ENTERPRISE JAVA PROGRAMMING / PROGRAMMING II

MODULES CT545 / CT875, SEMESTER 2, ACADEMIC YEAR 2020-2021
NATIONAL UNIVERSITY OF IRELAND, GALWAY

Lecture 5a

Lecturer: Dr Patrick Mannion

TOPICS

- *Graphical User Interfaces* (part I) and *event handling*
- Along the way (equally important), using GUIs as example: more about *inner classes* (part I), *anonymous classes*

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

GUI

Swing

- Older approach:
  Abstract Window Toolkit (AWT). Still used as a basis for Swing, but nowadays rarely used directly by application programmers (exception: Java on certain old mobile phones)

GUI
Swing

- Alternatives to Swing:
  - *The Standard Widget Toolkit* (SWT), originally by IBM. 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 really popular.
  - *JavaFX* is more advanced and "cleaner" than Swing, but Swing is still more popular.

  - **...or just use HTML+JavaScript (this is the recommended approach for GUIs in a client/server environment, where only the server can use Java - a typical situation**)
  - For this reason we are not going to spend a lot of time on traditional desktop GUI development. Most modern GUI development uses browser-based clients, which we will discuss in the Java EE part of the module.

GUI
Swing

- Remark: Java is nowadays mostly a server-side technology.
- GUIs using Java are mostly used in standalone Java programs or a so-called *fat clients* (exception: Android)
- Java is rarely used to implement plain clients - older Java technologies for that purpose (*applets*) are deprecated and should not be used anymore (very high security risk!).

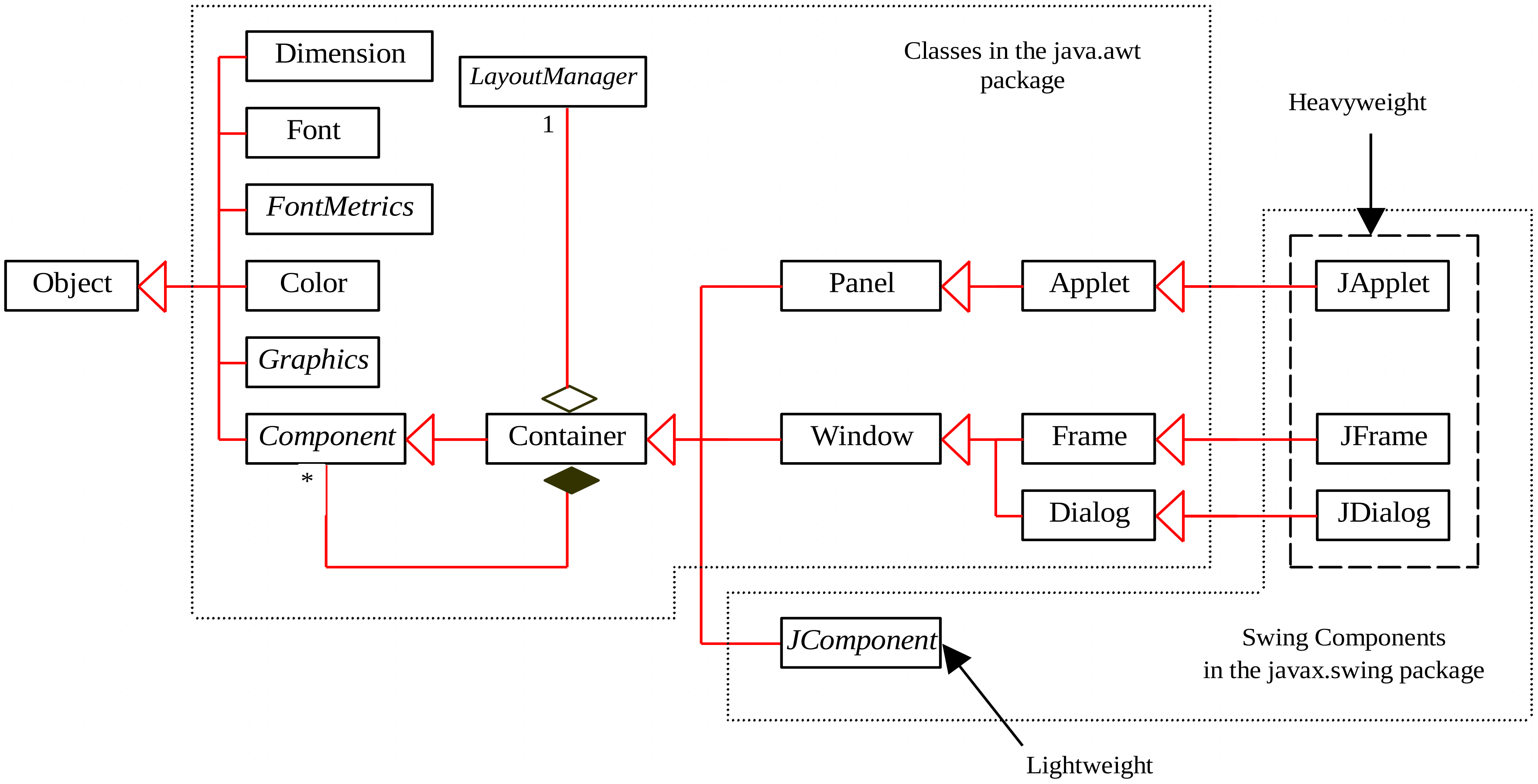
Instead, normally a HTML/JavaScript client is created which runs in a web browser and interacts with Java code on the server (e.g., *servlet* or socket-based).

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...

