CSE 11: Lecture 11

- ✔ Binary file I/O: DataInputStream, DataOutputStream
- ✔ Comparing text and binary I/O
- ✔ Buffering
- ✔ The File class
- ✔ GUI programming, Swing, JFC, and the AWT
- ✔ Components and Containers
- ✔ JPanel and Graphics objects
- ✔ JLabels, JButtons, and JTextFields

(Reading: Savitch, Ch. 9 and Ch. 12)

I/O streams

- ✓ Data, whether it is in a file or in main memory, consists of a sequence of bytes
- ✓ When such a sequence moves from your program to some destination, it is called an "output stream"
- ✓ When such a sequence moves from some source into your program, it is called an "input stream"
- ✓ If all the bytes in the stream have values that are to be interpreted as ordinary characters (as you might type at a keyboard), the stream can be considered a "text" stream
- ✓ If the bytes have arbitrary values, the stream is considered a "binary" stream
- ✓ Since you want to treat a text stream differently from a binary one in some ways, different classes are provided, with different methods, for dealing with the two types of streams
- ✓ However, keep in mind that they are really just sequences of bytes!

Binary file I/O

- So far we have discussed how to do I/O with text files: files whose contents are to be interpreted as text characters, usually organized into lines. These are the kinds of files you can create, read, and modify using a text editor
- Next we will talk about how to do I/O with binary files: files that contain data that may not correspond to printable characters at all
- Why use binary files? They cannot be easily viewed with a text editor, but...
- Binary files hold data in a compact form:
    - To store the integer 1,234,567,890 in a text file takes at least 10 bytes (one for each character in its printable representation)
    - To store that int (or any int!) in a binary file takes 4 bytes
- Binary files are fast:
    - Binary file I/O does not require converting to or from a character representation of a value, translating according to a Unicode localization encoding, etc.

DataInputStream and DataOutputStream

- ✔ **`DataOutputStream`** defines methods for binary output of primitive type values
- ✔ recall that for text file output, we create a PrintWriter that contains a FileWriter... for binary file output, create a DataOutputStream that contains a FileOutputStream:

```
DataOutputStream outStream = new DataOutputStream (
        new FileOutputStream( "mystuff.dat" ));
```

  - ✗ the 1-argument **`FileOutputStream`** constructor truncates; use second argument **`true`** to append
  - ✗ **`FileOutputStream`** constructors throw **`IOExceptions`**

- ✔ **`DataInputStream`** defines methods for binary input of primitive type values
- ✔ recall that for text file input, we create a BufferedReader that contains a FileReader... for binary file input, create a DataInputStream that contains a FileInputStream:

```
DataInputStream inStream = new DataInputStream (
        new FileInputStream( "mystuff.dat" ));
```

  - ✗ the **`FileInputStream`** constructor throws **`FileNotFoundExceptions`**

Some instance methods in DataOutputStream

```
public final void writeBoolean(boolean v) throws IOException;

public final void writeByte(int v) throws IOException;

public final void writeChar(int v) throws IOException;

public final void writeChars(String s) throws IOException;

public final void writeDouble(double v) throws IOException;

public final void writeFloat(float v) throws IOException;

public final void writeInt(int v) throws IOException;

public final void writeLong(long v) throws IOException;

public final void writeShort(int v) throws IOException;

public void close();
```

Some instance methods in DataInputStream

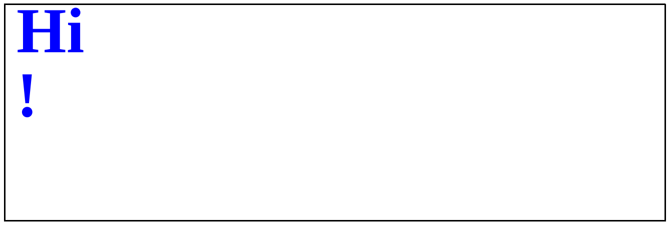
```
public final boolean readBoolean() throws IOException;

public final byte readByte() throws IOException;

public final char readChar() throws IOException;

public final double readDouble() throws IOException;

public final float readFloat() throws IOException;

public final int readInt() throws IOException;

public final long readLong() throws IOException;

public final short readShort() throws IOException;
```

Using DataOutputStream for binary output

- Consider this example:

```
import java.io.*;
public class Test {
   public static void main(String args[]) throws IOException {
      DataOutputStream dos = new DataOutputStream(
            new FileOutputStream("foo.txt"));
      dos.writeByte(72); dos.writeByte(105);
      dos.writeByte(10); dos.writeByte(33);
      dos.close();
   }
}
```

- After this runs, the file foo.txt contains 4 bytes whose values happen to be the ASCII codes for `H`, `i`, newline, and `!` When viewed in a text editor, it would look like



```
Hi
!
```

- .... and the same would result from `dos.writeInt(1214843425);`

End-of-file in binary files

- ✔ When you use one of the DataInputStream methods to read from a file, and the read operation does not succeed because you are at the end of the file, the method throws an **`EOFException`**

- ✔ (Note that this is different from the 2 ways to detect EOF when doing text file input with a BufferedReader...!)

- ✔ This leads to the technique of using an exception to break out of what would otherwise be an infinite loop...

