

# Welcome to CIS 068 !

1. GUIs: JAVA Swing

2. (Streams and Files we'll not cover this in this semester, just a review)

# Overview

- **JAVA and GUIs: SWING**
  - Container, Components, Layouts
  - Using SWING
- **Streams and Files**
  - Text Files, Binary Files

# The First Swing Program

Example:

## The First Swing Program



# The GUI

Container: JFrame

Layout: BorderLayout

Components:



JLabel

JButton, containing  
an ImageIcon

# Steps to build a GUI

1. import package
2. set up top level container  
(e.g. JFrame)
3. apply layout  
(e.g. BorderLayout)
4. add components  
(e.g. Label, Button)
5. REGISTER listeners
6. show it to the world !



# The Source

1. import package
2. set up top level container  
(e.g. JFrame)
3. apply layout  
(e.g. BorderLayout)
4. add components  
(e.g. Label, Button)
5. REGISTER listeners
6. show it to the world !

```
3 import javax.swing.*;
4 import java.awt.BorderLayout;
5 import java.awt.event.*;
6
7 class BearButtonTest extends JFrame
8 implements ActionListener
9 {
10     JLabel theLabel;
11     int counter = 0;
12
13     // the constructor
14     public BearButtonTest(){
15         // create Container
16         super("BearButtonTest"); // for the window's name only
17         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18
19         // if you don't like the default layout: apply new layout
20         getContentPane().setLayout(new BorderLayout());
21
22         // create graphical components
23         // 1. button
24         JButton theButton = new JButton();
25         theButton.setIcon(new ImageIcon("button.jpg"));
26         theButton.setPressedIcon(new ImageIcon("buttonPressed.jpg"));
27         // 2. Label
28         theLabel = new JLabel(" Hi.");
29
30         // add Components to Container
31         // specify the area (eg. "Center") if needed and/or possible
32         getContentPane().add("North", theLabel);
33         getContentPane().add("Center", theButton);
34
35         // Register Listeners to Event - Sources
36         // Here: the source is the button, the listener is THIS object
37         theButton.addActionListener(this);
38
39         // show it to the world !
40         pack(); // automatically adjusts sizes etc.
41         setVisible(true); // show it !
42     }
43 }
```

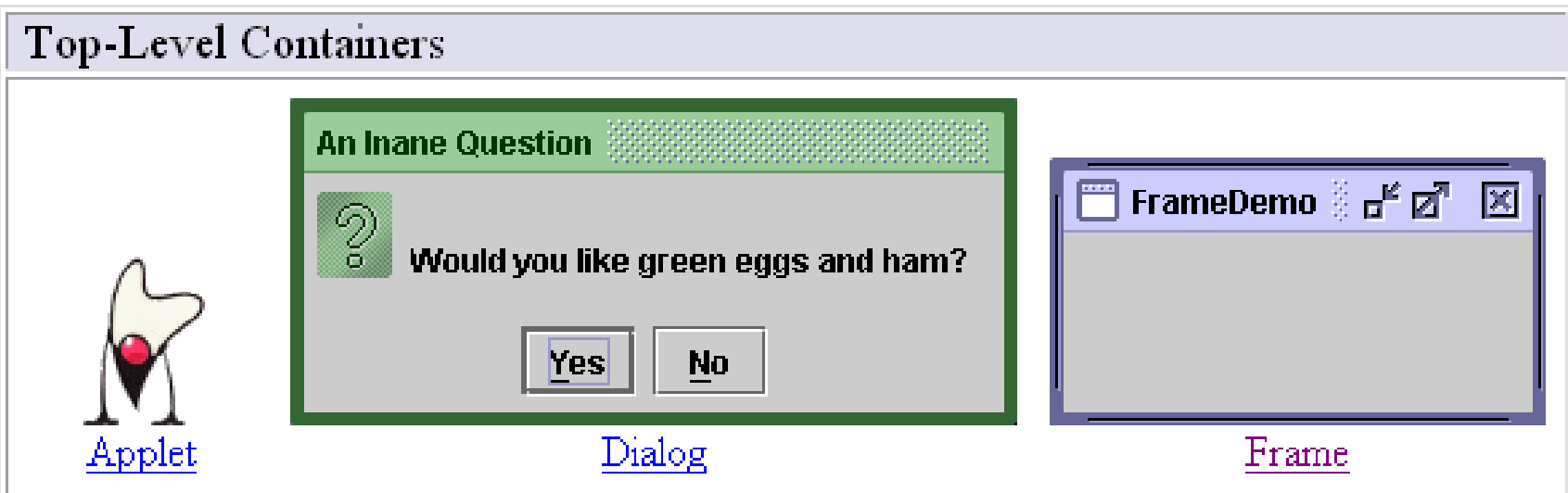
# Swing Components

- Top Level Containers
- General Purpose Containers
- Special Purpose Containers
- Basic Controls
- Uneditable Information Displays
- Interactive Displays of Highly Formatted Information

# Swing Components

## Top Level Containers

Top-Level Containers



Applet

Dialog

Frame

Your application usually extends one of these classes !



# Swing Components

## General Purpose Containers


General-Purpose Containers

A Label on a Panel

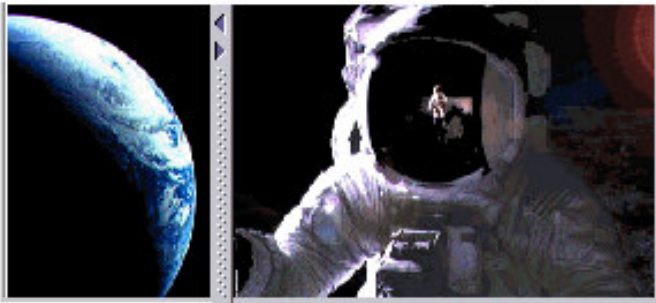
Color and font test:

- ◆ red
- ◆ blue
- ◆ green
- ◆ small


[Panel](#)




[Scroll pane](#)



[Split pane](#)



[Tabbed pane](#)

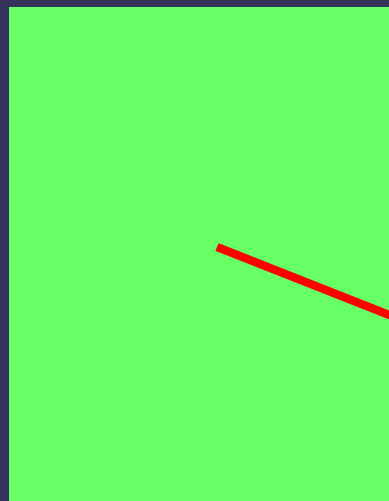


[Tool bar](#)