GUI

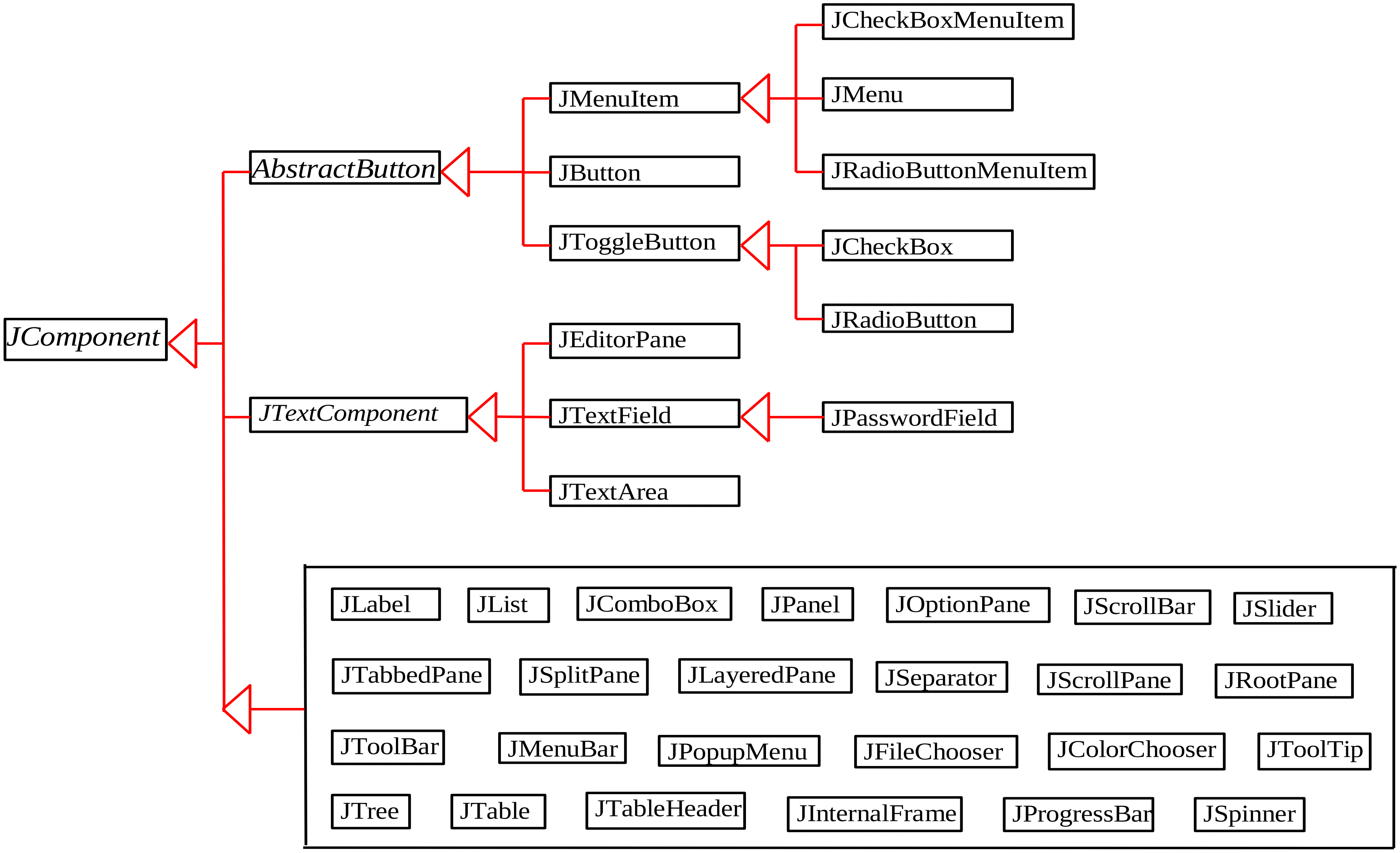
A few important Swing and AWT classes (in UML)



Some of the figures and examples in this lecture from Daniel Liang: Introduction to Java Programming, Prentice Hall 2007

GUI

Even more Swing classes



GUI

- The objects which the user sees on the screen (buttons, text fields, lists, windows, etc.) are called (GUI) *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*** in the 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 exactly reflect this terminology:

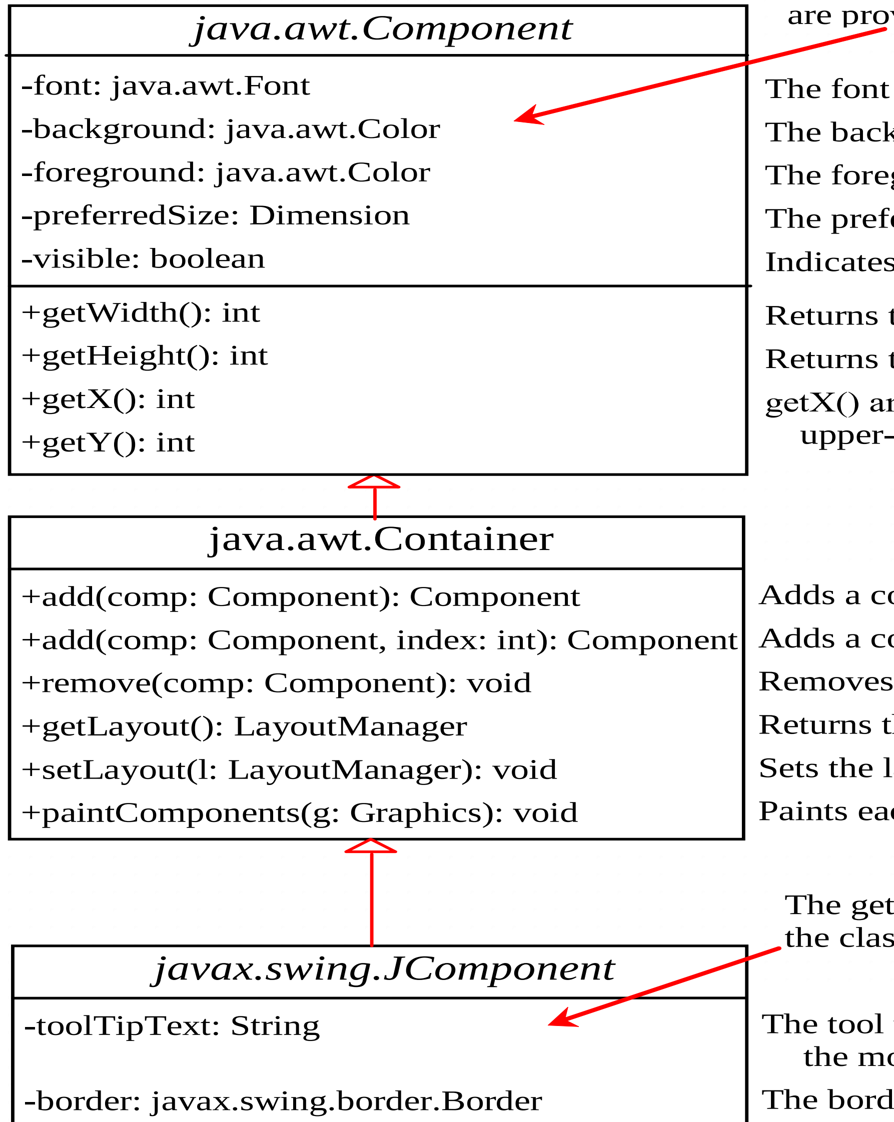
  The abstract class `JComponent` is derived from (i.e., is a subclass of) the AWT's container class `Container`, which is derived from AWT's `Component` class!

  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) 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



java.awt.Component
-font: java.awt.Font
-background: java.awt.Color
-foreground: java.awt.Color
-preferredSize: Dimension
-visible: boolean
+getWidth(): int
+getHeight(): int
+getX(): int
+getY(): int
java.awt.Container
+add(comp: Component): Component
+add(comp: Component, index: int): Component
+remove(comp: Component): void
+getLayout(): LayoutManager
+setLayout(l: LayoutManager): void
+paintComponents(g: Graphics): void
javax.swing.JComponent
-toolTipText: String
-border: javax.swing.border.Border

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 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

- On the next slide there'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

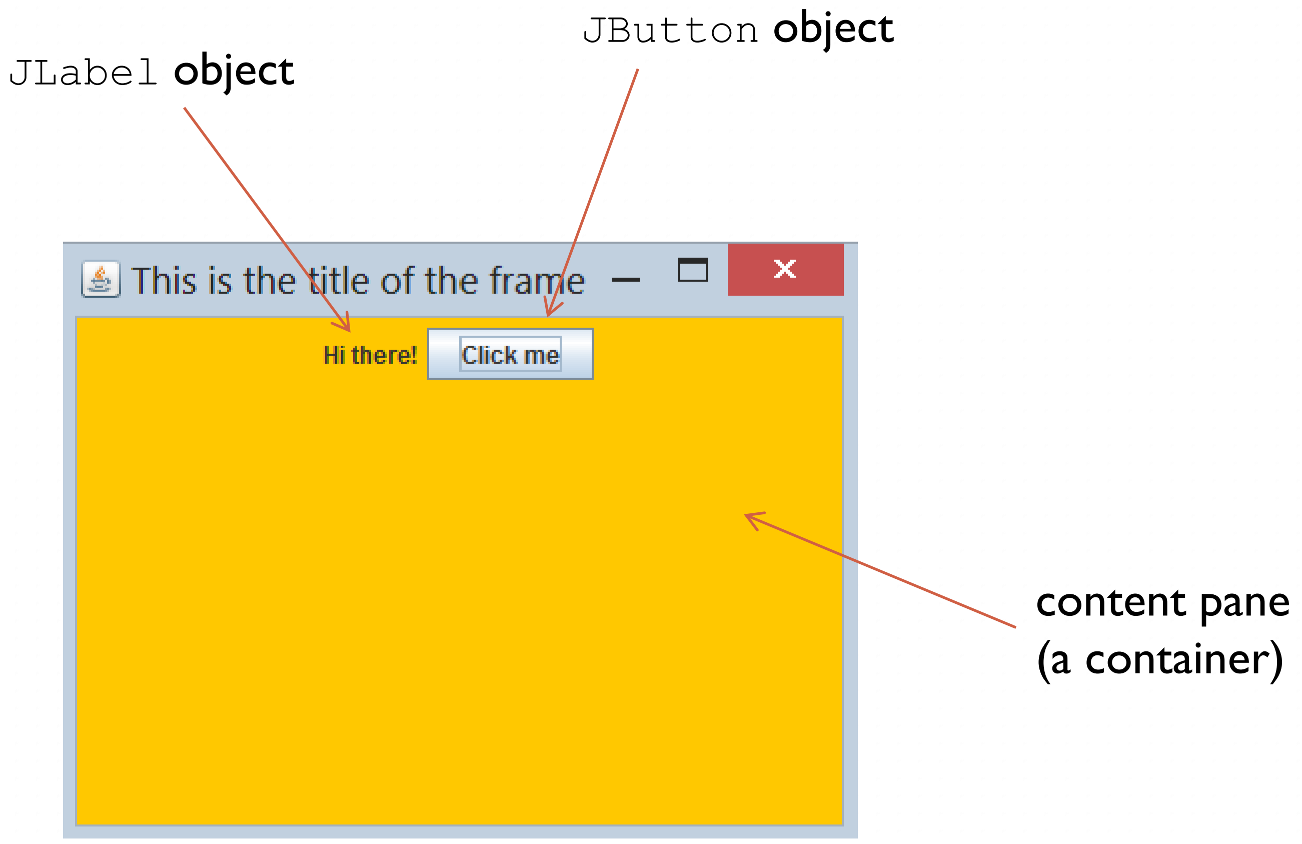
GUI

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


JLabel object
JButton object
This is the title of the frame
Hi there!
Click me
content pane
(a container)