Copying from one file to another, a byte at a time

```
import java.io.*;
public class CopyV3 {
   public static void main( String[] args ) throws IOException {
      System.out.print("Enter input file name: ");
      String inFileName = SavitchIn.readLine();
      System.out.print("Enter output file name: ");
      String outFileName = SavitchIn.readLine();

      DataInputStream in = new DataInputStream(
              new FileInputStream(inFileName));
      DataOutputStream out = new DataOutputStream(
              new FileOutputStream(outFileName));
      try {
      while (true)
         out.writeByte(in.readByte());
      } catch (EOFException e) {}    // catch it, and do nothing

      out.close();
   }
}
```

Binary file I/O: Some additional points

- ✔ Be sure to close a DataOutputStream when you are done with it -- that helps to prevent lost data

- ✔ Note that a binary file containing primitive type values does not store any type information! For example, you can write two `int`s to a file, and then read those 8 bytes back as a `double` (it *will* be some double value... but a meaningless one)

- ✔ You can have a DataOutputStream and a DataInputStream open to the same file... but it may corrupt the data in the file and it is not recommended! (If you want to do that you should use RandomAccessFile)

- ✔ DataInputStreams and DataOutputStreams are only for binary I/O of Java primitive types: to handle class types, use ObjectInputStream and ObjectOutputStream

Review: basic file I/O in Java

- ✔ Text output to a file: wrap a **`FileWriter`** object in a **`PrintWriter`** object, and call the **`PrintWriter`**’s methods to do output.

- ✔ Text input from a file: wrap a **`FileReader`** object in a **`BufferedReader`** object, and call the **`BufferedReader`**’s methods to do input.

- ✔ Primitve type binary output to a file: wrap a **`FileOutputStream`** object in a **`DataOutputStream`** object, and call the **`DataOutputStream`**’s methods to do output.

- ✔ Primitive type binary input from a file: wrap a **`FileInputStream`** object in a **`DataInputStream`** object, and call the **`DataInputStream`**’s methods to do input.

OutputStream vs. Writer: an example

- Consider this code:

```
 double d = 3.1415926535;
DataOutputStream ds
   =  new DataOutputStream(new FileOutputStream("out1"));
 ds.writeDouble(d);
 ds.close();

 PrintWriter pw
   = new PrintWriter(new FileWriter("out2"));
 pw.print(d);
 pw.close();
```

- What is the resulting size of file **out1**? ______ bytes
  - What does it contain?
- What is the resulting size of file **out2**? ______ bytes
  - What does it contain?

Buffering

- The I/O classes with “Buffered” in their names override the read() or write() methods to provide *buffering*

- I/O buffering is the use of an intermediate data structure (called the buffer; usually an array) to hold data items

    - Output buffering: the buffer holds items destined for output until there are enough of them to send to the destination; then they are sent in one large chunk

    - Input buffering: the buffer holds items that have been received from the source in one large chunk, until the user needs them

- The reason for buffering is that it is often more efficient to receive data from a source, or to send data to a destination, in large chunks, instead of one byte at at time

- This is true, for example, of disk files and internet sockets; even small buffers (1K bytes or less), can make a big difference in performance

The File class

- ✔ Sometimes in your code you want to check on the properties of files:
  - ✗ does a file with a certain name exist?
  - ✗ is the file readable or writable by the user running the program?
  - ✗ how big is the file?
  - ✗ what is the full pathname of a file?
- ✔ ... and you may want to delete or rename a file.
- ✔ You can do all these things with instance methods in the **`java.io.File`** class
- ✔ Note that **`File`** methods don’t do any I/O... they are for deleting, renaming, or determining properties of files

Some instance methods of the File class

```
// constructor,  name of file as argument
public File(String filename)

// return true if the File exists
public boolean exists()

// return true if the File is readable
public boolean canRead()

// return true if the File is writable
public boolean canWrite()

// delete the File; return true if successful
public boolean delete()

// rename the File; return true if successful
public boolean renameTo(File dest)

// return the number of bytes in the File
public long length()
```

Terminal, file, and graphical I/O

- So far we have looked at console terminal and file I/O
- Now we will start to look at issues related to graphical I/O and graphical user interface (GUI) programming
- Savitch says about text-oriented, console terminal applications: “Modern programs do not work this way.”
- Actually, lots of modern programs DO work that way, and they can be extremely useful tools
- But graphical applications are important too... so let’s learn about how to write them in Java
- Along the way we will see examples of class derivation, inheritance of methods, interfaces, abstract classes, anonymous inner classes, and event-driven programming

A brief history of GUI libraries in Java

- ✔ In JDK 1.0: the Abstract Window Toolkit (AWT) is introduced as the `java.awt` package. It provides functionality for basic windowing and simple graphics.
- ✔ In JDK 1.1: the AWT is heavily revised to move toward an "event-driven programming" model, and to impove speed. This revision is incompatible with the 1.0 AWT in many places
- ✔ In JDK 1.2: the AWT is extended with a set of classes sometimes referred to as the Java Foundation Classes (JFC), originally code named the "Swing Set"
  - ✗ this involved adding classes and subpackages to `java.awt`, and creating a new package hierarchy under the `javax.swing` package
  - ✗ the new JFC/Swing classes are compatible with the 1.1 AWT, while adding easier-to-use components, "pluggable look-and-feel", and many other enhancements
- ✔ We will concentrate on the JDK 1.2 JFC/Swing approach, which continues in JDK 1.3 and 1.4

Java GUI packages

- The GUI classes we will start talking about now are in the packages `java.awt`, `java.awt.event`, and `javax.swing`
- If you want the convenience of accessing classes in these packages just by referring to them by name, put these import statements at the top of your sourcecode file:

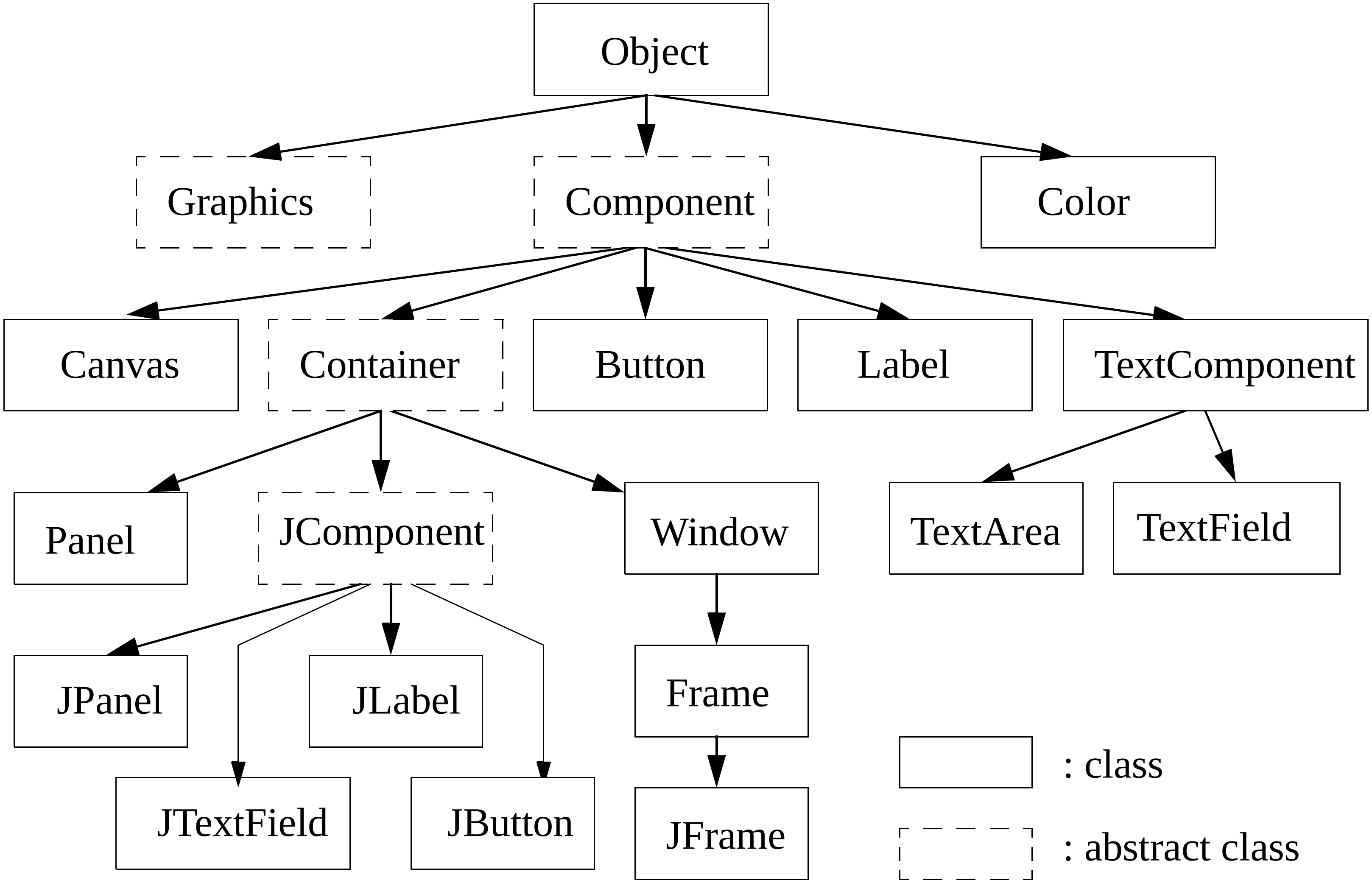
  ```
  import java.awt.*;
  import java.awt.event.*;
  import javax.swing.*;
  ```

  - (The ‘`*`’ means “all the classes in the package”)

- These classes exist in an inheritance hierarchy (see the next slide for some of them)... many JFC/Swing class names are distinguished from similar AWT names by starting with the letter “J”
- Using Swing and the AWT involves creating instances of these classes, and also writing your own classes that subclass (extend) these.

- Some of the Swing/AWT classes are “abstract classes”: you cannot create an instance of an abstract class, but you can create an instance of any nonabstract subclass of it
  - abstract classes often do not include definitions for some of their methods; they are used for the role they play in the Java type system, as base classes

Some of the java.awt and javax.swing classes



Opening a window

- It's pretty easy to write a Java application that creates and opens a window on your display. Here's a simple way of doing it:

```
import javax.swing.*;

public class Win {
   public static void main(String args[]) {

      JFrame w = new JFrame();  // create a JFrame object
      w.setSize(300,200);     // make it 300 pixels wide, 200 high
      w.setVisible(true);     // make it appear

   }
}
```

- `JFrame` is the JFC class that defines a window object, with borders, titlebar, etc. as supplied by your computer's windowing system

- When this program runs, it creates a window that doesn't contain anything and it doesn't do anything! We need to add Components to it, and define methods that do something...

Components and Containers

- ✔ Every **JFrame** object has a "content pane" which is-a **Container**
  - ✗ ... this is returned as the value of the **JFrame**'s **getContentPane()** instance method
- ✔ The **Container** class has a public instance method **add**, which takes as argument a **Component** and adds it to the **Container**
- ✔ Important kinds of Components we will look at adding to a **JFrame**'s content pane are:
  - ✗ **JPanel**
  - ✗ **JButton**
  - ✗ **JTextField**
  - ✗ **JLabel**
- ✔ Note: It is possible to add AWT components to a Swing container, but it is usually best to add only JFC/Swing components, i.e. instances of classes that are descendants of **JComponent**

Graphics and Swing

- ✔ We will talk later about JLabel, JTextField, and JButton objects
- ✔ Those Swing objects ‘know how’ to display themselves... you specify a String for them to display, add them to a Container, and they take care of the rest
- ✔ But for some GUI applications, you want to draw your own graphics on the display, under program control
- ✔ A good way to do this is to create an instance of the `JPanel` class, add it to your application JFrame’s and draw on the JPanel
    - It is possible to draw directly on a JFrame, but a JFrame does a lot of its own “drawing”, which may interfere with what you are trying to do
    - In fact, you can draw on any Component! But JPanel is a simple, undecorated component that is a good choice for doing general graphics
- ✔ So, let’s look at drawing on a JPanel...