# Swing Components

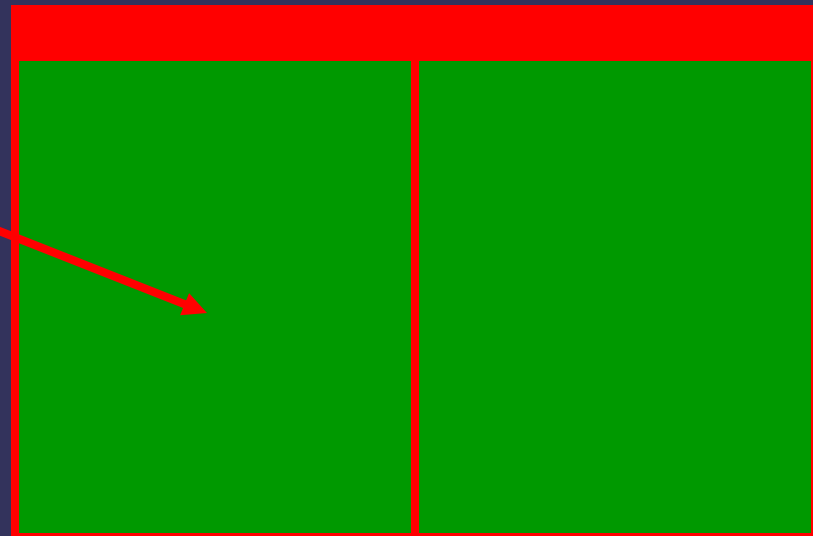
## General Purpose Containers

- typically used to collect Basic Controls (JButton, JCheckBox...)
- Added to layout of top-level containers



JPanel

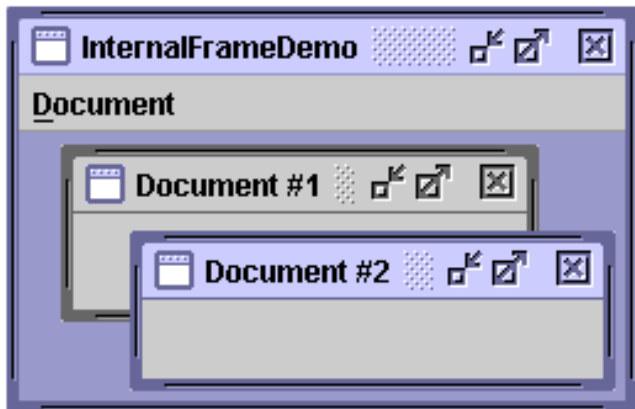
JFrame



# Swing Components

## Special Purpose Containers

Special-Purpose Containers



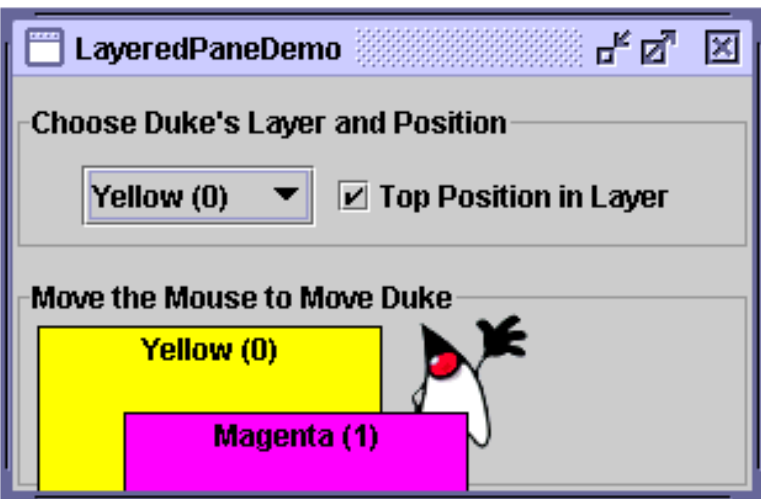
InternalFrameDemo

Document

Document #1

Document #2

Internal frame



LayeredPaneDemo

Choose Duke's Layer and Position

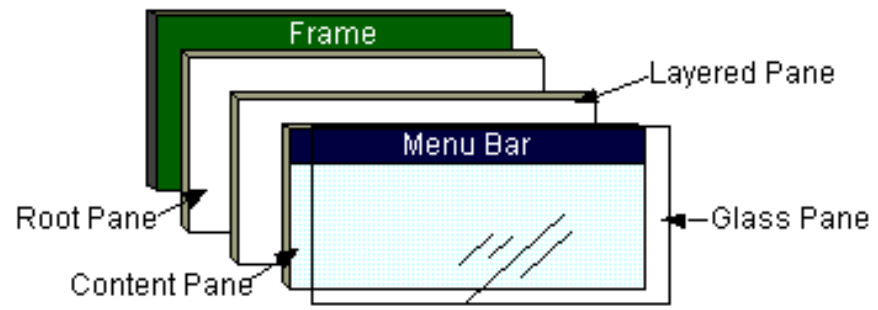
Yellow (0)  Top Position in Layer

Move the Mouse to Move Duke

Yellow (0)

Magenta (1)

Layered pane



Frame

Layered Pane

Menu Bar

Root Pane

Content Pane

Glass Pane

Root pane

The diagram illustrates the Swing component hierarchy. It shows a stack of components: a green 'Frame' at the top, followed by a white 'Root Pane', a white 'Content Pane', a dark blue 'Menu Bar', and a light blue 'Glass Pane' at the bottom. A 'Layered Pane' is shown as a separate component that can be layered on top of the 'Content Pane'. Arrows point from labels to each of these components.

# Swing Components

## Special Purpose Containers

- If you want to use them, go to [java.sun.com](http://java.sun.com)

# Swing Components

## Basic Controls

Basic Controls

The image displays a window titled "Basic Controls" containing several standard Java Swing components:

- Buttons:** A panel with four icons (a pencil, a paper, a pencil with a yellow highlighter, and a blue folder), a checked checkbox labeled "Check 1", a radio button labeled "Radio 2", and an "OK" button.
- Combo box:** A dropdown menu with "Pig" selected in the top box and a list of options: "Bird", "Cat", "Dog", "Rabbit", and "Pig". A mouse cursor is over "Cat".
- List:** A list box containing the months "January", "February", "March", and "April". "March" is selected.
- Menu:** A menu bar with two menus: "A Menu" and "Another Menu". Under "A Menu", there are five items: "A text-only menu item" (with "Alt-1" as a mnemonic), "Both text and icon" (with a yellow star icon), "A radio button menu item" (with a radio button), "A check box menu item" (with a checkbox), and "A submenu" (with a right-pointing arrow).
- Slider:** A slider control titled "Frames Per Second" with a range from 0 to 30. The slider knob is positioned at approximately 15.
- Spinner:** A spinner control showing the number "20".
- Text field or Formatted text field:** A text field labeled "Years:" containing the number "30".