GUI

Frames

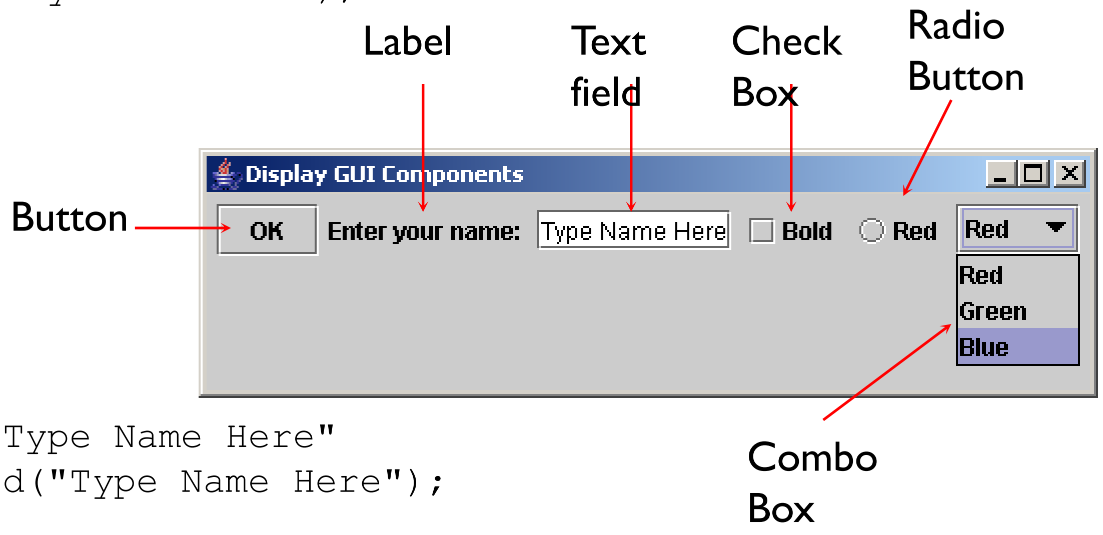
javax.swing.JFrame	
+JFrame()	Creates a default frame with no title.
+JFrame(title: String)	Creates a frame with the specified title.
+setSize(width: int, height: int): void	Specifies the size of the frame.
+setLocation(x: int, y: int): void	Specifies the upper-left corner location of the frame.
+setVisible(visible: 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