The java.awt.Graphics class

- ✔ To draw on a Component, such as a JPanel object:
  - ✗ you first obtain a Graphics object that is associated with the Component
  - ✗ ... then you call methods of that Graphics object to actually do the drawing!

- ✔ There are two ways to get a Graphics object for a Component:
  - ✗ ...call the Component's **`getGraphics()`** instance method

    ```
    JPanel p = new JPanel();
    Graphics g = p.getGraphics();
    ```

    - If you do this, call **`g.dispose()`** when you are done, to release resources

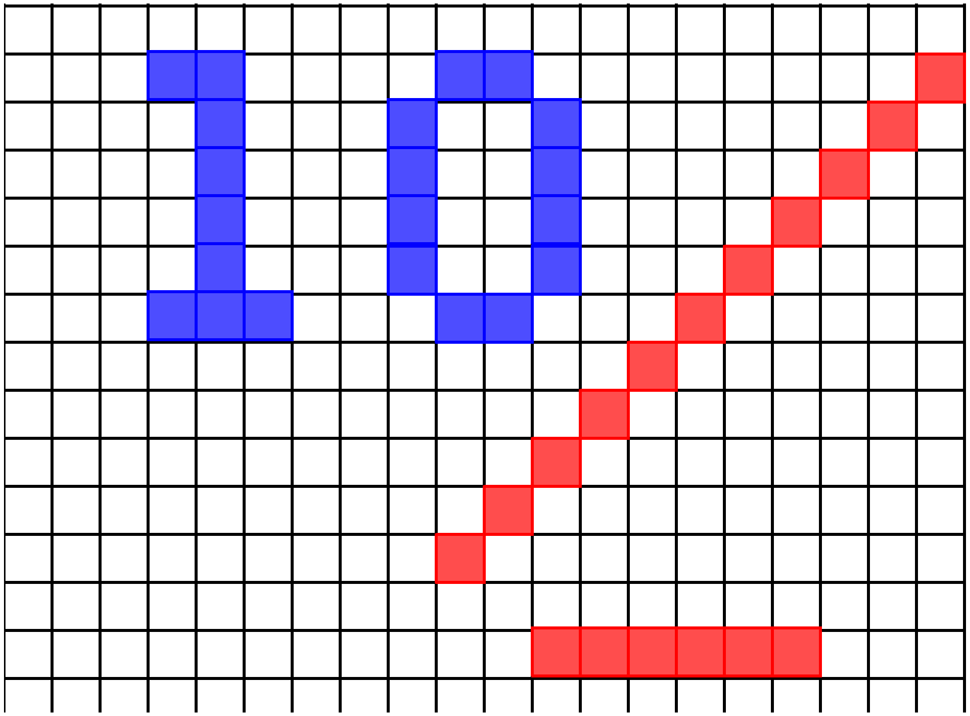
  - ✗ ...subclass JPanel and override the instance method **`paintComponent(Graphics)`**

    ```
    public class AppPanel extends JPanel {
       public void paintComponent (Graphics g) {
       // when paintComponent is called, g will be the Graphics
    ```

    - The **`paintComponent`** method will be called automatically when the system detects that the JPanel needs to be redrawn
    - ... or you can call the **`repaint()`** method of a JPanel to force this... but don't call the paintComponent() method directly in your own code.

Components and Pixels

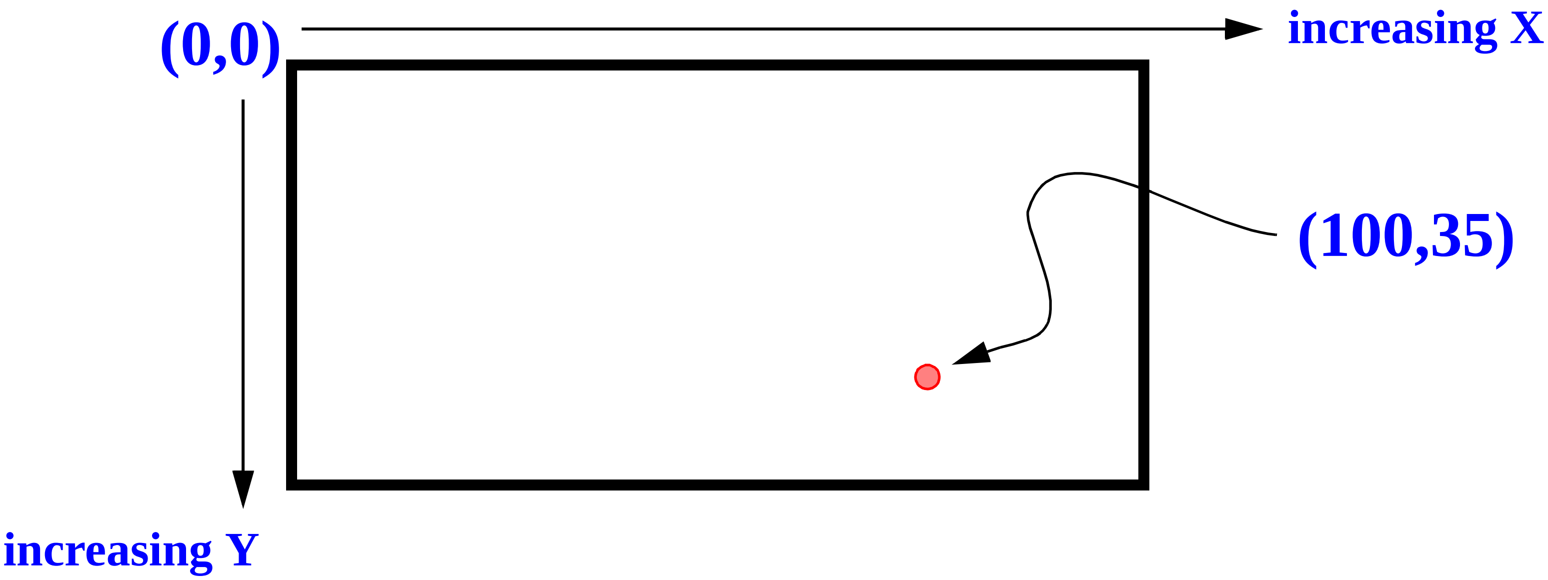
- ✔ Like your computer CRT display screen itself, a Component is divided into a rectangular array of *pixels*
- ✔ A pixel is small: on a typical CRT, there are about 5000 per square inch
- ✔ On a monochrome display, each pixel is either on or off (white or black); on a color display, each pixel can be a particular color
- ✔ The pattern of pixels determines the visual appearance of a Component