[Buttons](#)      [Combo box](#)      [List](#)

[Menu](#)      [Slider](#)

[Spinner](#)      [Text field](#) or [Formatted text field](#)

# Swing Components

## Basic Controls

- Unlike 'passive' containers, controls are the 'active' part of your GUI

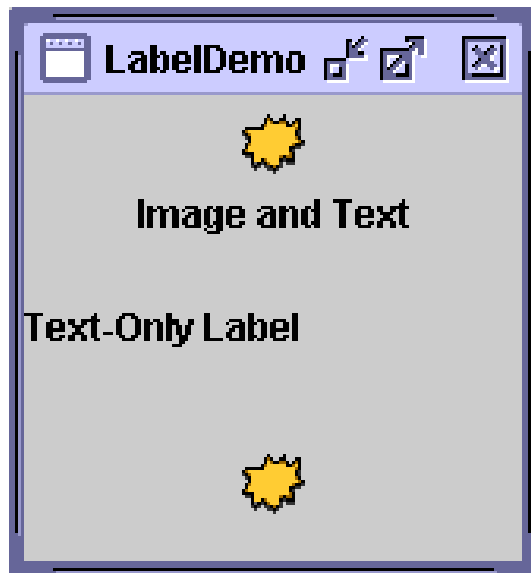
Remark: containers aren't only 'passive', they are also 'active' sources of events, eg. Mouse-events.

- Being the visible part of your interface, controls bring your application to life
- Controls are event sources !
- Objects of your application register to controls to handle the events

# Swing Components

## Uneditable Information Displays

### Uneditable Information Displays



Label



Progress bar



Tool tip

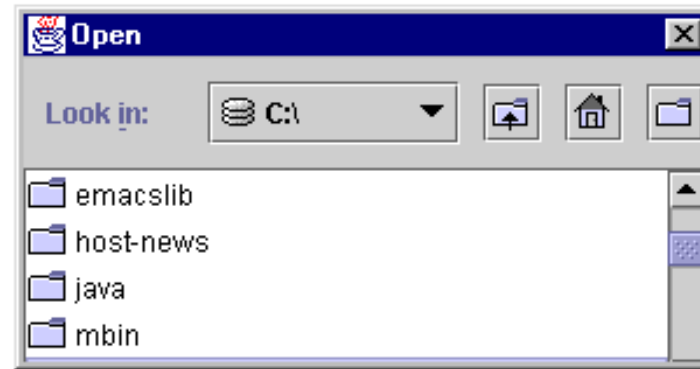
# Swing Components

## Interactive Displays of Highly Formatted Information

### Interactive Displays of Highly Formatted Information



[Color chooser](#)



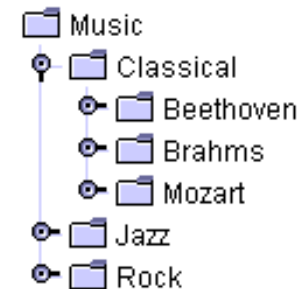
[File chooser](#)

First Name	Last Name	Favorite Food
Jeff	Dinkins	
Ewan	Dinkins	
Amy	Fowler	
Hania	Gajewska	
David	Geary	

[Table](#)



[Text](#)



[Tree](#)



# Swing Components

## Interactive Displays of Highly Formatted Information

- Define standard interfaces for frequently needed tasks

... go to [java.sun.com](http://java.sun.com) for further information ...

# Layout Management

How to glue it all together:

## The Layout Management

# Layout Management

- The process of determining the size and position of components
- A layout manager is an object that performs layout management for the components within the container.
- Layout managers have the final say on the size and position of components added to a container
- Using the add method to put a component in a container, you must **ALWAYS** take the container's layout manager into account

# Layout Management

... and finally, the layout manager preserves the world from home made layout-design !

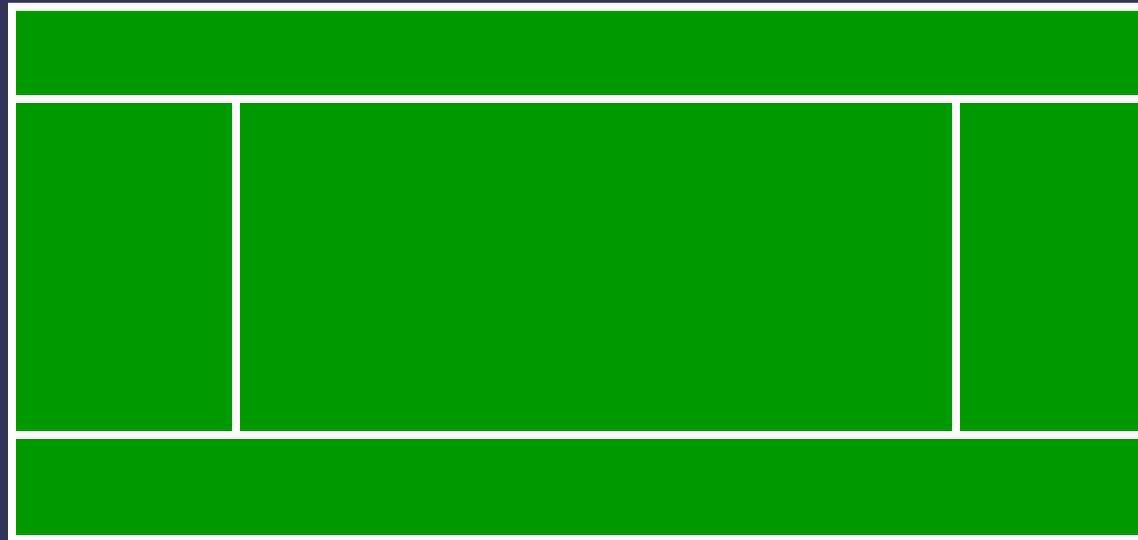
# Layout Management

Java supplies five commonly used layout managers:

1. BorderLayout
2. BoxLayout
3. FlowLayout
4. GridBagLayout
5. GridLayout

# Layouts

## BorderLayout



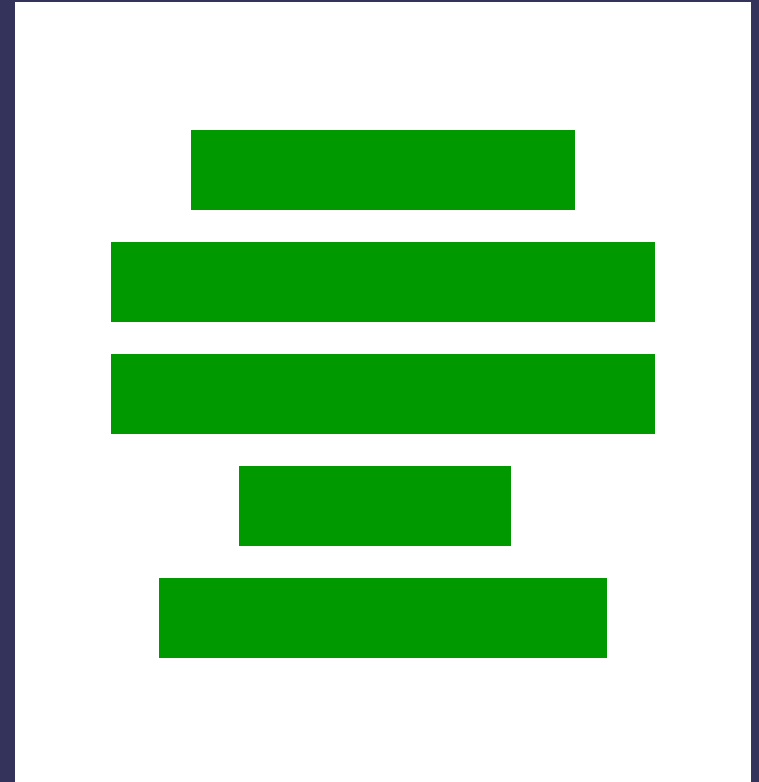
Position must be specified, e.g. `add ("North", myComponent)`

# Layouts

## BoxLayout

The BoxLayout class puts components in a single row or column.

It respects the components' requested maximum sizes.



# Layouts

## FlowLayout

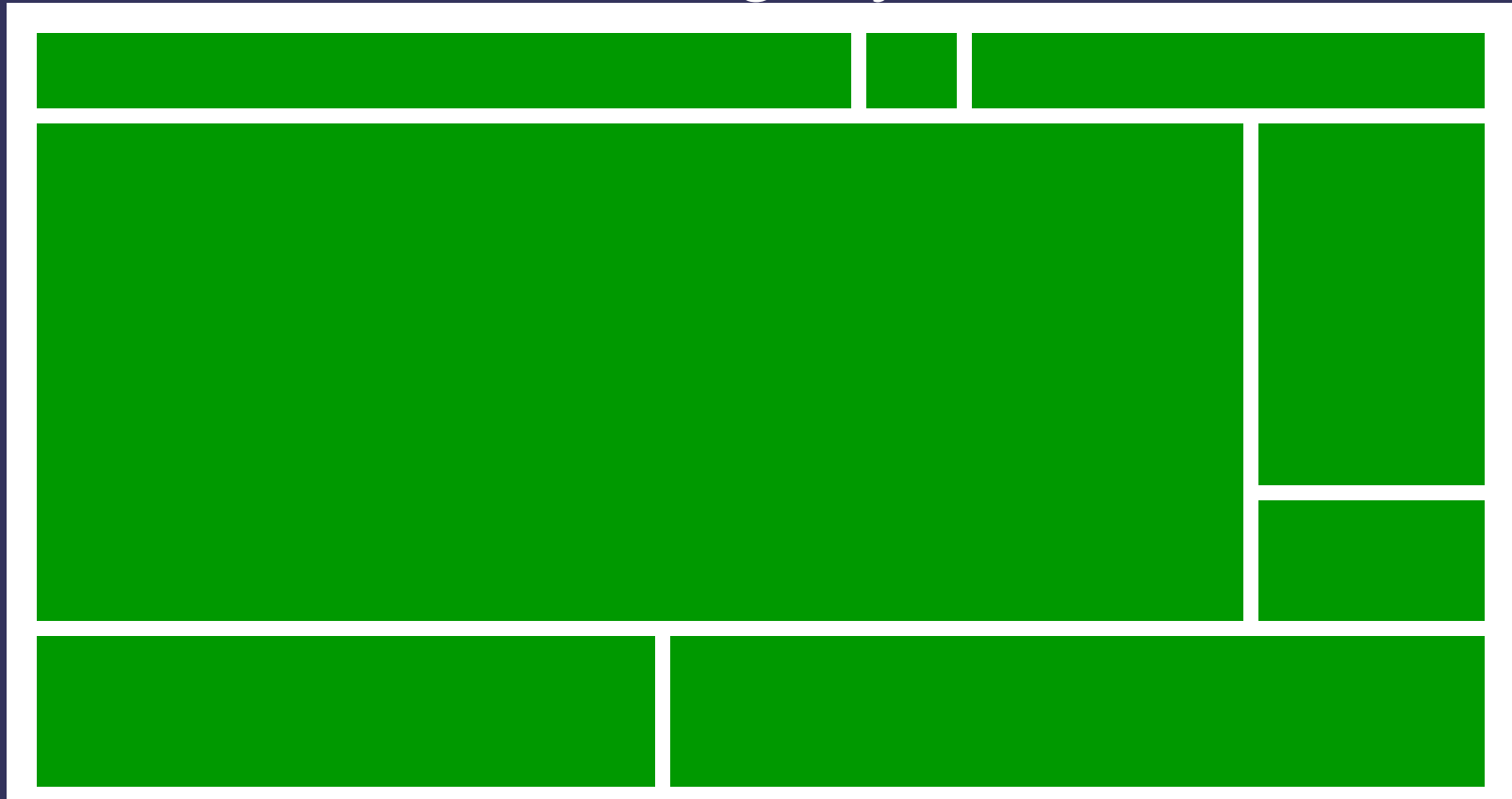


FlowLayout is the default layout manager for every JPanel. It simply lays out components from left to right, starting new rows if necessary



