

Object-Oriented Programming III

2017/2018 Semester 1

Week 9 & 10

- More about I/O streams
- Introduction to GUI programming

INPUT/OUTPUT

Streams

- ▶ Input/Output (I/O) is done in Java mainly using *I/O streams*
 - ▶ You have already used I/O streams before, e.g., `System.out` and `System.in`, and you've seen streams in the context of files and networking (where I/O streams are attached to sockets)
 - ▶ Today, some revision and further details...
-
- ▶ Remark: Don't confuse I/O streams with so-called *Java 8 Streams* (--> another lecture)

INPUT/OUTPUT

Streams

- ▶ In Java, special stream(-like) classes for *character streams* have been introduced. On top of the hierarchy of these new classes are
 - ▶ `java.io.Reader`
 - ▶ `java.io.Writer`
- ▶ These classes form their own class hierarchy outside older stream classes
- ▶ For all character in-/output, they are preferred over any other stream classes

INPUT/OUTPUT

Streams

- ▶ **Examples for Readers/Writers (also see Seamus' lectures):**
 - ▶ `java.io.FileWriter`
 - ▶ Similar to `FileOutputStream`, but for files which consist of characters only
 - ▶ `java.io.FileReader`
 - ▶ Similar to `FileInputStream`, but for characters
 - ▶ `java.io.PrintWriter`
 - ▶ For writing characters and strings. Preferred over `PrintStream`

E.g., `PrintWriter pw = new PrintWriter(socket.getOutputStream());`
`pw.print("Hi there!");`
- ▶ Specifically for reading text files token-by-token or line-by-line, also see class `Scanner` later in the lecture (and the example code from previous lecture on networking)

INPUT/OUTPUT

Streams

- ▶ *Byte streams* are the most basic kinds of streams
- ▶ They are low-level streams for the input/output of raw binary data in form of sequences of single bytes
- ▶ Examples:
 - ▶ `java.io.FileOutputStream`
 - ▶ `java.io.FileInputStream`
 - ▶ `java.io.ByteArrayOutputStream`
 - ▶ `java.io.ByteArrayInputStream`
- ▶ Use cases, e.g., writing/reading images or copying arbitrary files.

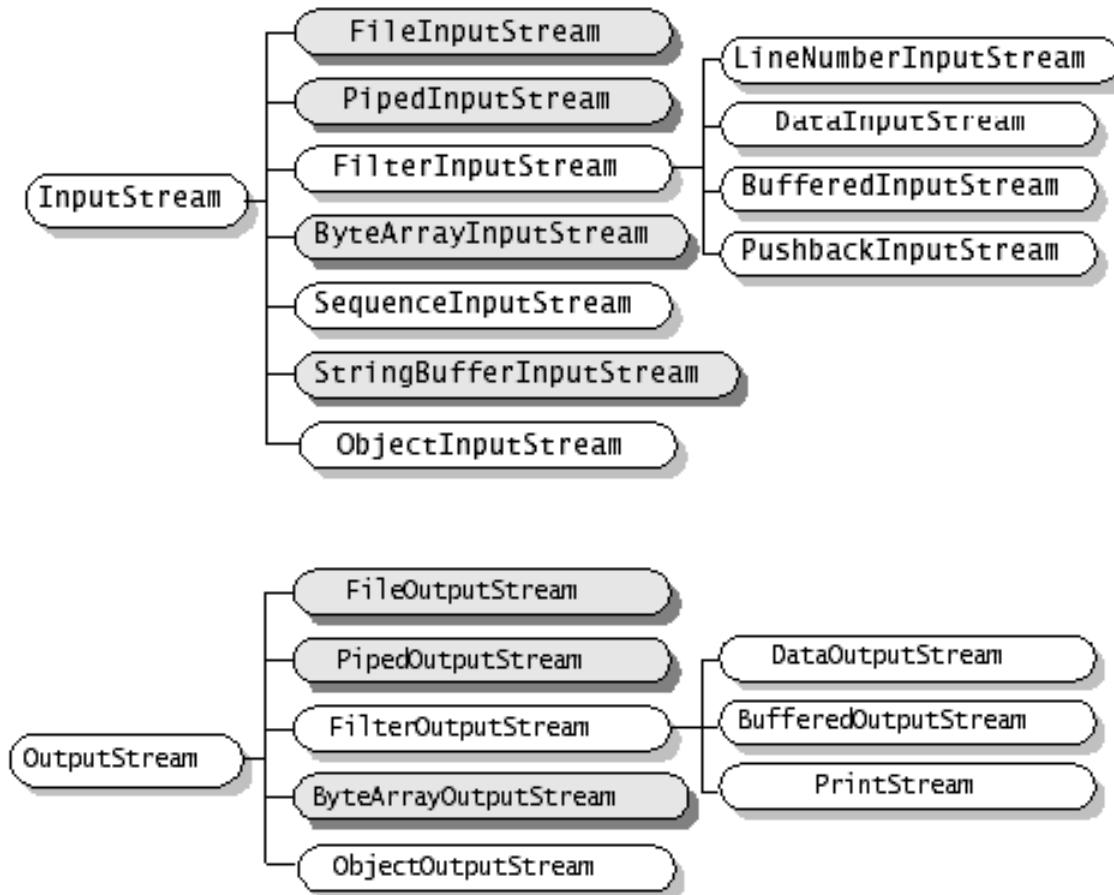
INPUT/OUTPUT

Streams

- ▶ Don't use byte streams directly if you want to write/read characters, numbers or higher-order items (entire strings or other Java objects).
- ▶ Always check whether there is a “higher” kind of stream class available for your application before using a byte stream directly (e.g., use one of the `Writer`-streams instead to write strings)
- ▶ But “higher” streams internally make use of underlying byte streams

INPUT/OUTPUT

Streams



INPUT/OUTPUT

Streams

- ▶ An important operation on streams is the "wrapping" of a higher-level stream around some underlying lower-level stream. Such a higher-level stream "wrapped around" another stream is called *wrapper stream* or *filter stream*.
- ▶ Wrapping is done by passing a stream object as constructor argument to the constructor of the wrapper stream class.
- ▶ Many of these "wrappers" are derived from one of the following classes:
 - ▶ `java.io.FilterOutputStream`
 - ▶ `java.io.FilterInputStream`
- ▶ An example for wrapper streams are *buffers* (see older lectures)
- ▶ Remark: Don't confuse this concept with *wrapper classes*, such as `Integer` as a wrapper class for primitive type `int`.

INPUT/OUTPUT

Streams

- ▶ Some types of streams need to be used as wrapper streams, they cannot be used standalone (e.g., `java.io.BufferedReader`)
- ▶ Wrapping can be nested; here an example for a "double-wrapping":

```
BufferedReader bufferReader =  
    new BufferedReader(  
        new InputStreamReader(System.in));
```

- ▶ Here, the byte stream `System.in` is wrapped inside a `InputStreamReader` stream (to allow reading of characters) which is wrapped inside a `BufferedReader` stream (for caching, in order to improve performance)

INPUT/OUTPUT

Streams

▶ Some other useful filter stream types (see Java API):

- ▶ `javax.crypto.CipherOutputStream`

 - ▶ Encrypted output

- ▶ `javax.crypto.CipherInputStream`

 - ▶ Decrypted input

- ▶ `java.util.zip.ZipOutputStream`

 - ▶ Writes compressed data

- ▶ `java.util.zip.ZipInputStream`

 - ▶ Un-compresses data

▶ Example:

```
ZipOutputStream out =  
    new ZipOutputStream(new FileOutputStream(fileName));
```

INPUT/OUTPUT

Streams

- ▶ *Data streams* are wrapper streams (wrapped around byte streams) which allow to write/read primitive data type values (such as `int`, `boolean`, `float`, `double`...).
- ▶ **Examples**
 - ▶ `java.io.DataOutputStream`
 - ▶ `java.io.DataInputStream`
- ▶ Use case: writing/reading primitive data (such as numbers) in form of byte sequences
- ▶ Portable - written data can be read by another Java program on another machine
- ▶ Data streams can also be used for writing/reading arbitrary strings, but this is not recommended any more. Use classes descending from classes `Reader/Writer` instead for this (e.g., `FileWriter`, `InputStreamReader`)

INPUT/OUTPUT

Streams

- ▶ The following example wraps a data stream around a file output stream, in order to write a few numbers and a boolean value into a file...
- ▶ Afterwards (slide after next slide) we show how to read the written values back from file data.dat
- ▶ Observe that with data output streams, values are not written in human-readable (textual) form into the file.
- ▶ Data is written sequentially into the file, in the order determined by the order of the stream writing instructions in your program
- ▶ Not suitable to write arbitrary objects as a whole.
See *object serialization* for streams which can write/read arbitrary objects to/from I/O streams.