Component coordinates

- ✔ To draw on a Component, it is important to understand the AWT coordinate system!

- ✔ The Java AWT uses a Cartesian (x,y) coordinate system in which:
  - ✗ x=0, y=0 are the coordinates of the pixel in the upper-left corner of the Component
  - ✗ x increases to the right, y increases downward: x=0, y=1 is the pixel below x=0,y=0; x=1,y=0 is the pixel to the right of x=0,y=0; etc.



AWT Colors

- ✔ To do pretty things on a window, it also helps to understand Colors
- ✔ a java.awt.Graphics object has an instance method setColor that determines the color that will be used for subsequent drawing:

```
public void setColor (Color c)
```

- ✔ The java.awt.Color class has a constructor that creates a color made up of the given red, green, and blue components (each in the range 0 through 255):

```
public Color(int r, int g, int b)
```

- ✔ ... and the Color class has various public static final constant Colors that you can use, with names as shown on the next slide

Color constants

```
public static final Color black;
public static final Color blue;
public static final Color cyan;
public static final Color darkGray;
public static final Color gray;
public static final Color green;
public static final Color lightGray;
public static final Color magenta;
public static final Color orange;
public static final Color pink;
public static final Color red;
public static final Color white;
public static final Color yellow;
```

Useful Graphics methods

- ✔ Some instance methods of the Graphics class that are useful for doing... graphics

```
// display the String, starting at x,y
public void drawString(String str, int x, int y);

// draw a line from x1,y1 to x2,y2
// (just draw a single pixel if x1==x2 and y1==y2)
public void drawLine(int x1, int y1, int x2, int y2);

// draw a rectangle with upper left corner at x,y
public void drawRect(int x, int y, int width, int height);

// draw a filled (solid) rectangle with upper left corner at x,y
public void fillRect(int x, int y, int width, int height);

// draw an ellipse whose bounding rectangle has U.L.C. at x,y
// (a circle if width == height)
public void drawOval(int x, int y, int width, int height);

// draw a filled ellipse whose bounding rectangle has U.L.C. at x,y
public void fillOval(int x, int y, int width, int height);
```

Using a JPanel for graphics: an example

- Define a class that extends JPanel, and override the paintComponent() method:

```
import java.awt.*;
import javax.swing.*;
public class MyPanel extends JPanel {
// paintComponent() will be called when the JPanel needs redrawing
  public void paintComponent(Graphics g) {
    g.setColor(Color.red);
    g.drawLine(0,0,60,60);

    g.setColor(Color.blue);
    g.fillRect(30,30,10,40);

    g.setColor(Color.green);
    g.drawOval(0,90,50,25);

    g.setColor(Color.black);
    g.drawString("over here", 150, 150);

    g.drawLine(1000,-3456,-1100,5000);
  }
```