# Layouts

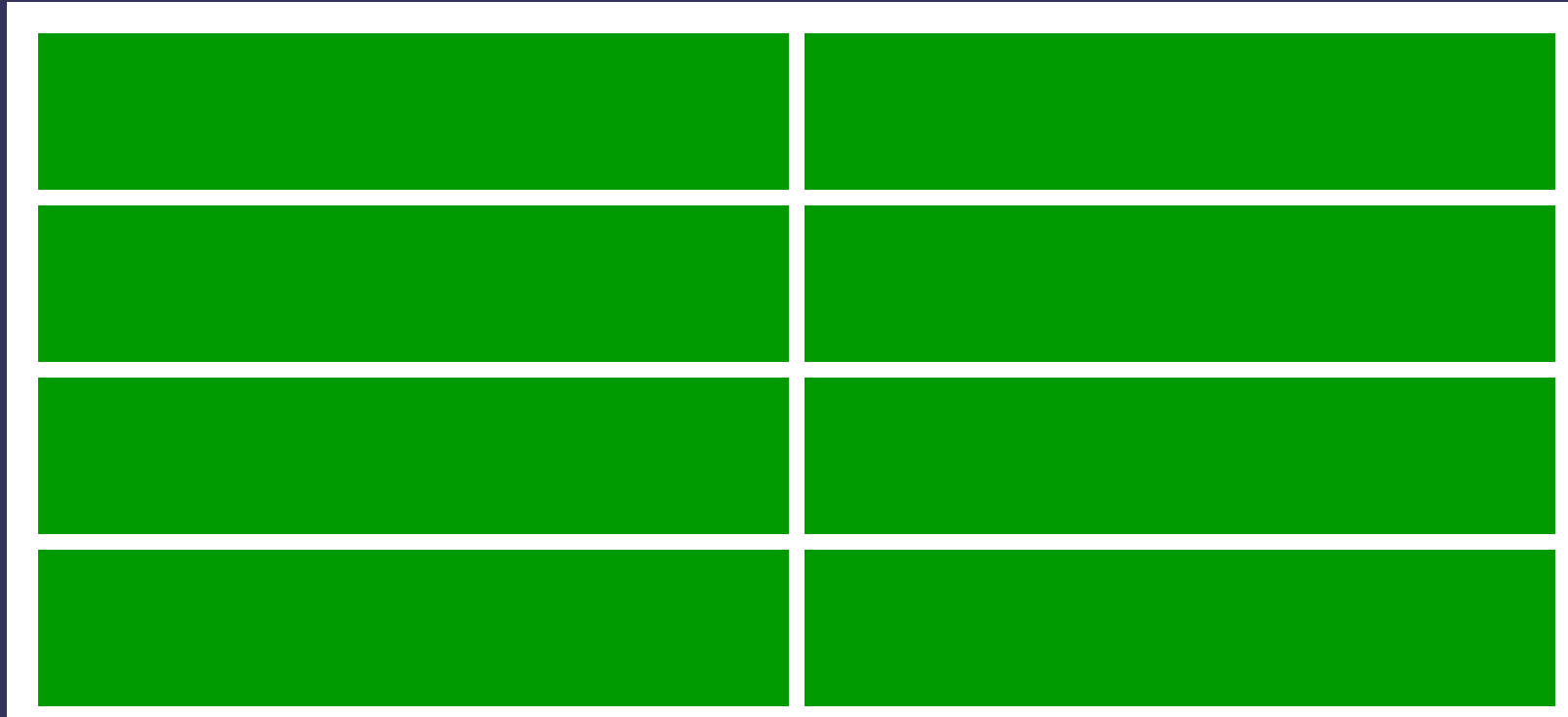
## GridBagLayout



GridBagLayout is the most sophisticated, flexible layout manager the Java platform provides. If you really want to use it, go to [java.sun.com](http://java.sun.com) ...

# Layouts

## GridLayout



GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns .

# Using Components

## Examples:

- Using a JButton
- Using a JSlider
- Using a JCheckBox

# Using a JButton

## Some Constructors:

<b>JButton()</b>	Creates a button with no text or icon
<b>JButton(Icon icon)</b>	Creates a button with an icon
<b>JButton(String text)</b>	Creates a button with text
<b>JButton(String text, Icon icon)</b>	Creates a button with initial text and an icon

# Using a JButton

## Some Methods:

<b>addActionListener( ActionListener a)</b>	Registers ActionListener to JButton Inherited from AbstractButton
<b>setFont(Font font)</b>	Specifies Font (Type, Style, Size) Inherited from JComponent
<b>setBackground( Color color)</b>	Sets background color Inherited from JComponent
<b>setActionCommand (String text)</b>	Used to specify button if listener is registered to multiple buttons (see ActionEvent.getActionCommand())

# Using a JSlider



## Some Constructors:

<code>JSlider()</code>	Creates a horizontal slider with the range 0 to 100 and an initial value of 50
<code>JSlider( int min, int max, int value)</code>	Creates a horizontal slider using the specified min, max and value.
<code>JSlider( int orientation int min, int max, int value)</code>	Creates a slider with the specified orientation and the specified minimum, maximum, and initial values.

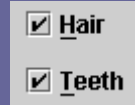
# Using a JSlider



## Some Methods:

<code>addChangeListener (ChangeListener cl)</code>	Registers ChangeListener to slider
<code>int getValue()</code>	Returns the slider's value
<code>setValue(int value)</code>	Sets the slider's value

# Using a JCheckBox



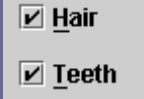
## Some Constructors:

JCheckBox()	Creates an initially unselected check box button with no text, no icon.
JCheckBox( String text)	Creates an initially unselected check box with text.
<b>JCheckBox</b> ( String text, Icon icon, boolean selecte	Creates a check box with text and icon, and specifies whether or not it is initially selected.

d)



# Using a JCheckBox



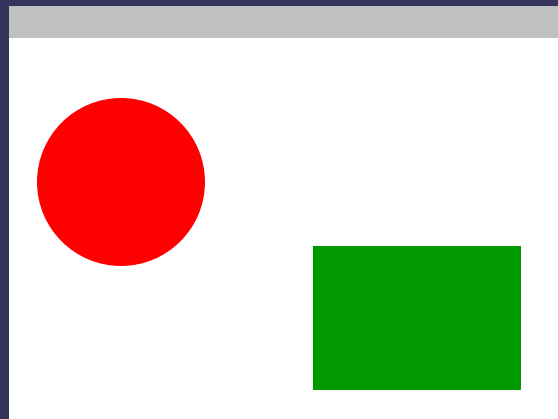
## Some Methods:

<code>addItemListener (ItemListener il)</code>	Registers ItemListener to checkbox Inherited from AbstractButton
<code>setSelected( boolean select)</code>	Sets the state of checkbox Inherited from AbstractButton
<b>boolean getSelected()</b>	Gets the state of checkbox. calling method often saves from registering to the checkbox !