- 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();
```

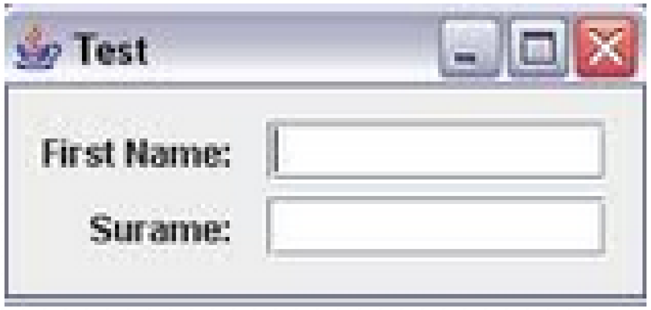
- You can also let your program set the text:

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

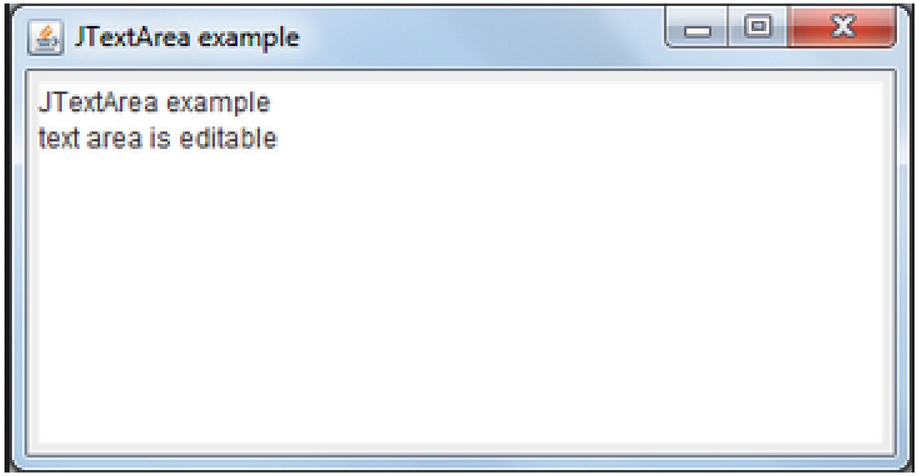
- For multi-line text areas, use class `JTextArea` instead.

GUI
Displaying and entering text

- Two `JTextField` components (plus two `JLabels`):



- `JTextArea`:

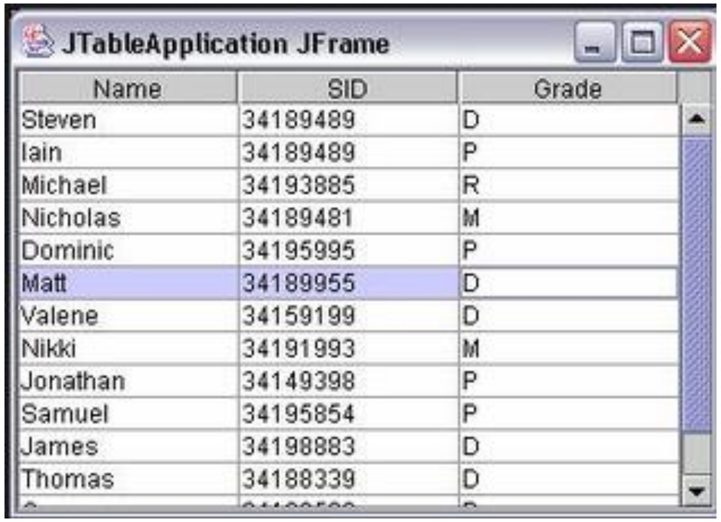


JTextArea example
text area is editable

(all within a `JFrame` window)

GUI
Displaying and entering text

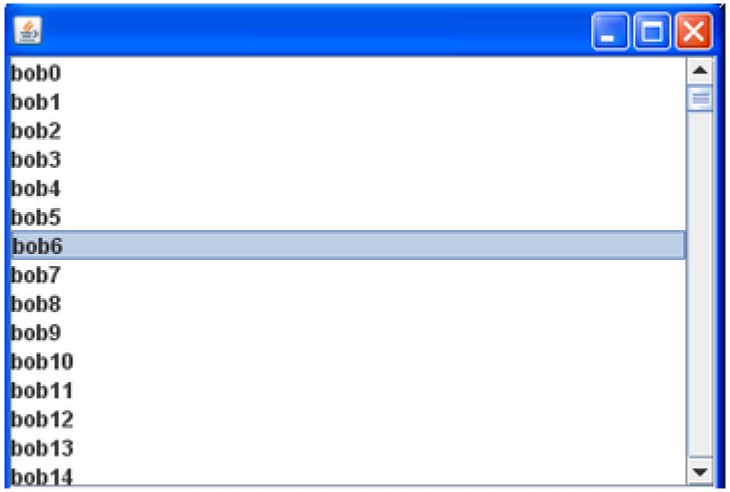
- A few other, more complex types of GUI components:
  - `JTable:`



JTableApplication JFrame

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

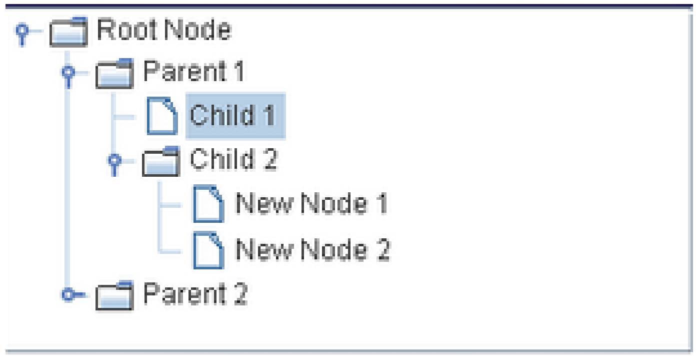
  - `JList:`



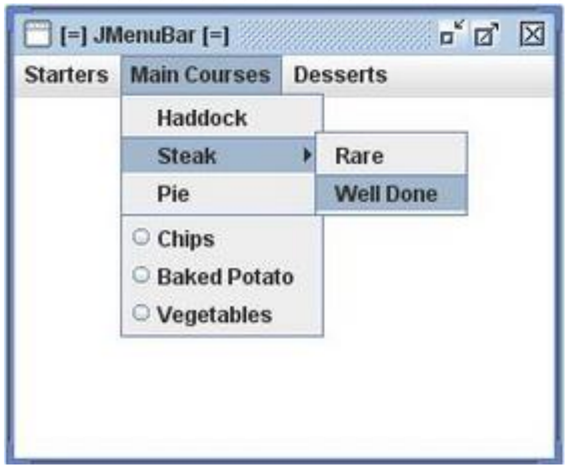
```
bob0
bob1
bob2
bob3
bob4
bob5
bob6
bob7
bob8
bob9
bob10
bob11
bob12
bob13
bob14
```

GUI
Displaying and entering text

- `JTree`:



- `JMenuBar` (attached to a frame):



GUI
Displaying and entering text

- Some of these components will be explained in detail later. But you will also need to lookup Swing classes and their methods in the Java API:
  - Java <= 6:

    `http://docs.oracle.com/javase/6/docs/api/javax/swing/package-summary.html`
  - Java 7, 8, 9 (mostly):

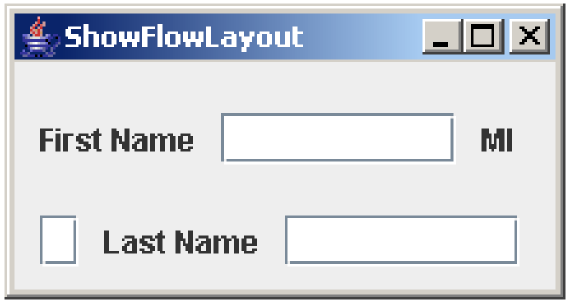
    `http://docs.oracle.com/javase/7/docs/api/overview-summary.html`
- Java Swing is one of the rare cases where using Java 6 and 7/8 makes a (small) *syntactic* difference. E.g. component `JComboBox` is generic in Java 7 but not in Java 6.