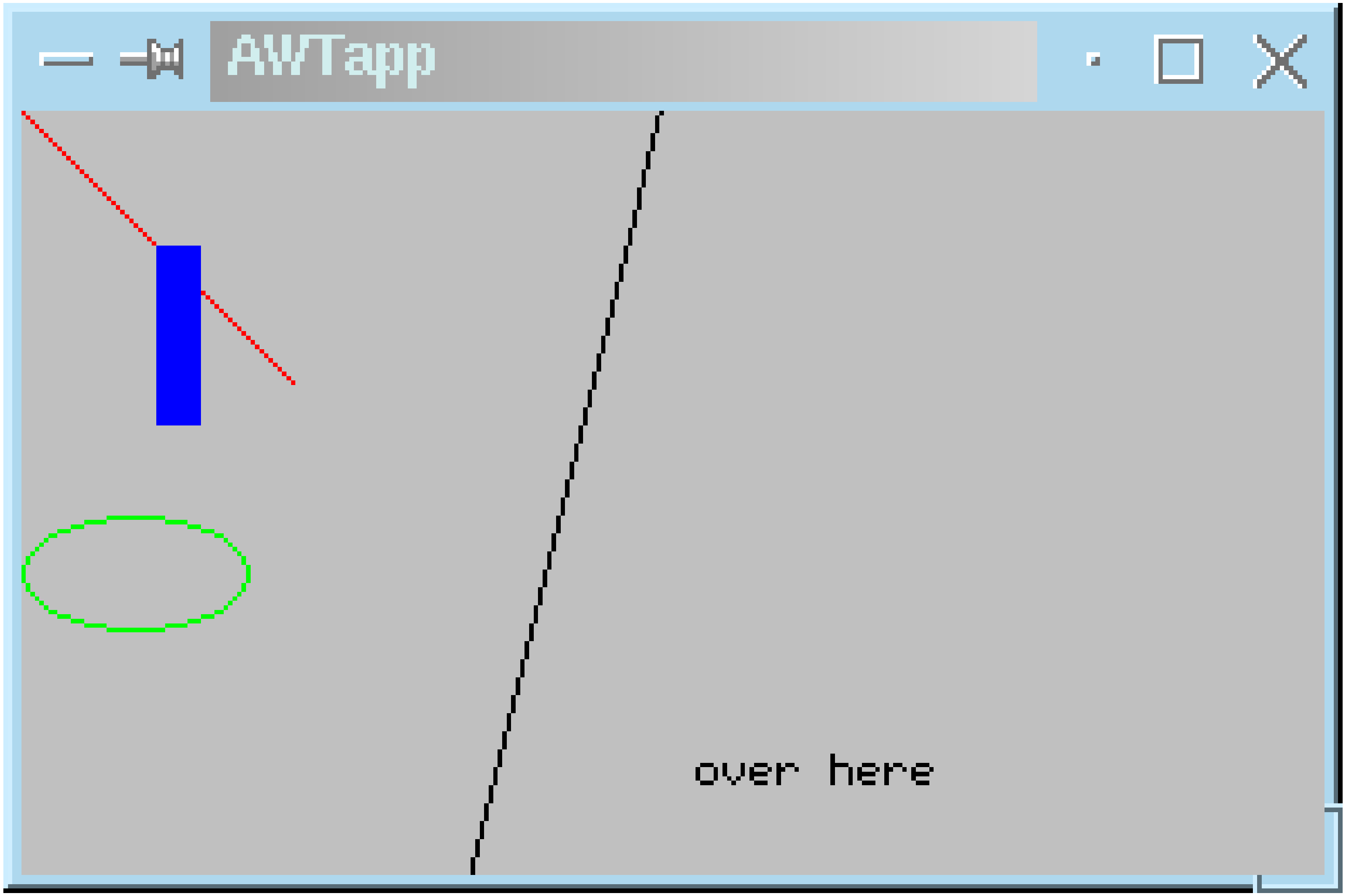
a JPanel example, cont’d

- ... and consider this program:

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

   JFrame app = new JFrame();

   app.setSize(300,200);

   MyPanel p = new MyPanel();

   app.getContentPane().add(p);

   app.setVisible(true);

}
```

- When the JFrame becomes visible for the first time, the paintComponent() methods of all its components are automatically called. This creates a window as shown on the next page...

An application JFrame with a painted JPanel component



Other Swing Components

- A JPanel is a kind of Swing component good to use for general drawing
- There are other Swing components that are readymade for more specific purposes, and that define paintComponent() to display themselves in a nice way. Some of these are:
  - **`JButton`**: a GUI button that displays a text label, and that can be clicked on.
  - **`JTextField`**: a small window displaying text that can be edited by the user.
  - **`JLabel`**: used to display a single line of text, not editable by the user
- There are many more than these in the JFC; but we will look at these

The JButton Component

- ✔ A JButton represents a raised GUI pushbutton that displays some specified text on it. It is intended to be clicked on by the user, to cause some event to happen when clicked on...

- ✔ The text in a JButton can be set by calling the **`setLabel`** method of the JButton

  (you can also do this by passing a String to the JButton constructor)

- ✔ The text in a JButton can be retrieved by calling the **`getLabel`** method of the JButton

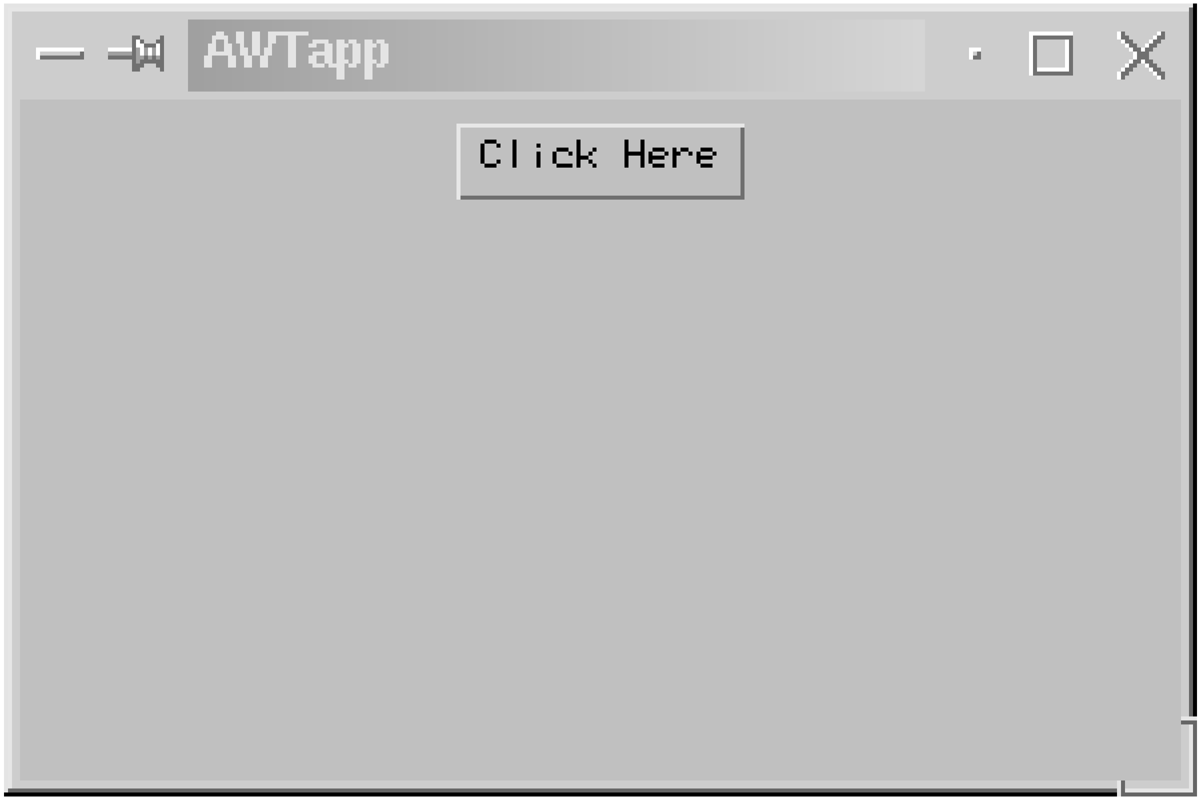
Adding a JButton to a JFrame

- ✔ To display a JButton, create it and add it to a JFrame’s contentPane. Simple!