# Custom Painting

creating your own graphics:

## Custom Painting



# Custom Painting

Decide which superclass to use, for example:

- JPanel: Generating and displaying graphs in top of a blank or transparent background
- JLabel: Painting on top of an image
- JButton: custom button
- ...

Every class derived from JComponent can be used  
for custom drawing !

(Recommended: JPanel)

# Custom Painting

## The Graphics Object

- provides both a context for painting and methods for performing the painting.
- Example of methods
  - drawImage
  - drawString
  - drawRect
  - fillRect
  - setColor
  - ...
- passed as argument to the paintComponent - method

# Custom Painting

## The paintComponent method

- Method of class JComponent
- Inherited to all subclasses, e.g. JPanel, JButton,...
- The place where all custom painting belongs !
- Invoked by the event-scheduler or by the repaint() - method

# Using Swing

(demo program)

# At last...

This was a BRIEF overview and introduction to SWING. SWING has MUCH more to offer, see

- <http://java.sun.com/docs/books/tutorial/uiswing/>
  - <http://java.sun.com/j2se/1.4.1/docs/api/>

- class javax.swing.**JComponent** (implements java.io.[Serializable](#))
  - class javax.swing.**AbstractButton** (implements java.awt.[ItemSelectable](#), javax.swing.[SwingConstants](#))
    - class javax.swing.**JButton** (implements javax.accessibility.[Accessible](#))
      - class javax.swing.plaf.basic.**BasicArrowButton** (implements javax.swing.[SwingConstants](#))
        - class javax.swing.plaf.metal.**MetalScrollbarButton**
        - class javax.swing.plaf.metal.**MetalComboBoxButton**
    - class javax.swing.**JMenuItem** (implements javax.accessibility.[Accessible](#), javax.swing.[MenuItem](#))
      - class javax.swing.**JCheckBoxMenuItem** (implements javax.accessibility.[Accessible](#), javax.swing.[SwingConstants](#))
      - class javax.swing.**JMenu** (implements javax.accessibility.[Accessible](#), javax.swing.[MenuItem](#))
      - class javax.swing.**JRadioButtonMenuItem** (implements javax.accessibility.[Accessible](#))
    - class javax.swing.**JToggleButton** (implements javax.accessibility.[Accessible](#))
      - class javax.swing.**JCheckBox** (implements javax.accessibility.[Accessible](#))
      - class javax.swing.**JRadioButton** (implements javax.accessibility.[Accessible](#))
  - class javax.swing.plaf.basic.**BasicInternalFrameTitlePane**
    - class javax.swing.plaf.metal.**MetalInternalFrameTitlePane**
  - class javax.swing.**Box** (implements javax.accessibility.[Accessible](#))
  - class javax.swing.**Box.Filler** (implements javax.accessibility.[Accessible](#))
  - class javax.swing.**JColorChooser** (implements javax.accessibility.[Accessible](#))
  - class javax.swing.**JComboBox** (implements javax.accessibility.[Accessible](#), java.awt.event.[ActionListener](#), java.awt.[ItemSelectable](#), javax.swing.event.[ListDataListener](#))
  - class javax.swing.**JFileChooser** (implements javax.accessibility.[Accessible](#))
  - class javax.swing.**JInternalFrame** (implements javax.accessibility.[Accessible](#), javax.swing.[RootPaneContainer](#), javax.swing.[WindowConstants](#))
  - class javax.swing.**JInternalFrame.JDesktopIcon** (implements javax.accessibility.[Accessible](#))
  - class javax.swing.**JLabel** (implements javax.accessibility.[Accessible](#), javax.swing.[SwingConstants](#))
    - class javax.swing.plaf.basic.**BasicComboBoxRenderer** (implements javax.swing.[ListCellRenderer](#), java.io.[Serializable](#))
      - class javax.swing.plaf.basic.**BasicComboBoxRenderer.UIResource** (implements javax.swing.plaf.[UIResource](#))
    - class javax.swing.**DefaultListCellRenderer** (implements javax.swing.[ListCellRenderer](#), java.io.[Serializable](#))
      - class javax.swing.**DefaultListCellRenderer.UIResource** (implements javax.swing.plaf.[UIResource](#))
      - class javax.swing.plaf.metal.**MetalFileChooserUI.FileRenderer**
      - class javax.swing.plaf.metal.**MetalFileChooserUI.FilterComboBoxRenderer**
  - class javax.swing.table.**DefaultTableCellRenderer** (implements java.io.[Serializable](#), javax.swing.table.[TableCellRenderer](#))

# Part II

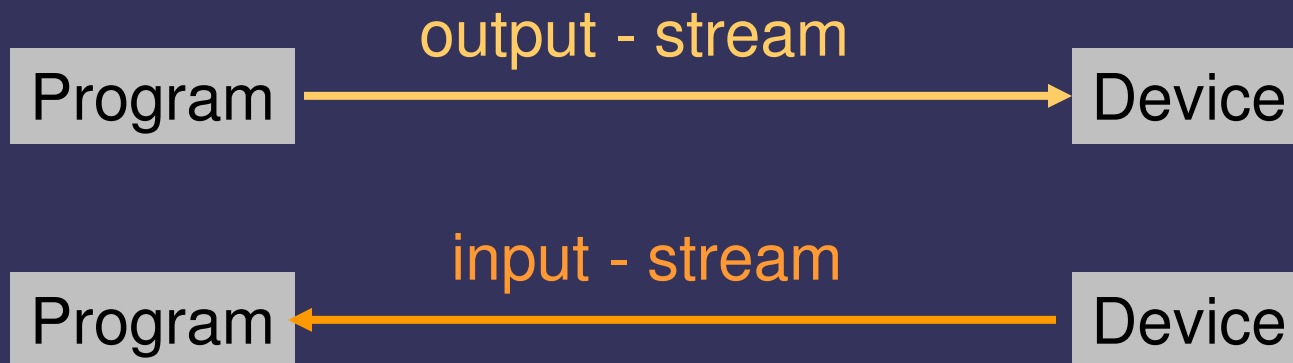
JAVA I/O:

## Streams and Files



# I/O

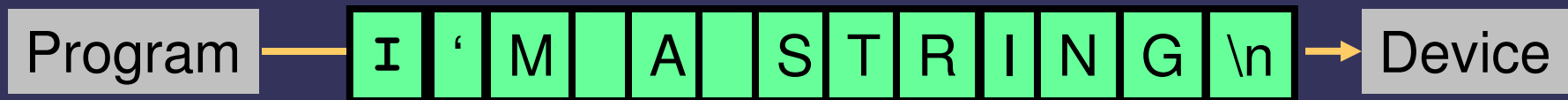
- Usual Purpose: storing data to 'nonvolatile' devices, e.g. harddisk
- Classes provided by package java.io
- Data is transferred to devices by 'streams'



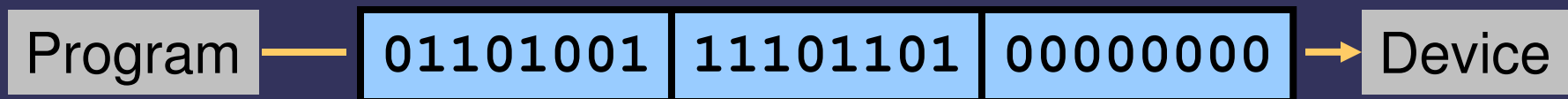
# Streams

JAVA distinguishes between 2 types of streams:

- Text – streams, containing ‘characters’



- Binary Streams, containing 8 – bit information



# Streams

Streams in JAVA are Objects, of course !

Having

- 2 types of streams (text / binary) and
- 2 directions (input / output)

results in 4 base-classes dealing with I/O:

1. Reader: text-input
2. Writer: text-output
3. InputStream: byte-input
4. OutputStream: byte-output

# Streams

InputStream

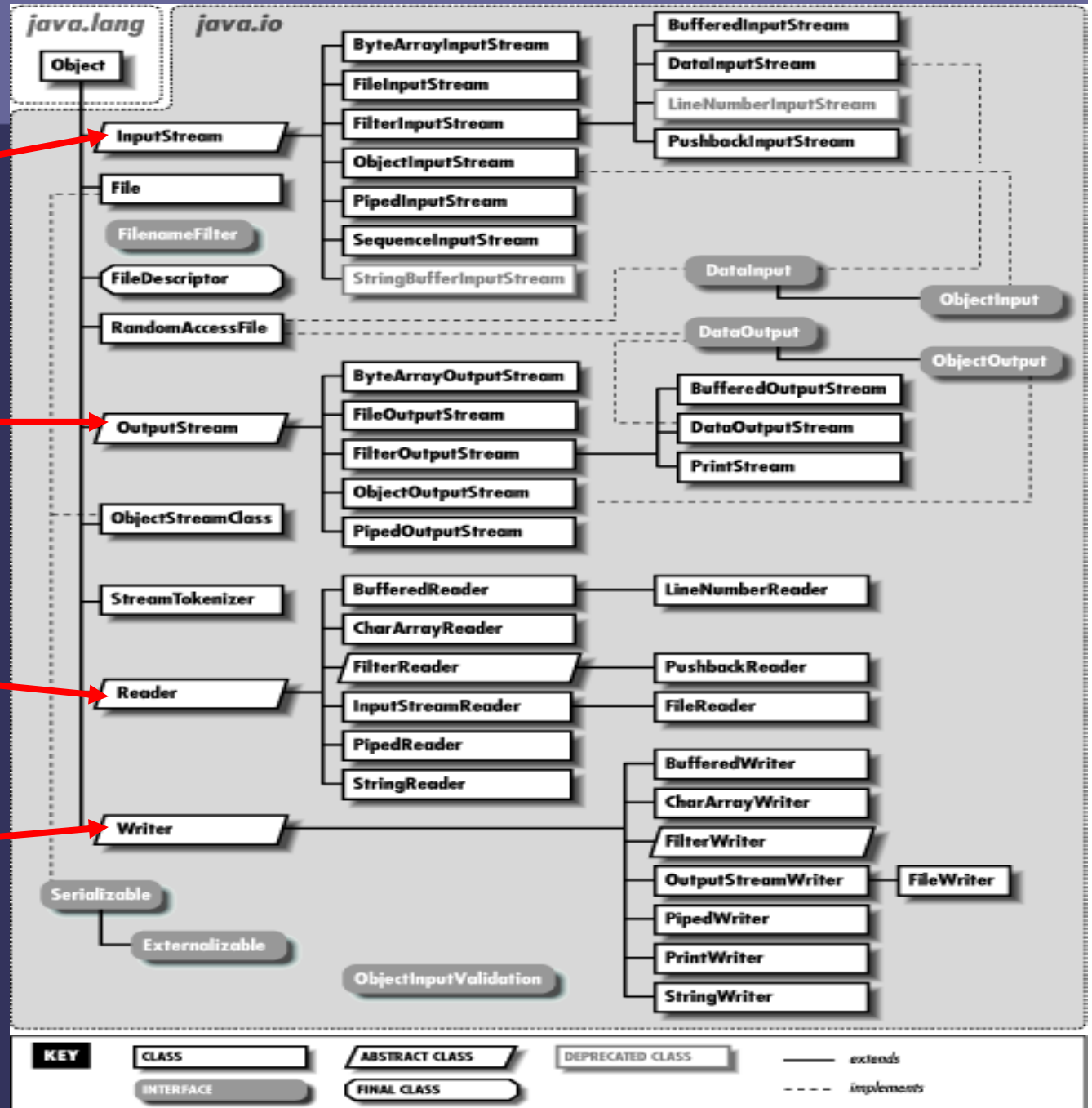
OutputStream

binary

Reader

Writer

text



# Streams

- `InputStream`, `OutputStream`, `Reader`, `Writer` are abstract classes
- Subclasses can be classified by 2 different characteristics of sources / destinations:
  - For final device (data sink stream)  
purpose: serve as the source/destination of the stream  
(these streams 'really' write or read !)
  - for intermediate process (processing stream)  
Purpose: alters or manages information in the stream  
(these streams are 'luxury' additions, offering methods for convenient or more efficient stream-handling)

# I/O: General Scheme

## In General:

Reading (writing):

- open an input (output) stream
- while there is more information
  - read(write) next data from the stream
- close the stream.

## In JAVA:

- Create a stream object and associate it with a disk-file
  - Give the stream object the desired functionality
- while there is more information
  - read(write) next data from(to) the stream
- close the stream.