GUI
Layout Managers

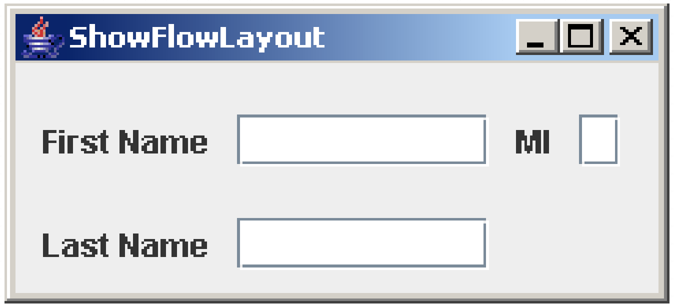
- Java’s *layout managers* provide a means to automatically <u>arrange</u> a number of components in a certain order.
- The GUI components are placed in containers. Each container has a layout manager to arrange the components within that container.
- Layout managers are specified by calling the container's `setLayout(LayoutManager)` method.
- A few frequently used layout manager classes (but there are many more):
  `FlowLayout, BorderLayout, GridLayout`

GUI
Layout Managers

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

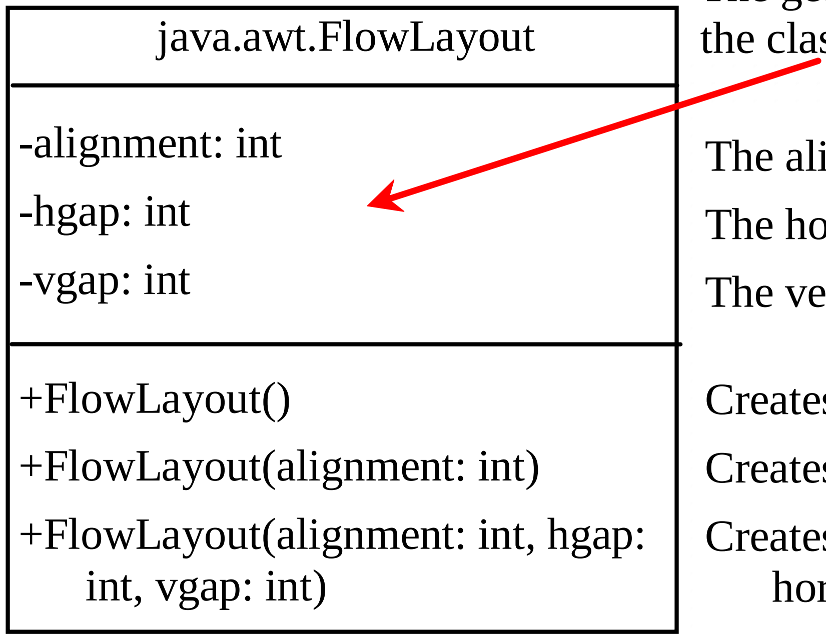


if we resize the frame: $\longrightarrow$



GUI

Layout Managers



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

java.awt.FlowLayout	
-alignment: int	The alignment of this layout manager (default: CENTER).
-hgap: int	The horizontal gap of this layout manager (default: 5 pixels).
-vgap: int	The vertical gap of this layout manager (default: 5 pixels).
+FlowLayout()	Creates a default FlowLayout manager.
+FlowLayout(alignment: int)	Creates a FlowLayout manager with a specified alignment.
+FlowLayout(alignment: int, hgap: int, vgap: int)	Creates a FlowLayout manager with a specified alignment, horizontal gap, and vertical gap.

GUI
Layout Managers

- Example (`c` stands for some container):

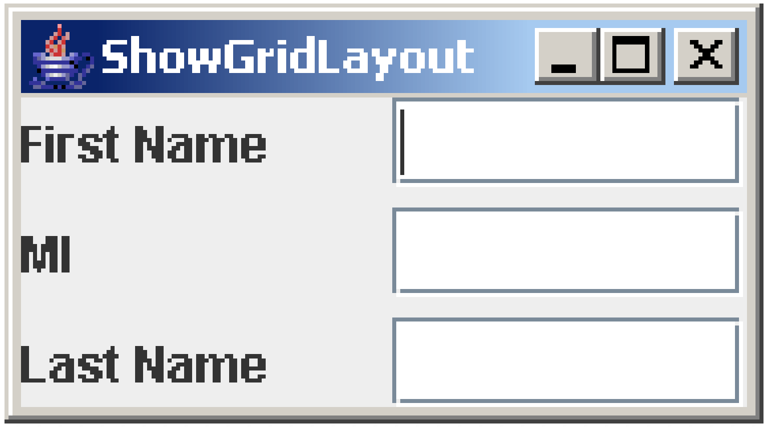
```
c.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 20));
// Add labels and text fields to the frame
c.add(new JLabel("First Name"));