INPUT/OUTPUT

Streams

...

```
try {
```

```
    DataOutputStream outputStream =
```

```
        new DataOutputStream(
```

```
            new FileOutputStream("data.dat"));
```

name of the file



Low-level stream for writing raw bytes into a file



```
    outputStream.writeInt(12); // writes number 12 into the file
```

```
    outputStream.writeDouble(534.0276);
```

```
    outputStream.writeBoolean(false);
```

```
    outputStream.close();
```

```
} catch (IOException e) {
```

```
    System.out.println(e.getMessage());
```

```
}
```

INPUT/OUTPUT

Streams

```
try {
    DataInputStream inputStream = new DataInputStream(
        new FileInputStream("data.dat"));

    int value1 = inputStream.readInt();
    double value2 = inputStream.readDouble();
    boolean value3 = inputStream.readBoolean();
    inputStream.close();

    System.out.println("value1 = " + value1 + ", value2 = "
        + value2 + ", value3 = " + value3);

} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

INPUT/OUTPUT

Tokenizing streams

- ▶ Often it is required to break the data delivered by a character input stream into *tokens*.
- ▶ A *token* is a delimited sub-string encountered during parsing of texts, e.g., a word or a number (both in form of character sequences).
To "*tokenize*" means to split some given character sequence (e.g., a string or a stream) into tokens.
- ▶ Tokens are separated by *delimiters* (whitespace, commas or user-defined characters).
- ▶ Which delimiter to use is decided by the programmer (default typically whitespace, e.g., ' ', '\n', '\t'...)

INPUT/OUTPUT

Tokenizing streams

- ▶ E.g., here's a string consisting of four tokens

“123.45,222.99,.743,3.141592653”

Here the tokens are substrings which represent numbers, and the delimiter is ','

- ▶ Another example:

“The fox jumps over the wall”

The tokens of this string are the words "The", "fox", "jumps", "over", "the", and "wall".

Delimiter: whitespace, which is the default delimiter used by class `Scanner`...


INPUT/OUTPUT

Tokenizing streams

```
import java.io.*;
import java.util.Scanner;

