run it
  - Download `tiger.gif` and place in same folder as files

Building FlyingTiger1

- Step 1: Add these instance variables
  - `appWidth, appHeight` to keep track of width of applet
  - `x, y` to keep track of our tiger
  - `dx, dy` to keep track of how far we want to move
  - `imgWidth, imgHeight` to keep track of image size

- Step 2: In the `init()` method
  - Change the name of the file to “tiger.gif”
  - Set background to black
  - Initialize `appWidth` and `appHeight`
  - Initialize `x` and `y` to (approximately) center of applet

Building FlyingTiger1

- In `paint()` you’ll first move `x, y` then paint the image
- Step 3: Modify the `paint()` method
  - Initialize `imgWidth` and `imgHeight` if less than 0
  - Add `dx` and `dy` to `x` and `y` respectively
  - Check to see if `x` or `y` have moved off screen
    - If so, then change the sign of `dx` or `dy`
  - Draw the image at the new location of `x, y`
  - Use `Thread.sleep()` in a try-catch block
  - Call the `repaint()` method
Some Nifty Tiger Mods

- How do you change the animation speed? [FlyingTiger2]
  - Change either the "sleep" period or the value of dx, dy
- Why the image flicker? [FlyingTiger3]
  - Entire background erased.
  - Override update() method to eliminate
  - Leaves "tiger trails" on page
- Erase only the part that needs erasing [FlyingTiger4]
  - Use clearRect() to erase only tiger part
  - Leaves portions of tiger scattered around screen
- Real solution is double-buffering

Double Buffering

- Use an “off-screen” (in memory) Image object
  - Represents entire applet, not just image part
  - Paint entire applet without erasing background at all
- How you do it [FlyingTiger5]
  - Add two instance variables: an Image and a Graphics
  - Use createImage() to initialize the off-screen buffer
  - Use getGraphics() to init the off-screen graphic context
  - Draw on the off-screen graphics context
  - Finish with one drawImage() passing your buffer
- You’ll use this technique with Homework 14

What is a Thread?

- Threads are subprograms that:
  - “Execute asynchronously in the same context”
  - Use a separate flow of control
  - Have access to all object fields
  - The AWT, for instance, uses a thread for painting
- Threads are thus like methods, except:
  - Two threads can operate simultaneously
- The Thread.sleep() method, pauses the current thread
  - Use this to control the animation [FlyingTiger6]
"Rolling Your Own"

- 5 EZ Steps to creating your own thread
  - Add implements Runnable to class header
  - Create a Thread object as a field
  - Pass this to the Thread constructor
  - Start execution by calling the thread's start() method
  - Write a run() method to carry out the actions
    - The run() method is typically an endless loop
    - Use a boolean flag to drop out of the loop
  - Let's look at an example: [Butterfly]

Butterfly I

- Step 1: implement the Runnable interface
  - public class Butterfly extends Applet
    implements Runnable

- Step 2: create a Thread as an instance variable
  - volatile says it can be modified by another Thread
  - private volatile Thread animator = null;

Butterfly II

- Step 3: create and initialize in the start() method
  - animator = new Thread(this);
    - animator.start();

- Step 4: "kill" the thread in stop() by setting to null
  - animator = null;
Butterfly III

- Step 5: Add the run() method required by Runnable
  - We'll have code similar to the paint() method
  - Write an "endless" loop by testing like this:

```java
Thread current = Thread.currentThread();
while (loadOK && current == animator)
{
    // painting code
}
```

- Introducing Exercise 14D

Lesson E: AudioClip

- Java supports playing sounds from a file
- AudioClip interface specifies very basic sound support
  - Java 1.1 only plays sound files in Sun's .au format
  - Java 2 browsers play .wav and .mid formats
  - Java 2 browsers also support the Java Sound API
- Load a sound file with getAudioClip()
  - Same arguments as getImage()
  - Waits for the clip to be loaded (unlike getImage())
  - Returns null if the clip can't be loaded
  - Can only load clips from same host (like images)

Playing Sounds

- The AudioClip interface only specifies three methods
  - play(): plays a clip one time
  - loop(): plays a clip continuously
  - stop(): stops playing a clip (usually a looped clip)
- Typical uses: [Butterfly2]
  - Load your clips during the init() method
  - Control background music with loop() in start(), stop()
  - Use play() when a particular even occurs
- Introducing Exercise 14E
Finish Up

- Assignments due by 11:59 PM, Thursday, May 27
- Quiz 14, Lab 14 [ SketchPad ]
- Homework 13 [ OrderPlease II ]
- Homework 14 [ SuperSketch ]
  - Allow users to select an image to paint on
  - Use off-screen double-buffering to make image persistent
  - Add choice to draw freehand, lines, rectangles & ovals