c.add(new JTextField(8));
c.add(new JLabel("MI"));
c.add(new JTextField(1));
c.add(new JLabel("Last Name"));
c.add(new JTextField(8));
```

- If you are using this layout for a frame, do the above <u>before</u> 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

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

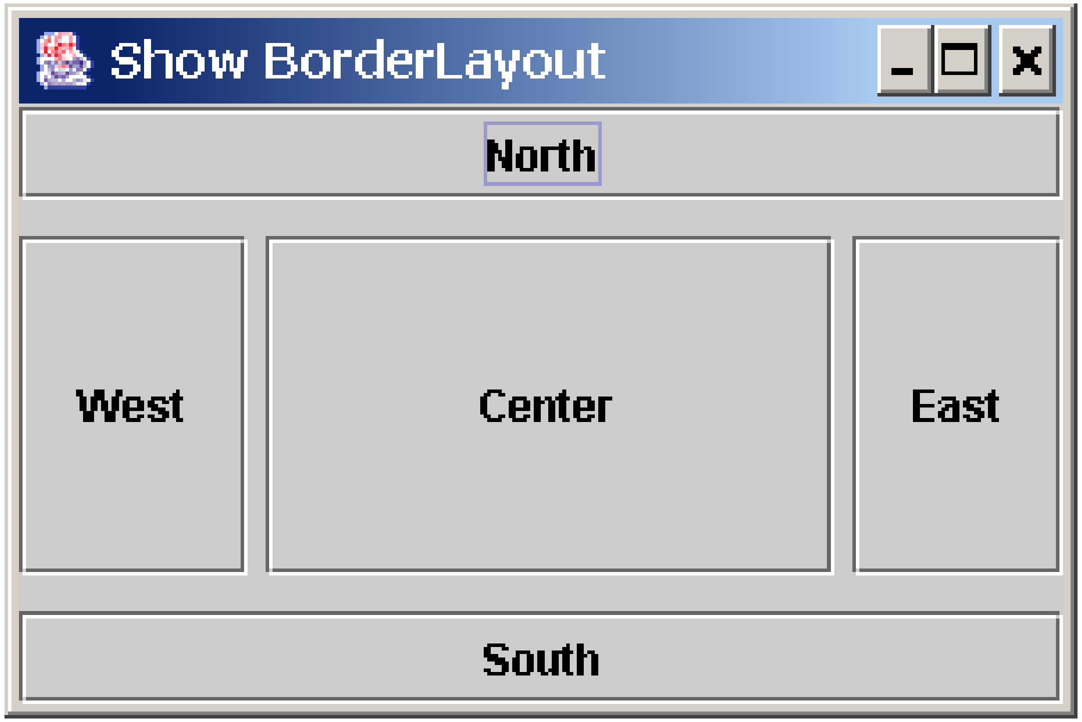
java.awt.GridLayout	
-rows: int	The number of rows in this layout manager (default: 1).
-columns: int	The number of columns in this layout manager (default: 1).
-hgap: int	The horizontal gap of this layout manager (default: 0).
-vgap: int	The vertical gap of this layout manager (default: 0).
+GridLayout()	Creates a default GridLayout manager.
+GridLayout(rows: int, columns: int)	Creates a GridLayout with a specified number of rows and columns.
+GridLayout(rows: int, columns: int, hgap: int, vgap: int)	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

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

java.awt.BorderLayout	
-hgap: int	The horizontal gap of this layout manager (default: 0).
-vgap: int	The vertical gap of this layout manager (default: 0).
+BorderLayout()	Creates a default BorderLayout manager.
+BorderLayout(hgap: int, vgap: int)	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
Colors

- You can set colors of GUI components by using the `java.awt.Color` class. Colors are made of red, green, and blue components, each of which is represented by a byte value that describes its intensity, ranging from 0 (darkest shade) to 255 (lightest shade).
This is known as the *RGB model.*