public void tokenizeFile() throws IOException {
    Scanner s = null;
    try {
        s = new Scanner(
            new BufferedReader(new FileReader("text.txt")));
        s.useDelimiter("\\s*,\\s*"); // use commas as delimiters.
                                   // (default delimiter is
                                   // whitespace!)
        while (s.hasNext()) { // any more tokens in stream?
            System.out.println(s.next()); // print next token
        }
    } finally {
        if (s != null) s.close();
    }
}
```

*we use Scanner like a wrapper stream,
although it is actually not a stream class*



INPUT/OUTPUT

Tokenizing streams

- ▶ Class `Scanner` isn't actually a stream, but it can be used as a kind of wrapper stream for reading tokens from other kinds of streams. See lecture about networking for another code example.

GUI

Swing

- ▶ Graphical User Interfaces (GUIs)...
- ▶ We will make use of *Java Swing*, which is a part of the *Java Foundation Classes* (JFC), which are a part of the regular Java API.
- ▶ Main relevant package `javax.swing`, plus a few others
- ▶ We will also use this topic to take a closer look at a certain important type of inner classes: *anonymous classes*

GUI

Swing

- ▶ Older approach: *Abstract Window Toolkit* (AWT). Rarely used directly.
- ▶ Alternatives to Swing in the Java-world:
 - ▶ *JavaFX* is newer and "cleaner" than Swing, but future of JavaFX is unclear whereas Swing is still popular in industry
 - ▶ *HTML 5.0* (with JavaScript): this is the recommended approach for creating basic GUIs in a client/server environment, where the server uses Java. Requires a web browser.
 - ▶ *The Standard Widget Toolkit* (SWT). SWT is fast and creates GUIs with a more native "look and feel" than Swing, but not as widely used as Swing and less configurable. Not very popular, used mainly for the Eclipse GUI.
- ▶ Lot of similarity between all existing GUI frameworks (e.g., event handling) - if you know one, it's easy to learn others

GUI

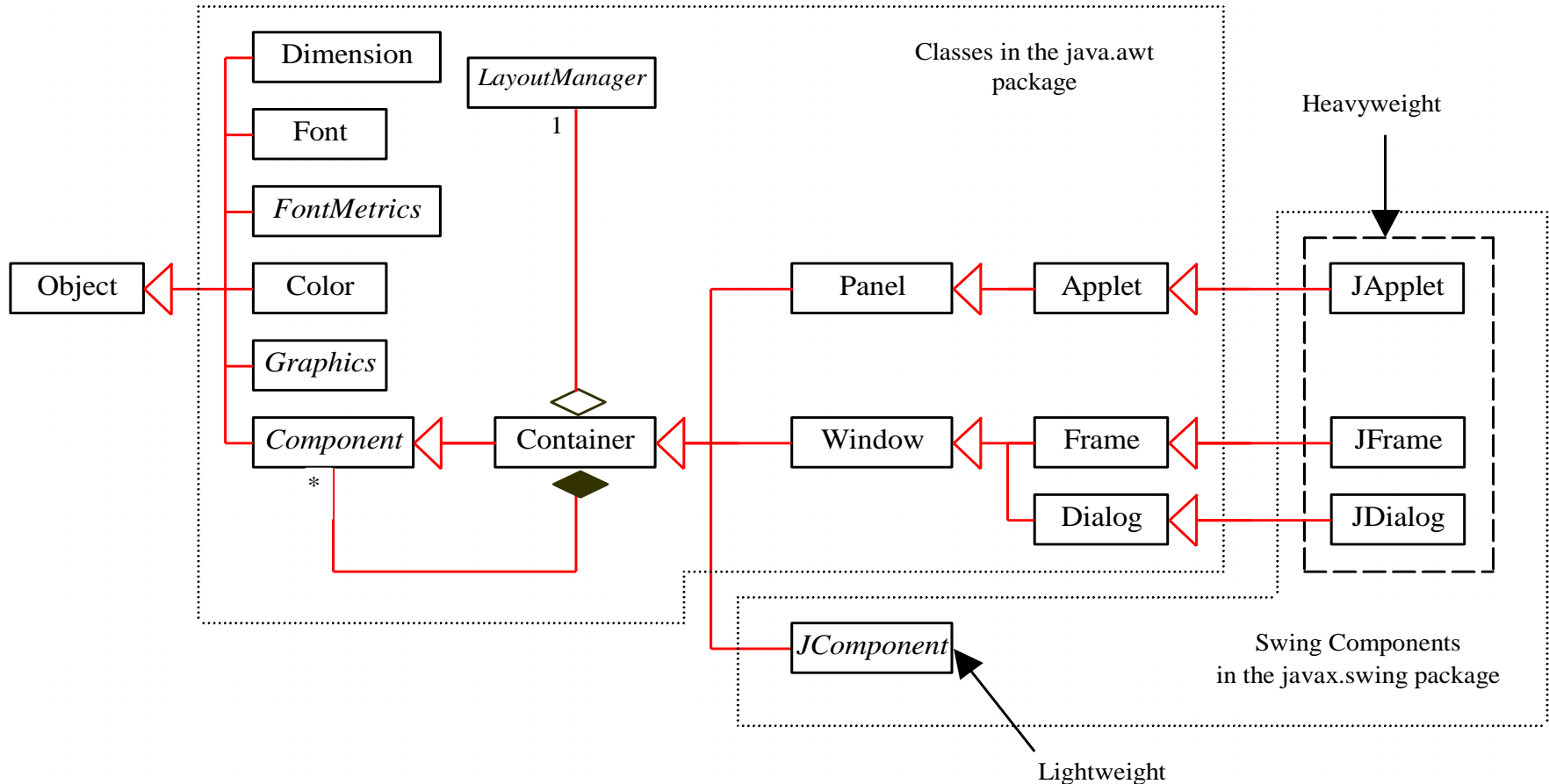
AWT

- ▶ Swing makes heavy use of an older part Java, namely the Abstract Window Toolkit (AWT).
- ▶ The AWT can be used to create GUIs without Swing, but is now mainly hidden from the "normal programmer".
- ▶ However, sometimes you will have to use AWT classes even as a Swing programmer
- ▶ Just to give you an overview of AWT...

No worries, you don't have to remember details of the following two diagrams!

GUI

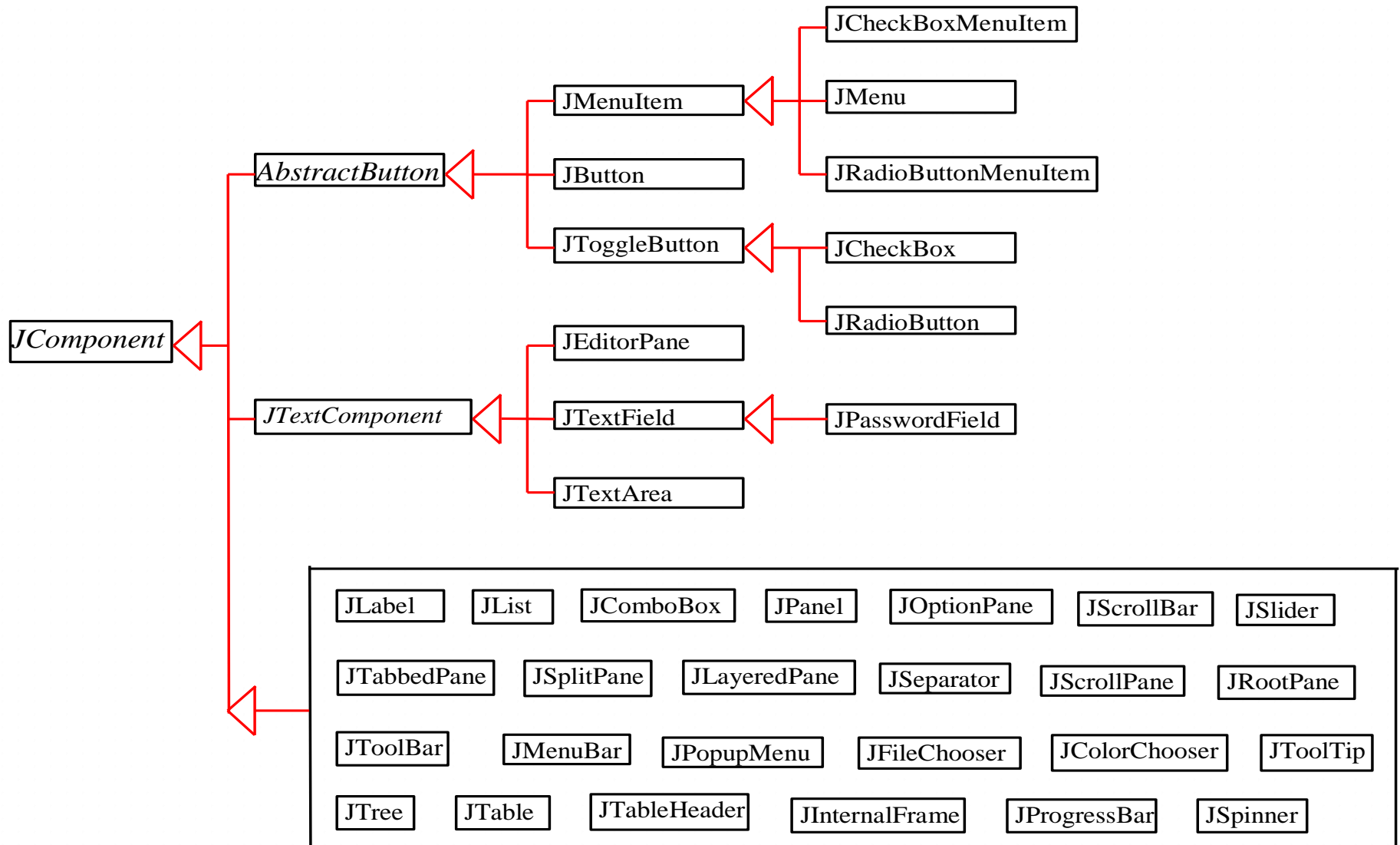
A few important Swing and AWT classes (in UML)



Some of the figures and examples in this lecture from D. Liang: Introduction to Java Programming, Prentice Hall

GUI

Even more Swing classes



GUI

- ▶ The objects which the user sees on the screen (buttons, text fields, lists, windows, etc.) are called *components*.
- ▶ A component which can contain other components on the screen is called a (GUI) *container*.
- ▶ A component might or might not be visible - there are not only visible components, but also components set to state "invisible", or components which don't have any graphical representation, or components shadowed by others.
- ▶ **Don't confuse these with *collections*** (→ Java Collections Framework, e.g. `ArrayList<...>`), which are sometimes called "container classes". However, internally, some GUI containers are implemented using these collection classes, and collections are often a good choice for the *data model* of GUI components/containers.

GUI

- ▶ "Component" and "container" are standard terminology - however, the Java class inheritance hierarchy does not precisely reflect this terminology:

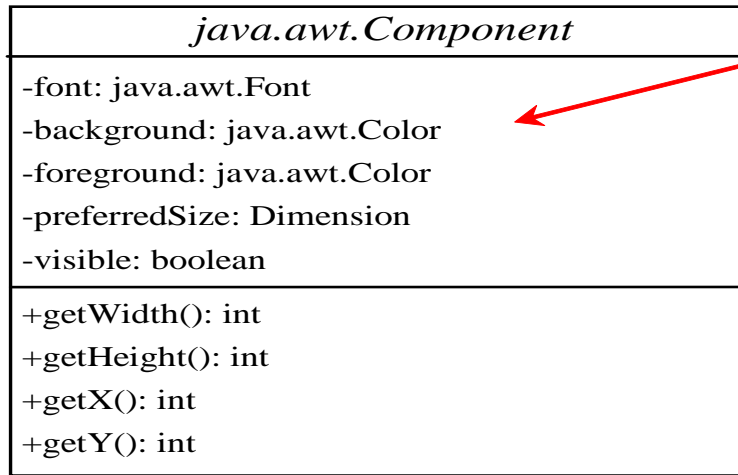
The abstract class `JComponent` is derived from (i.e., is a subclass of) the AWT's container class `Container`, which is derived from AWT's `Component` class!

However, Swing's windows classes (e.g., `JFrame`) are not derived from `JComponent` but still (indirectly) from the AWT's classes `Component` and `Container`...

- ▶ But fortunately, despite such details working with Swing is quite easy and intuitive in practice
- ▶ Remark: other words for "components" are "widgets" (Linux, Mac) and "controls" (Windows)

GUI

Components and containers



The get and set methods for these data members (i.e., fields) are provided in the class, but omitted in the UML diagram for

The font of this component.

The background color of this component.

The foreground color of this component.

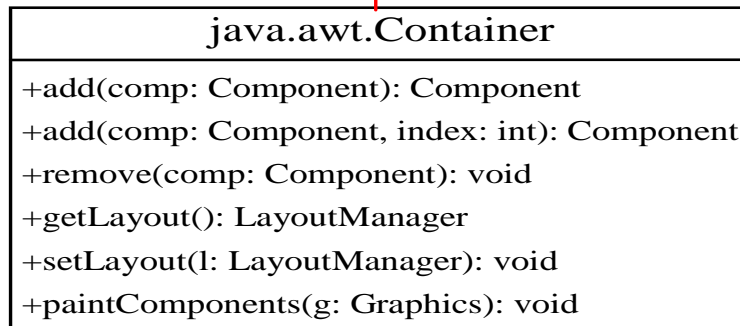
The preferred size of this component.

Indicates whether this component is visible.

Returns the width of this component.

Returns the height of this component.

getX() and getY() return the coordinate of the component's upper-left corner within its parent component.



Adds a component to the container.

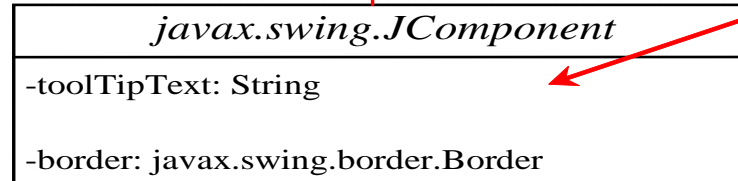
Adds a component to the container with the specified index.

Removes the component from the container.

Returns the layout manager for this container.

Sets the layout manager for this container.

Paints each of the components in this container.



The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The tool tip text for this component. Tool tip text is displayed when the mouse points on the component without clicking.

The border for this component.

GUI

Frames

- ▶ A *frame* is a window that is not contained inside any other window (we therefore say a frame is a *top-level component*). A frame contains other components, such as text, buttons, tables, or drawings.
- ▶ Use the `JFrame` class to create a frame.
- ▶ A `JFrame` object is a container. However, components are not directly added to a frame but to a sub-container called *content pane*
- ▶ It is also possible to add other containers (e.g., panels) to the content pane

GUI

Frames

- ▶ Here's an example...
- ▶ Note that often `JFrame` isn't used directly but a subclass is derived from it whose constructor configures the frame (title text, initial size, color, etc). However, this is optional.
- ▶ In the following example, remember that components which should appear inside the window are not directly added to the frame but to the frame's content pane

GUI

Frames

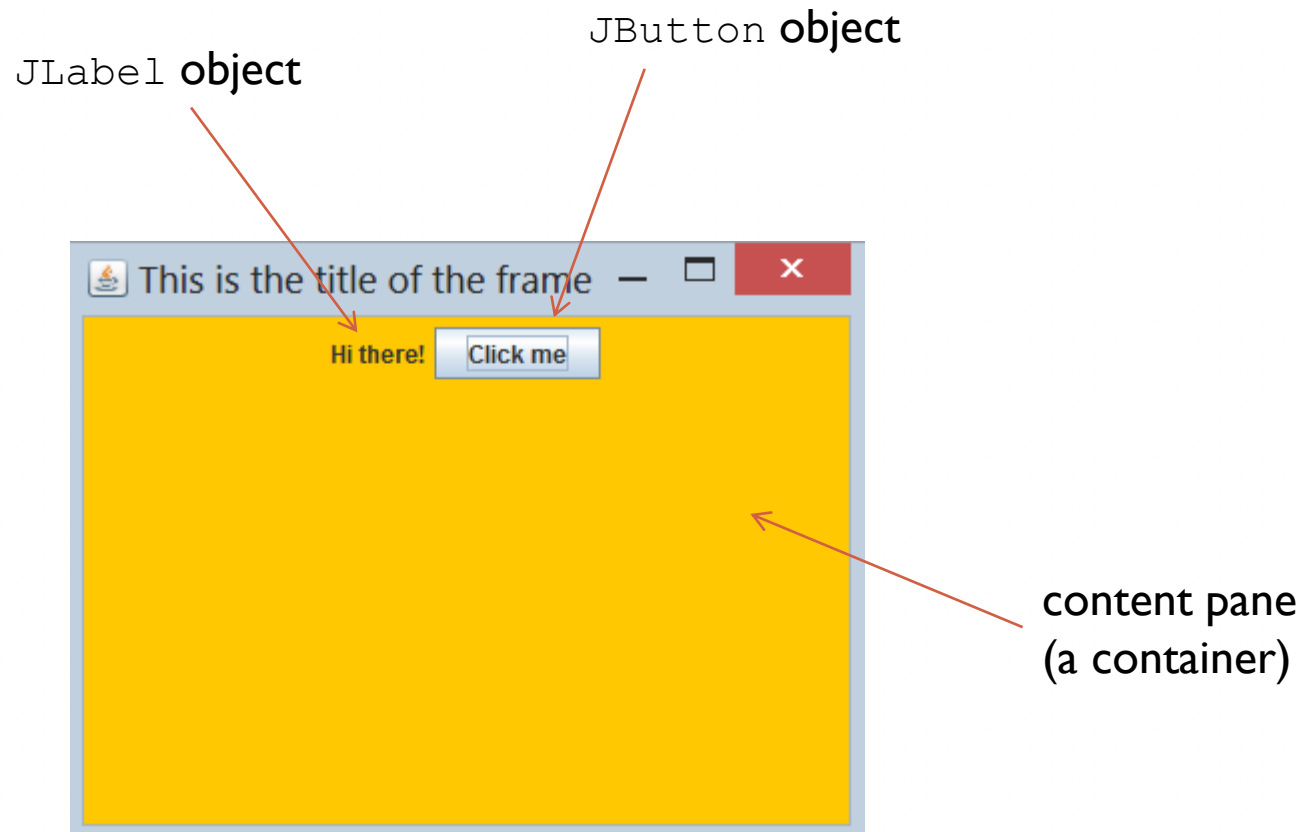
```
import javax.swing.*;