```
import javax.swing.*;

public class WinB {
   public static void main(String args[]) {

      JFrame w = new JFrame();
      w.setSize(300,200);

      JButton b = new JButton("Click Here");  // create a JButton

      w.getContentPane().add(b);    // add the Button to the JFrame

      w.setVisible(true);           // make the JFrame appear

   }
}
```

A window displaying a JButton

- When run, the program on the previous slide will display a window like the one shown here:



- The text in the JButton cannot be edited by the user
- JButtons can also display images and graphics, and can have various shapes (not just rectangles)

The JLabel Component

- ✓ A JLabel is intended to display a single line of text
- ✓ The text in a JLabel can be set by calling the **`setText`** method of the JLabel

  (you can also do this by passing a String to the constructor, but setText() can change the label after it has been created)
- ✓ The text in a JLabel can be retrieved by calling the **`getText`** method of the JLabel
- ✓ The text in a JLabel cannot be edited by the user in the window in which it appears, so its **`getText`** method is not all that useful

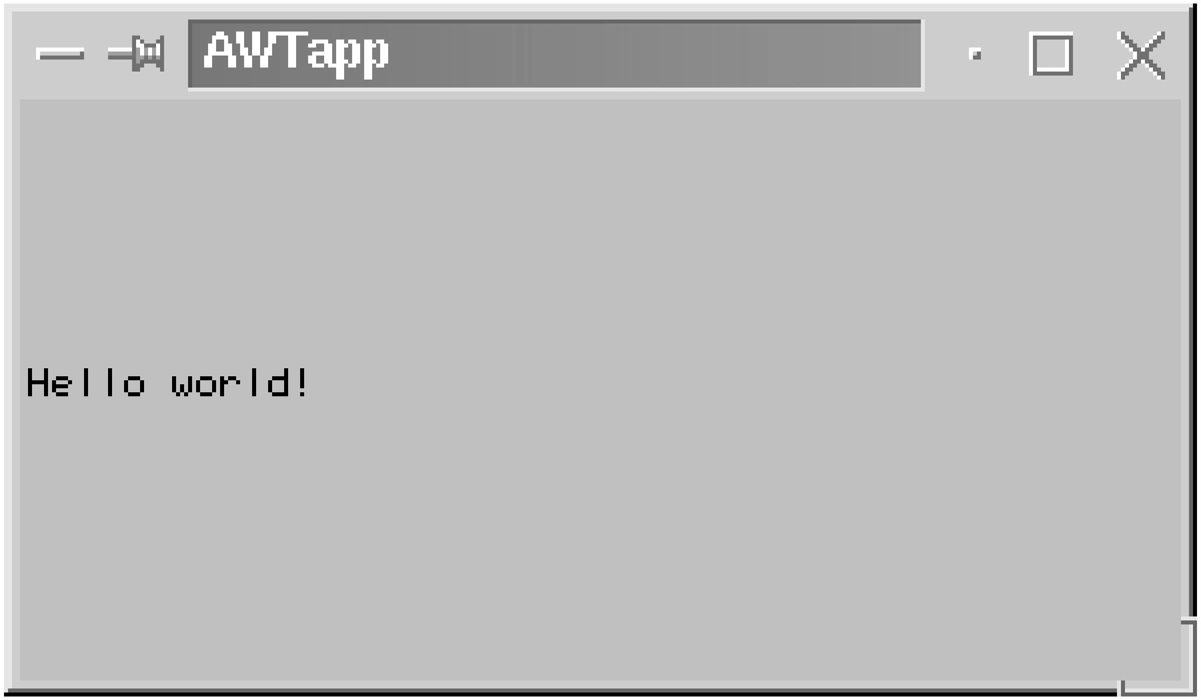
Adding a JLabel to a JFrame

- Using a JLabel permits writing a simple “Hello World!” GUI application

```
import javax.swing.*;

public class WinL {
   public static void main(String args[]) {

      JFrame w = new JFrame();  // create a Frame object
      w.setSize(300,200);     // make it 300 pixels wide, 200 high