# Example 1

## Writing a textfile:

```
import java.io.*;

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

            FileWriter out = new FileWriter("test.txt");
            BufferedWriter b = new BufferedWriter(out);
            PrintWriter p = new PrintWriter(b);

            p.println("I'm a sentence in a text-file");

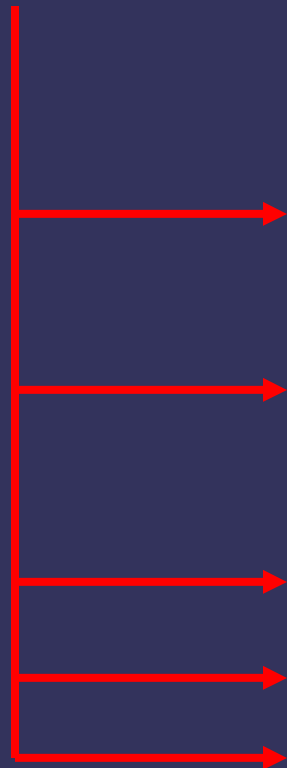
            p.close();
        } catch(Exception e){}
    }
}
```

- Create a stream object and associate it with a disk-file
- Give the stream object the desired functionality
- write data to the stream
- close the stream.

# Writing Textfiles

Class: FileWriter

Frequently used methods:



Method Summary	
abstract void	<b>close</b> () Close the stream, flushing it first.
abstract void	<b>flush</b> () Flush the stream.
void	<b>write</b> (char[] cbuf) Write an array of characters.
abstract void	<b>write</b> (char[] cbuf, int off, int len) Write a portion of an array of characters.
void	<b>write</b> (int c) Write a single character.
void	<b>write</b> (String str) Write a string.
void	<b>write</b> (String str, int off, int len) Write a portion of a string.



# Writing Textfiles

## Using FileWriter

- is not very convenient (only String-output possible)
- Is not efficient (every character is written in a single step, invoking a huge overhead)

Better: wrap FileWriter with processing streams

- BufferedWriter
- PrintWriter

# Wrapping Textfiles

## BufferedWriter:

- Buffers output of FileWriter, i.e. multiple characters are processed together, enhancing efficiency

## PrintWriter

- provides methods for convenient handling, e.g. `println()`

( remark: the `System.out.println()` – method is a method of the PrintWriter-instance `System.out` ! )

# Wrapping a Writer

A typical codesegment for opening a convenient, efficient textfile:

```
FileWriter out = new FileWriter("test.txt");  
BufferedWriter b = new BufferedWriter(out);  
PrintWriter p = new PrintWriter(b);
```

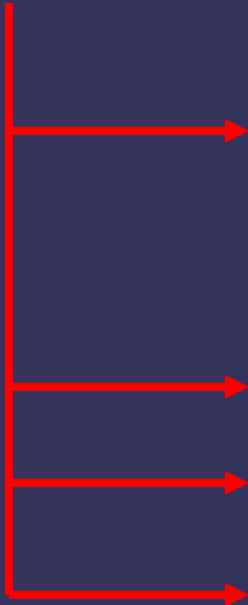
Or with anonymous ('unnamed') objects:

```
PrintWriter p = new PrintWriter(  
    new BufferedWriter(  
        new FileWriter("test.txt")));
```

# Reading Textfiles

## Class: ReadText

### Frequently used Methods:



Method Summary	
abstract void	<b><u>close</u></b> () Close the stream.
void	<b><u>mark</u></b> (int readAheadLimit) Mark the present position in the stream.
boolean	<b><u>markSupported</u></b> () Tell whether this stream supports the mark() operation.
int	<b><u>read</u></b> () Read a single character.
int	<b><u>read</u></b> (char[] cbuf) Read characters into an array.
abstract int	<b><u>read</u></b> (char[] cbuf, int off, int len) Read characters into a portion of an array.
boolean	<b><u>ready</u></b> () Tell whether this stream is ready to be read.
void	<b><u>reset</u></b> () Reset the stream.
long	<b><u>skip</u></b> (long n) Skip characters.

(The other methods are used for positioning, we don't cover that here)

# Wrapping a Reader

Again:

Using `FileReader` is not very efficient. Better wrap it with `BufferedReader`:

```
BufferedReader br =  
    new BufferedReader(  
        new FileReader("name"));
```

Remark: `BufferedReader` contains the method `readLine()`, which is convenient for reading textfiles

# EOF Detection

## Detecting the end of a file (EOF):

- Usually amount of data to be read is not known
- Reading methods return 'impossible' value if end of file is reached
- Example:
  - `FileReader.read` returns -1
  - `BufferedReader.readLine()` returns 'null'
- Typical code for EOF detection:

```
while ((c = myReader.read()) != -1){ // read and check c
    ...do something with c
}
```

# Example 2: Copying a Textfile

```
import java.io.*;
public class IOTest
{
    public static void main(String[] args)
    {
        try{
            BufferedReader myInput = new BufferedReader(new
                FileReader("IOTest.java"));
            BufferedWriter myOutput = new BufferedWriter(new
                FileWriter("Test.txt"));

            int c;
            while ((c=myInput.read()) != -1)
                myOutput.write(c);

            myInput.close();
            myOutput.close();
        }catch(IOException e){}
    }
}
```

# Binary Files

- Stores binary images of information identical to the binary images stored in main memory
- Binary files are more efficient in terms of processing time and space utilization
- drawback: not 'human readable', i.e. you can't use a texteditor (or any standard-tool) to read and understand binary files



# Binary Files

Example: writing of the integer '42'

- TextFile: '4' '2' (internally translated to 2 16-bit representations of the characters '4' and '2')
- Binary-File: 00101010, one byte (= 42 decimal)

# Writing Binary Files

Class: `FileOutputStream`

... see `FileWriter`

The difference:

No difference in usage, only in output format

# Reading Binary Files

Class: `FileInputStream`

... see `FileReader`

The difference:

No difference in usage, only in output format

# Binary vs. TextFiles

	pro	con
Binary	Efficient in terms of time and space	Preinformation about data needed to understand content
Text	Human readable, contains redundant information	Not efficient

# Binary vs. TextFiles

## When use Text- / BinaryFiles ?

- **ALWAYS** use TextFiles for final results if there's no imperative reason to favor efficiency against readability.  
Example: SIP - Standard
- Binary Files might be used for non-final interchange between programs
- Binary Files are always used for large amount of data (images, videos etc.), but there's always an *exact* definition of the meaning of the bytestream  
Example: JPG, MP3, BMP

# ... outlook

Next time:

- Exception handling
- Other types than Files (Console, Serializing)