public class MyFrame extends JFrame {
    MyFrame() {
        super("Frame Title");
        setSize(400, 300); // initial window size
        setDefaultCloseOperation( // what happens if
            //user clicks on the close icon:
            JFrame.EXIT_ON_CLOSE); // close frame & end program
        Container content = getContentPane();
        content.setBackground(Color.ORANGE);
        content.setLayout(new FlowLayout()); // how components are arranged
        content.add(new JLabel("Hi there!"));
        content.add(new JButton("Click me"));
        setVisible(true); // makes the frame visible
    }

    public static void main(String[] args) {
        MyFrame myFrame = new MyFrame(); // create frame
    }
}
```

GUI

Frames



GUI

Frames

javax.swing.JFrame

+JFrame()

Creates a default frame with no title.

+JFrame(title: String)

Creates a frame with the specified title.

+setSize(width: int, height: int): void

Specifies the size of the frame.

+setLocation(x: int, y: int): void

Specifies the upper-left corner location of the frame.

+setVisible(visible: boolean): void

Sets true to display the frame.

+setDefaultCloseOperation(mode: int): void

Specifies the operation when the frame is closed.

+setLocationRelativeTo(c: Component):
void

Sets the location of the frame relative to the specified component.
If the component is null, the frame is centered on the screen.

+pack(): void

Automatically sets the frame size to hold the components in the frame.

GUI

Frames

- ▶ Tip: Try this example (and others) yourself. The best way to learn GUI programming is via coding practice.

GUI

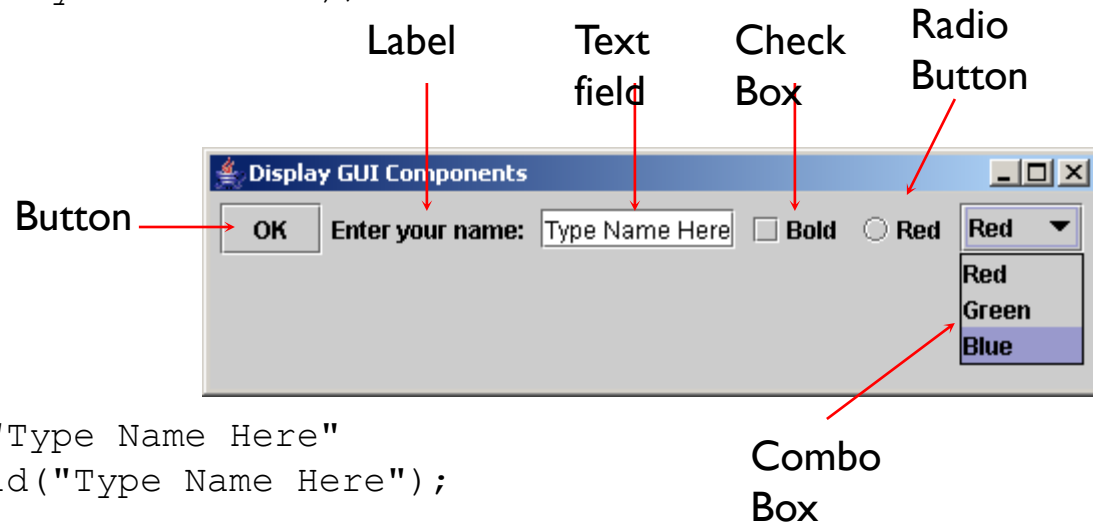
Frames

- ▶ Typical components within a frame are buttons (class `JButton`) and text labels (class `JLabel`), but there are many more....The following slide shows just a few of them...

GUI