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

      w.getContentPane().add(lab); // add the Label to the Frame

      w.setVisible(true);          // make the Frame appear

   }
}
```

A window using a Label

- ✔ When run, the program on the previous slide will display a window like the one shown here:



Hello world!

The JTextField Component

- ✔ A JTextField is intended to display a single line of text that can be edited by the user
- ✔ The text in a JTextField can be set by calling the **`setText`** method of the JTextField

  (you can also do this by passing a String to the constructor, but setText() permits changing the text after the JTextField is created)
- ✔ The text in a JTextField can be retrieved by calling the **`getText`** method of the JTextField
- ✔ Since text in a TextField can be edited by the user in the window in which it appears, the **`getText`** method is quite useful here

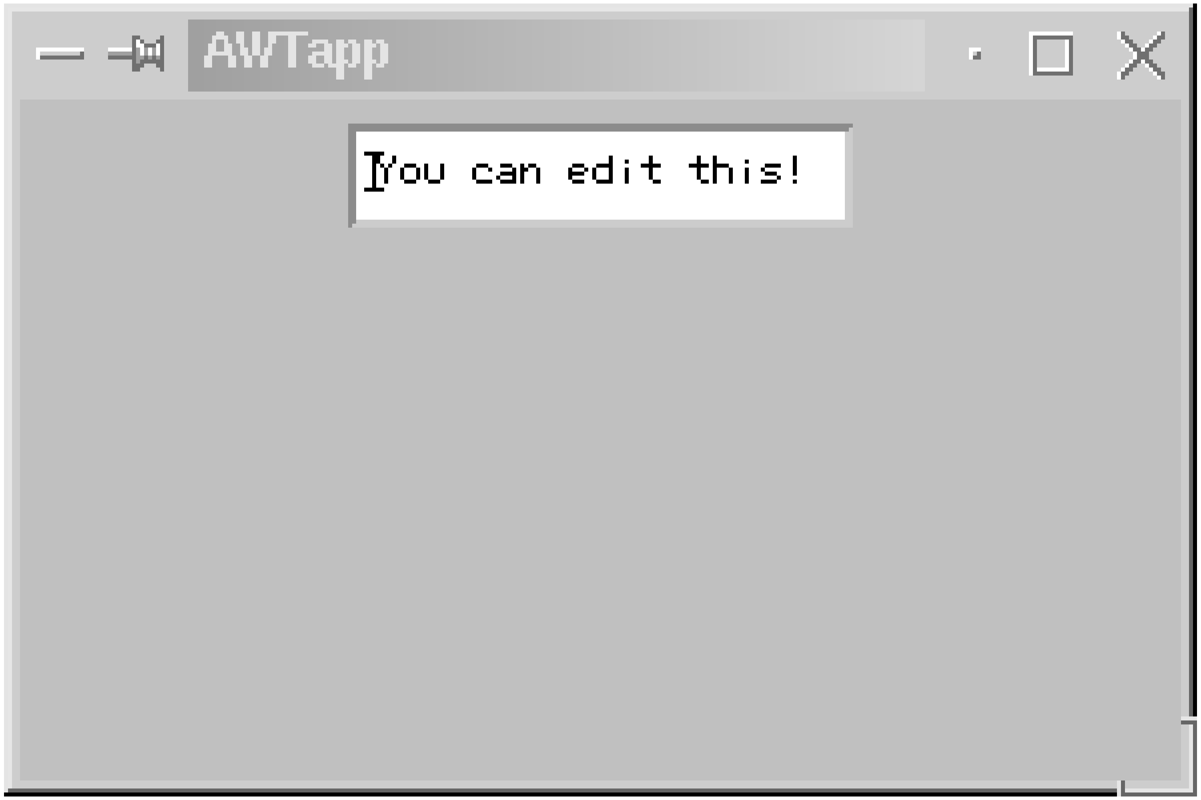
Adding a TextField to a Frame

- A TextField is like a Label, except the text displayed in it can be edited by the user

```
import javax.swing.*;

public class WinT {
  public static void main(String args[]) {

   JFrame w = new JFrame();
   w.setSize(300,200);

   w.getContentPane().setLayout(new java.awt.FlowLayout());

   JTextField tf = new JTextField();  // create a TextField
   tf.setText("You can edit this!"); // set its text

   w.getContentPane().add(tf);  // add the JTextField to the JFrame

   w.setVisible(true);             // make the Frame appear

   }
}
```

A window using a TextField

- ✔ When run, the program on the previous slide will display a window like the one shown here:



You can edit this!

- ✔ The text in the TextField can be edited by the user
- ✔ When the TextField's getText method is called, it returns the contents of the field as a String

Layout managers

- ✔ If you add a Component to a Container, how do they get arranged in the Container?
- ✔ An if you resize the Container, how are the Components in it resized or rearranged?
- ✔ In Swing and AWT, these are the jobs of a `LayoutManager` object
- ✔ There are three main types of `LayoutManager` we'll talk about (there are others):
  - ✗ `FlowLayout`
  - ✗ `GridLayout`
  - ✗ `BorderLayout`
- ✔ Different Containers have different default LayoutManagers
- ✔ To associate a LayoutManager with a Container (e.g. a JFrame) so that it governs where Components are placed in the Container, create a LayoutManager of the appropriate type and pass it to the `setLayout` method of the Container's contentPane:

```
JFrame f = new JFrame();
// make f use FlowLayout for management
f.getContentPane().setLayout(new FlowLayout());
```

- ✔ Different kinds of LayoutManager work differently, as we will see next time

Next time

- ✓ Layout Managers
- ✓ Event-driven programming
- ✓ Listeners and events
- ✓ ActionEvent and WindowEvent
- ✓ ActionListener, WindowListener, and WindowAdapter

(Reading: Savitch, Ch. 12)