```
JButton jbtOK = new JButton("OK"); // Create a button with text OK
```

```
// Create a label with text "Enter your name: "  
JLabel jlblName = new JLabel("Enter your name: ");
```



```
// Create a text field with text "Type Name Here"  
JTextField jtfName = new JTextField("Type Name Here");
```

```
// Create a check box with text bold  
JCheckBox jchkBold = new JCheckBox("Bold");
```

```
// Create a radio button with text red  
JRadioButton jrbRed = new JRadioButton("Red");
```

```
// Create a combo box with choices red, green, and blue  
JComboBox jcboColor = new JComboBox(new String[]{"Red",  
    "Green", "Blue"});
```

GUI

Displaying and entering text

- ▶ Buttons will be explained in detail later...
- ▶ To display single lines of text which cannot be modified by the user, you can use component class `JLabel`. A label is created simply with

```
JLabel labelA = new JLabel("label-A text");
```

- ▶ The Java program can also change the text of the label later, using `labelA.setText("some new text");`
- ▶ Don't forget to add the new component to the respective container after creation. If you add them to a panel, the panel needs to be added to the frame's content pane (or another panel...)

GUI

Displaying and entering text

- ▶ For a single line of user-editable text, use class `JTextField`.

- ▶ Example:

```
JTextField myTextField = new JTextField(10);  
// 10 is the length of the field (in number of  
characters shown)
```

- ▶ You can retrieve the text the user has entered at any time using

```
String content = myTextField.getText();
```

- ▶ You can also let your program set the text:

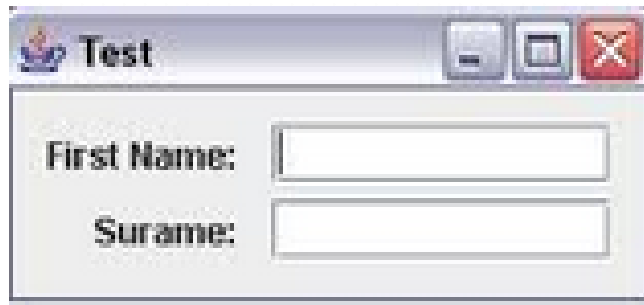
```
myTextField.setText("new content");
```

- ▶ For multi-line text areas, use class `JTextArea` instead.

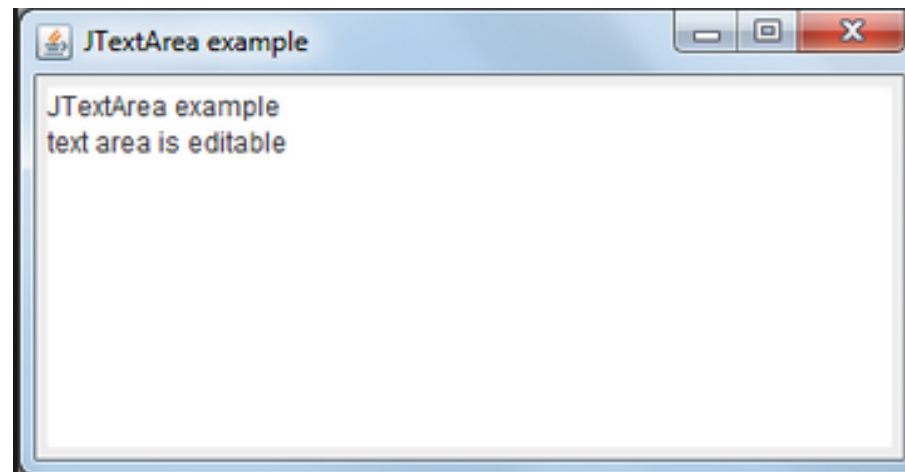
GUI

Displaying and entering text

- ▶ Two `TextField` components (plus two `JLabels`):



- ▶ `TextArea`:



(all within a `JFrame` window)

GUI

Displaying and entering text

- ▶ A few other, more complex types of GUI components:

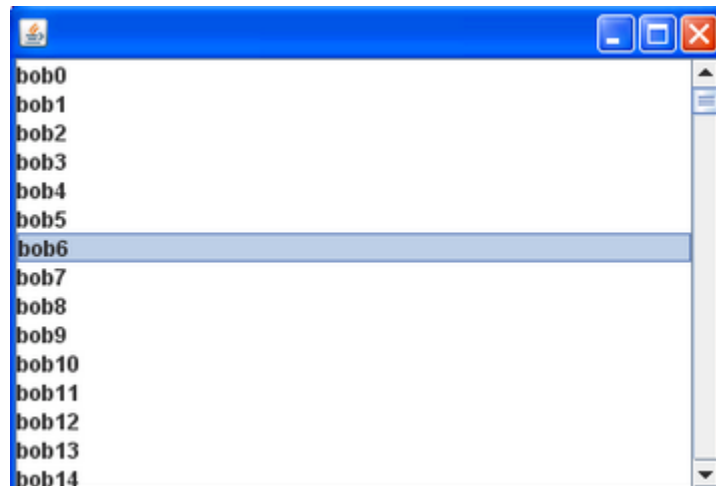
- ▶ JTable:



A screenshot of a Java Swing window titled "JTableApplication JFrame". Inside the window is a JTable component displaying a list of students. The table has three columns: "Name", "SID", and "Grade". The data is as follows:

| Name | SID | Grade |
|----------|----------|-------|
| Steven | 34189489 | D |
| Iain | 34189489 | P |
| Michael | 34193885 | R |
| Nicholas | 34189481 | M |
| Dominic | 34195995 | P |
| Matt | 34189955 | D |
| Valene | 34159199 | D |
| Nikki | 34191993 | M |
| Jonathan | 34149398 | P |
| Samuel | 34195854 | P |
| James | 34198883 | D |
| Thomas | 34188339 | D |

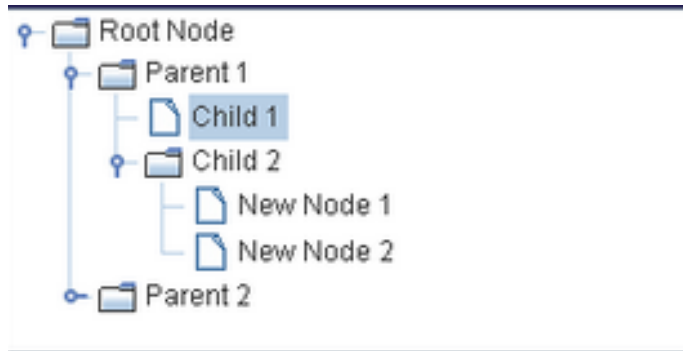
- ▶ JList:



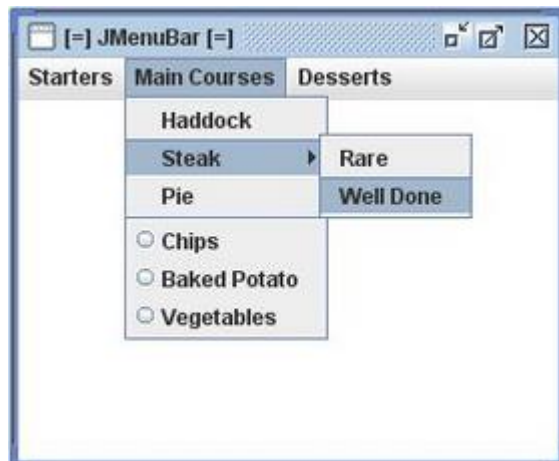
GUI

Displaying and entering text

► JTree:



► JMenuBar (attached to a frame):



GUI

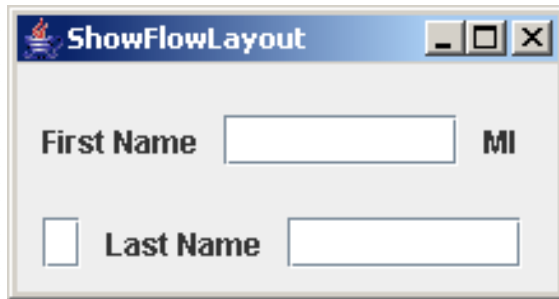
Layout Managers

- ▶ Java's *layout managers* provide a means to automatically arrange components within their container.
- ▶ The GUI components are placed inside containers. Each container has an individual layout manager attached.
- ▶ Layout managers are specified by calling the container's `setLayout(LayoutManager)` method.
- ▶ Frequently used layout manager classes are, e.g., `FlowLayout`, `BorderLayout`, `GridLayout`
- ▶ But there are many more (see Java API documentation)

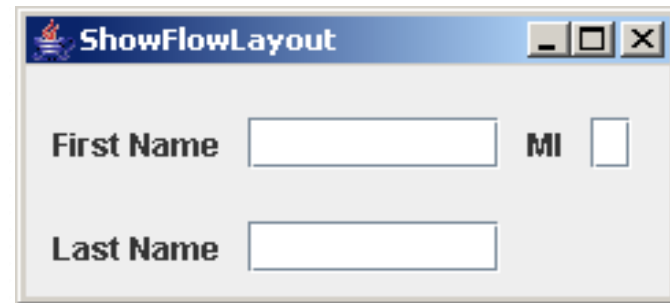
GUI

Layout Managers

- ▶ A *flow layout* arranges components in a left-to-right flow, like in a long wrapped line:



after we have resized the frame: →



GUI

Layout Managers

| java.awt.FlowLayout |
|---|
| -alignment: int -hgap: int -vgap: int |
| +FlowLayout() +FlowLayout(alignment: int) +FlowLayout(alignment: int, hgap: int, vgap: int) |

The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The alignment of this layout manager (default: CENTER).

The horizontal gap of this layout manager (default: 5 pixels).

The vertical gap of this layout manager (default: 5 pixels).

Creates a default FlowLayout manager.

Creates a FlowLayout manager with a specified alignment.

Creates a FlowLayout manager with a specified alignment, horizontal gap, and vertical gap.

GUI

Layout Managers

► Example (c stands for some container):

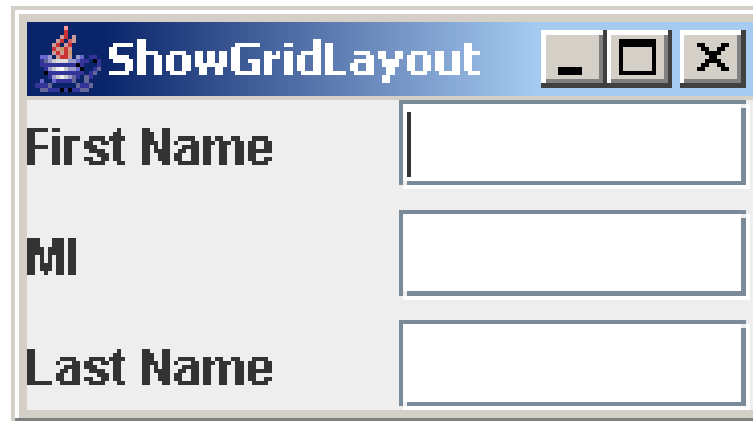
```
c.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 20));  
// Add labels and text fields to the frame  
c.add(new JLabel("First Name"));  
c.add(new JTextField(8));  
c.add(new JLabel("MI"));  
c.add(new JTextField(1));  
c.add(new JLabel("Last Name"));  
c.add(new JTextField(8));
```

► If you are using this layout for a frame, do the above before making the frame visible using `myFrame.setVisible(true);`

GUI

Layout Managers

- ▶ A *grid layout* manager aligns the components in rows and columns along the lines of an invisible grid:



GUI

Layout Managers

| java.awt.GridLayout |
|---|
| -rows: int -columns: int -hgap: int -vgap: int |
| +GridLayout() +GridLayout(rows: int, columns: int) +GridLayout(rows: int, columns: int, hgap: int, vgap: int) |

The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.



The number of rows in this layout manager (default: 1).

The number of columns in this layout manager (default: 1).

The horizontal gap of this layout manager (default: 0).

The vertical gap of this layout manager (default: 0).

Creates a default GridLayout manager.

Creates a GridLayout with a specified number of rows and columns.

Creates a GridLayout manager with a specified number of rows and columns, horizontal gap, and vertical gap.

GUI

Layout Managers

```
// Set GridLayout with three rows, two columns, and gaps  
// of size five between components (horizontal  
// and vertical gaps).  
// Variable c stands for some container (e.g., a panel)
```

```
c.setLayout(new GridLayout(3, 2, 5, 5));
```

```
// Add labels and text fields to a container:
```

```
c.add(new JLabel("First Name"));
```

```
c.add(new JTextField(8));
```

```
c.add(new JLabel("MI"));
```

```
c.add(new JTextField(1));
```

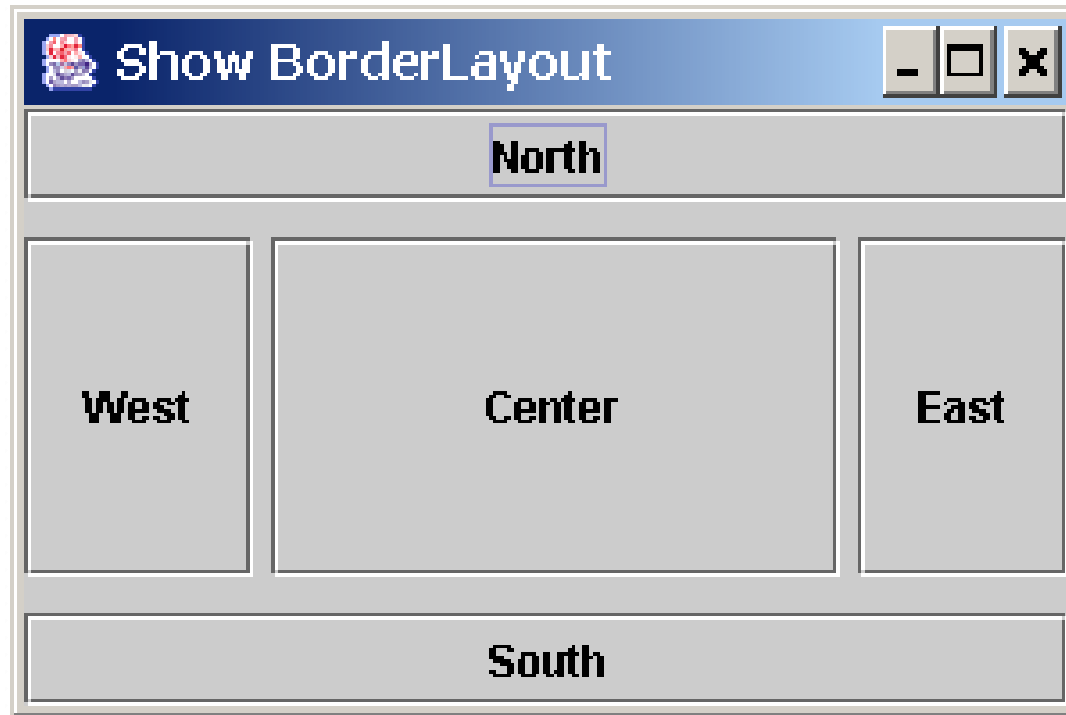
```
c.add(new JLabel("Last Name"));
```

```
c.add(new JTextField(8));
```

GUI

Layout Managers

- ▶ *Border layout* divides the container into five areas: East, South, West, North, and Center.

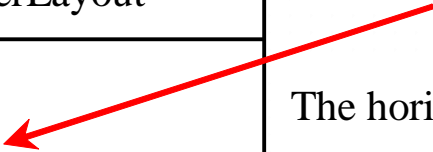


GUI

Layout Managers

| java.awt.BorderLayout |
|--|
| -hgap: int -vgap: int |
| +BorderLayout() +BorderLayout(hgap: int, vgap: int) |

The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.



The horizontal gap of this layout manager (default: 0).

The vertical gap of this layout manager (default: 0).

Creates a default BorderLayout manager.

Creates a BorderLayout manager with a specified number of horizontal gap, and vertical gap.

GUI

Layout Managers

```
// Set BorderLayout with horizontal gap of 5 and vertical gap of 10:
```

```
c.setLayout(new BorderLayout(5, 10));
```

```
// Add buttons to container c:
```

```
c.add(new JButton("East"), BorderLayout.EAST);
```

```
c.add(new JButton("South"), BorderLayout.SOUTH);
```

```
c.add(new JButton("West"), BorderLayout.WEST);
```

```
c.add(new JButton("North"), BorderLayout.NORTH);
```

```
c.add(new JButton("Center"), BorderLayout.CENTER);
```

GUI

Panels

- ▶ *Panels* are containers for *grouping* components *inside other containers*, for improved organization and easier placement of groups of components.
- ▶ You can add almost any types of components to a panel (including other panels)

GUI

Panels

▶ You can use `new JPanel()` to create a panel with a default `FlowLayout` manager, or `new JPanel(layoutManager)` to create a panel with the specified layout manager. Use the `add(Component)` method to add components to the panel, as to any kind of container.

▶ For example:

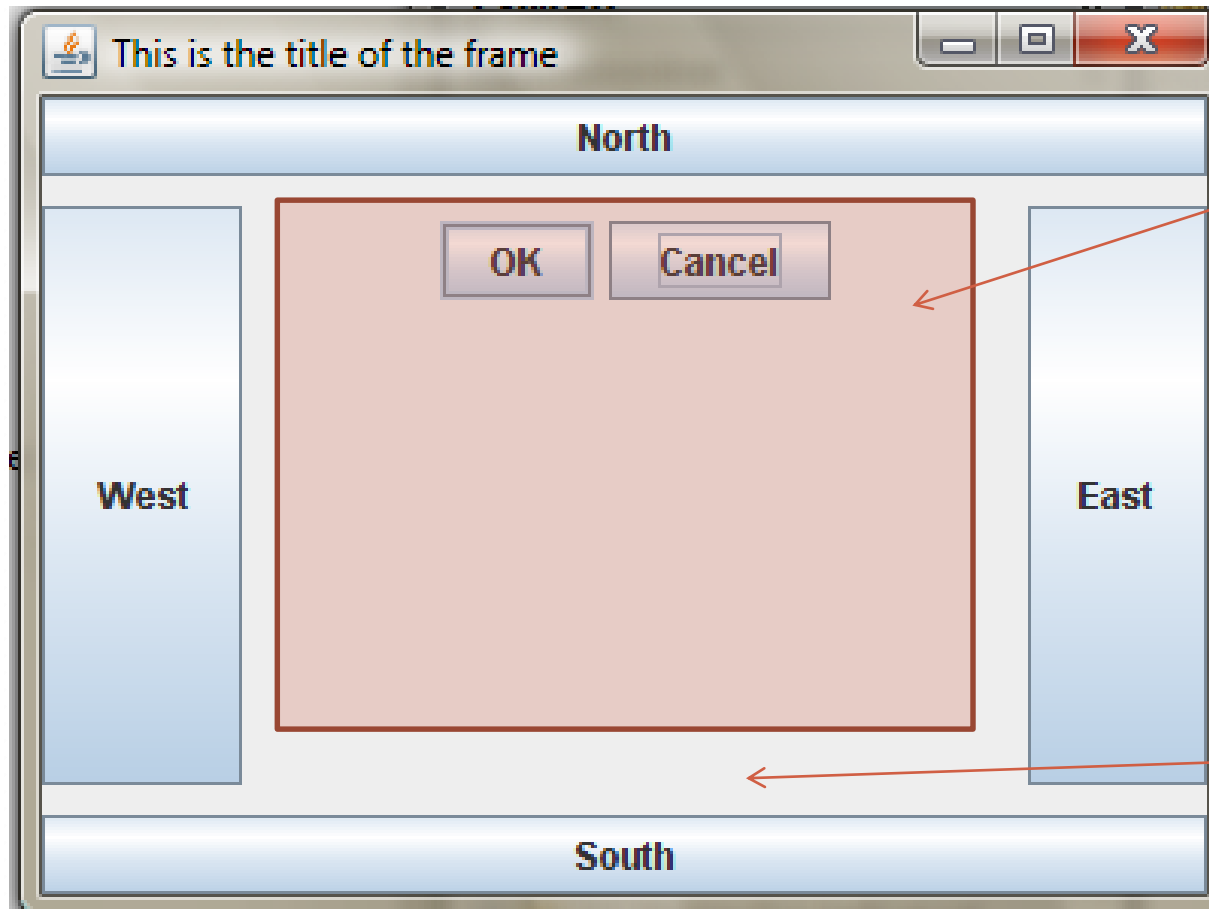
```
JPanel p = new JPanel();  
p.add(new JButton("OK"));  
p.add(new JButton("Cancel"));  
myContainer.add(p); // adds panel to some "outer" container
```

▶ The panel itself is also a component and can be added to outer containers (e.g., the frame or another panel) in the same fashion, to yield nested containers.

▶ The panel can have a different layout manager than the outer container.

GUI

Panels



Panel with two buttons. Has its own layout manager

Outer container which contains the panel and four buttons

GUI

Basic event handling

- ▶ We have seen how to create buttons, but until now pushing them had no effect...
- ▶ Every time the user pushes a button, Java creates an event. Making buttons work is done by *event handling*.
- ▶ We say that buttons are an *event source* which *fire* events.
- ▶ Event handling is also important for, e.g., *menus* and updating the data in complex components such as → JLists or JTables

GUI

Basic event handling

- ▶ Events appear in Java Swing programs as objects of certain event classes.
 - ▶ E.g., "pushing" a button component (e.g., by clicking on it) automatically generates an instance of class `ActionEvent`
 - ▶ Clicking somewhere creates an `MouseEvent` (in addition to an `ActionEvent`, in case a button was clicked).
 - ▶ Pushing a keyboard key generates an object of class `KeyEvent`

GUI

Basic event handling

- ▶ The object which handles the generated event is called *listener* (Swing terminology) or *event handler* (in many other GUI frameworks). It specifies what happens when the event (e.g., mouse click) occurs.
- ▶ Java provides several interfaces and classes for listeners. You create your own event handlers by creating classes which implement these interfaces.
- ▶ You need to associate your buttons with appropriate listeners.

Example:

```
JButton okButton = new JButton("OK");

ActionListener okListener = new OKListener();
/* OKListener being your implementation of the built-in interface
ActionListener */

okButton.addActionListener(okListener);
```


GUI

Basic event handling

- ▶ Your listener class just needs to provide a single method which "catches" (handles) the respective events at runtime:

```
class OKListener implements ActionListener {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("User has clicked on the OK button!");  
    }  
}
```

- ▶ The method `actionPerformed` is called at runtime automatically by Java each time the respective button is pushed. It is executed while the rest of your program continues running
- ▶ Not only buttons can have listeners, but several other kinds of GUI components too, such as panels, list and tables components.

GUI

Pulldown menus

- ▶ A *pulldown menu* also requires event handling...

```
JMenuBar menuBar = new JMenuBar();
```

```
JMenu menu1 = new JMenu("Menu 1");
```

```
menu1.setMnemonic(KeyEvent.VK_A); /*A so-called mnemonic or shortcut key: menu  
pops up if Alt+A is pressed (Windows)*/
```

```
menuBar.add(menu1);
```

```
JMenuItem menuItem1 = new JMenuItem("Item 1.1");
```

```
menu1.add(menuItem1);
```

```
JMenuItem menuItem2= new JMenuItem("Item 1.2");
```

```
menu1.add(menuItem2);
```

```
topLevelFrame.setJMenuBar(menuBar);
```

GUI

Pulldown menus & anonymous classes

- ▶ Event handling for such menus is basically the same as for push buttons.
- ▶ This time we implement the event listener using a so-called *anonymous class* - an important kind of inner class which does not have a class name:

```
menuItem2.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent event) {  
  
        ... // here goes the code which specifies what happens when  
             the user selects menuItem2  
  
    }  
});
```

GUI

Pulldown menus & anonymous classes

- ▶ The code on the previous slide is a sort of shortcut for creating a "disposable" class which doesn't require a name:

```
menuItem2.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent event) {  
        ... // what happens when the user selects menuItem2  
    }  
});
```

- ▶ An new object is created and provided as argument for method `addActionListener(...)`. The new object is an instance of an unnamed class which implements interface `ActionListener`. The only purpose of this *anonymous class* is to provide a method implementation (of method `actionPerformed(...)` which is specified in interface `ActionListener`) and to pass this method on to some other method (here: `addActionListener(...)`).

GUI

Pulldown menus & anonymous classes

- ▶ You could likewise use an ordinary class (in the example before, you'd create a normal class which implements `ActionListeners` and pass on an object of this class to `menuItem2.addActionListener()`)
- ▶ But that would require more coding overhead. And since you don't require this class elsewhere, the class name would be unnecessary.
- ▶ Anonymous classes can also be used to specify more than one method (see the `JTable` example later).
- ▶ Instead of an interface, the anonymous class could also extend a class, using exactly the same syntax pattern:

```
new Superclass() {
```

```
    (...overriding/implementing methods SuperClass...)
```

GUI

Tables & anonymous classes

- ▶ An example for anonymous classes with multiple methods is the specification of *data models* for `JTable` and `JList` components, that is, to specify how the data that these GUI components should display should be computed:

```
TableModel dataModel = new AbstractTableModel() { // defines table content
    public int getColumnCount() { return 5; } // number of table columns
    public int getRowCount() { return 30; } // number of table rows
    public Object getValueAt(int row, int column) {
        return "This is cell " + row + ", " + column;
    }
    public String getColumnName(int column) { return ""+column; }
};

JTable table = new JTable(dataModel);

JScrollPane scrollPane = new JScrollPane(table); // makes the table scrollable
...
someContainer.add(scrollPane);
...
```

GUI

Tables & anonymous classes

► Alternative approach to set up a very basic table (less powerful):

```
Object[][] data = { { "some value", "xyz", new Double(123.45) },  
                    { "some other value", new Integer(33), new Boolean(false) },  
                    // ...  
};
```

```
String[] columnNames = { "Column A", "Column B", "Column C" };
```

```
JTable table = new JTable(data, columnNames);  
JScrollPane scrollPane = new JScrollPane(table);  
getContentPane().add(scrollPane);  
...
```

GUI

Tables & anonymous classes

| 0 | 1 | 2 | 3 | 4 | |
|-------------------|-------------------|-------------------|-------------------|-------------------|---|
| This is cell 0, 0 | This is cell 0, 1 | This is cell 0, 2 | This is cell 0, 3 | This is cell 0, 4 | ▲ |
| This is cell 1, 0 | This is cell 1, 1 | This is cell 1, 2 | This is cell 1, 3 | This is cell 1, 4 | |
| This is cell 2, 0 | This is cell 2, 1 | This is cell 2, 2 | This is cell 2, 3 | This is cell 2, 4 | |
| This is cell 3, 0 | This is cell 3, 1 | This is cell 3, 2 | This is cell 3, 3 | This is cell 3, 4 | |
| This is cell 4, 0 | This is cell 4, 1 | This is cell 4, 2 | This is cell 4, 3 | This is cell 4, 4 | ≡ |
| This is cell 5, 0 | This is cell 5, 1 | This is cell 5, 2 | This is cell 5, 3 | This is cell 5, 4 | |
| This is cell 6, 0 | This is cell 6, 1 | This is cell 6, 2 | This is cell 6, 3 | This is cell 6, 4 | |
| This is cell 7, 0 | This is cell 7, 1 | This is cell 7, 2 | This is cell 7, 3 | This is cell 7, 4 | |
| This is cell 8, 0 | This is cell 8, 1 | This is cell 8, 2 | This is cell 8, 3 | This is cell 8, 4 | |
| This is cell 9, 0 | This is cell 9, 1 | This is cell 9, 2 | This is cell 9, 3 | This is cell 9, 4 | |
| This is cell 1... | This is cell 1... | This is cell 1... | This is cell 1... | This is cell 1... | |
| This is cell 1... | This is cell 1... | This is cell 1... | This is cell 1... | This is cell 1... | |
| This is cell 1... | This is cell 1... | This is cell 1... | This is cell 1... | This is cell 1... | |
| This is cell 1... | This is cell 1... | This is cell 1... | This is cell 1... | This is cell 1... | |
| This is cell 1... | This is cell 1... | This is cell 1... | This is cell 1... | This is cell 1... | ▼ |
| This is cell 1... | This is cell 1... | This is cell 1... | This is cell 1... | This is cell 1... | |

GUI

Tables & anonymous classes

- ▶ Table and list components can send and react on events:
 - ▶ user-triggered events (e.g., selection of a certain item)
 - ▶ data manipulation events (data in the table/list has changed and some other component needs to be informed about this update)
- ▶ Example (handling table row selection events):

```
someJTableComponent.getSelectionModel().addListSelectionListener(  
    new ListSelectionListener() {  
        // method which handles the event that the user selected a row  
        public void valueChanged(ListSelectionEvent e) {  
            ...  
        }  
    }  
);
```

GUI

Pulldown menues & anonymous classes

- ▶ In Java ≥ 8 , anonymous classes can be replaced by the more elegant *lambda expressions*... (anonymous functions), which will be covered in one of the next lectures.
- ▶ Also observe that anonymous classes are not restricted to GUI programming.

USER INTERFACES

Swing and multithreading

- ▶ Recall that one use case for multithreading is GUIs...
- ▶ The use of Swing with multiple threads requires some care.
- ▶ If two or more threads operate on the same GUI, the GUI becomes a shared resource.
- ▶ Letting multiple threads modify a GUI without proper thread synchronization can cause all kinds of concurrency-related problems, such as race conditions.
- ▶ Thread synchronization in Swing is closely related to event handling.

USER INTERFACES

Swing and multithreading

- ▶ Unfortunately, most Swing components are not thread-safe (that is, they are not automatically protected against race conditions and other multithreading-related problems). Reasons:
 - ▶ Thread-safe code might slow down single-thread applications.
 - ▶ Locking objects in order to prevent race conditions can cause deadlocks if inexperienced programmers use such objects improperly.
- ▶ The fact that Swing is (with a few exceptions) not thread-safe means that you need to make sure yourself that no multithreading-related problems occur with your GUI in case your program uses multiple threads
- ▶ Only a few Swing methods are thread-safe and can thus be invoked by any threads directly. E.g., `setText` of `JTextComponent` or `repaint/revalidate` of all components (see Java API documentation).
- ▶ Further details about Swing-multithreading are quite complex and out of scope of today's lecture