```
Color c = new Color(r, g, b);
```

`r`, `g`, and `b` specify a color by its red, green, and blue components.

- Example:

```
Color c = new Color(228, 100, 255);
```

GUI
Color

- There are thirteen "standard colors" (*black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow*).

- For easier use, the standard colors are predefined instances of class `Color`, defined as public final static fields in class `Color`:

```
BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN,
LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED,
WHITE,YELLOW.
```

GUI
Colors

- You can use the following methods to set a component's background and foreground colors, e.g.:

```
button.setBackground(Color.RED);

button.setForeground(new Color(34,200,0));
```

GUI

Panels

- *Panels* act as containers for grouping user interface components *inside other containers*, for improved organization and easier placement of groups of components.
- You can add almost any other components to a panel
- You can also place panels inside a panel...

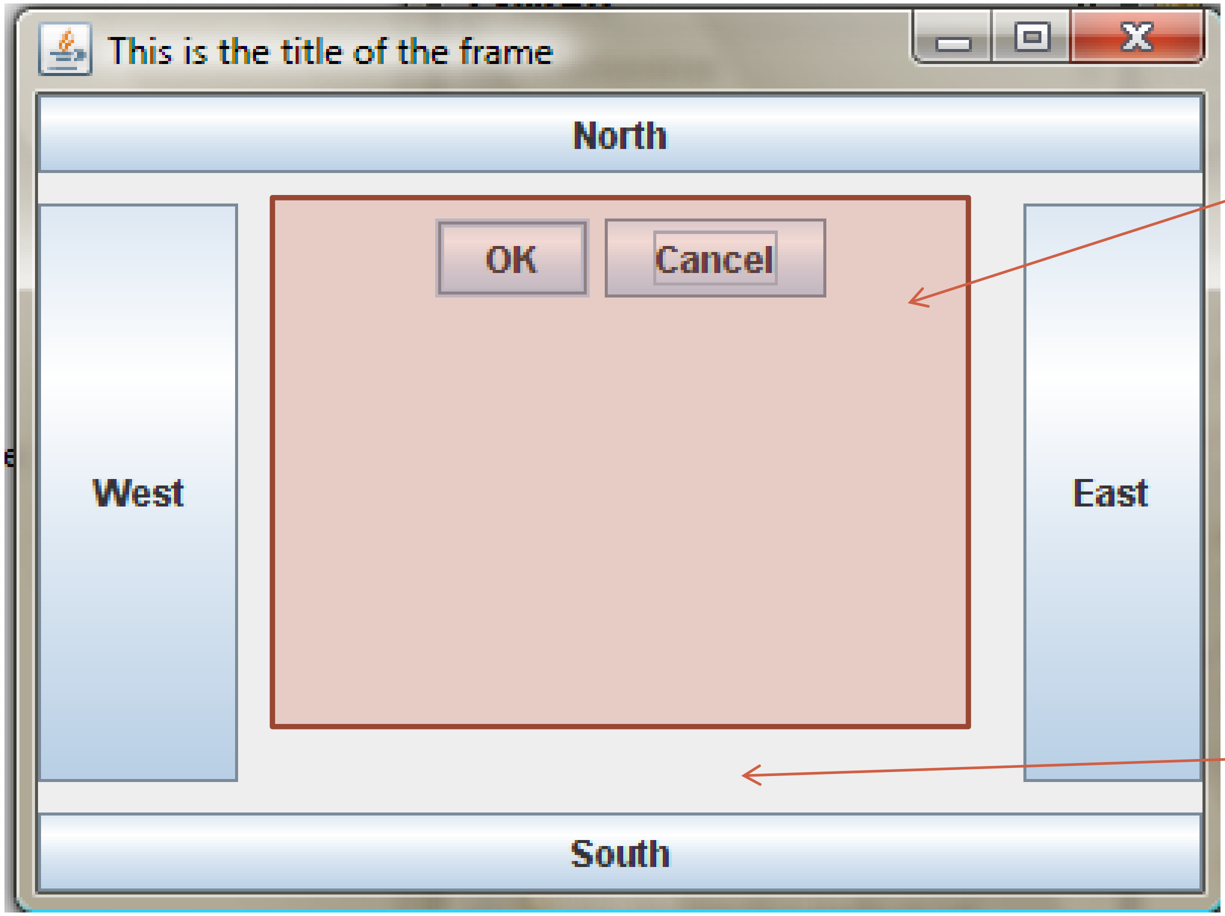
GUI
Panels

- 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

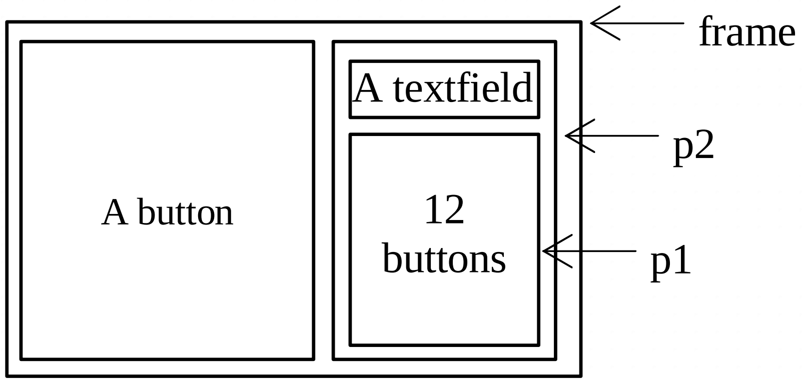


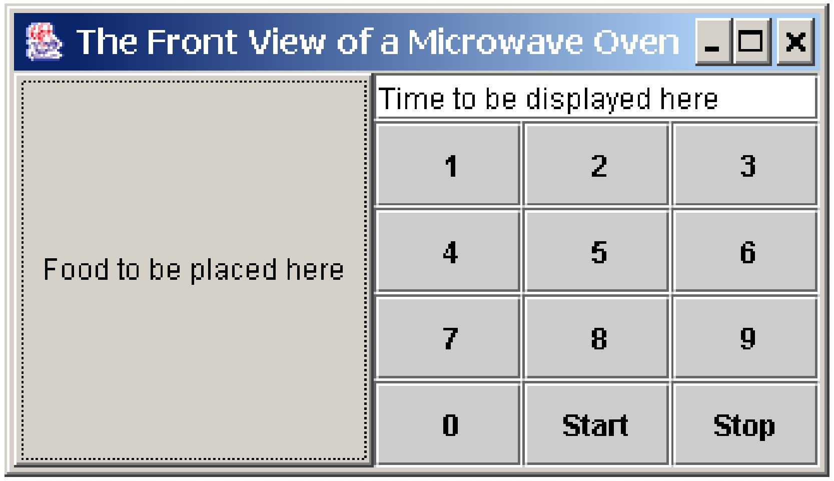
Panel with two buttons. Has its own layout manager

Outer container which contains the panel and four buttons

GUI
Panels

- Yet another example: a GUI of a (virtual) microwave oven:





GUI

Another example for the use of panels

```
JPanel p1 = new JPanel();
p1.setLayout(new GridLayout(4, 3));

// Add buttons to the panel
for (int i = 1; i <= 9; i++) {
  p1.add(new JButton("" + i));  // Why the "" + ?
}

p1.add(new JButton("" + 0));
p1.add(new JButton("Start"));
p1.add(new JButton("Stop"));

// Create panel p2 to hold a text field and panel p1:
JPanel p2 = new JPanel(new BorderLayout());
p2.add(new JTextField("Time to be displayed here"),
  BorderLayout.NORTH);
p2.add(p1, BorderLayout.CENTER);

// add panel p2 and another button to the frame:
contentPane.add(p2, BorderLayout.EAST);
contentPane.add(new JButton("Food to be placed here"),
  BorderLayout.CENTER);
```

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
- The core principles of event handling are similar in most major programming languages and GUI toolkits, so you can apply them also in, e.g., JavaScript…
  Technical details can be quite different though!

GUI
Basic event handling

- Events appear in Java Swing programs are 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 most other frameworks). It specifies what happens if the event (e.g., user action) 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 Swing 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 menues

- 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 menues & anonymous classes

- Event handling for such menus is basically the same as for push buttons. Here's an implementation using a functor.
- However, this time the functor is implemented using a so-called *anonymous class* - an certain kind of *inner class* without a class name:

```
menuItem2.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent event) {

       … // what happens when the user selects menuItem2
  }
});
```

GUI
Pulldown menues & anonymous classes

- This code is a way of creating a "disposable" functor class which doesn't require a class name:

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

- A new object is created and provided as argument for method `addActionListener(...)`.The new object is an instance of an <u>unnamed class</u> 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 menues & 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 (functor) of this class to `menuItem2.addActionListener()`)
- But that would require more coding overhead. And since you don't need this class elsewhere, a class name would be redundant.
- 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

- Another example for anonymous classes is the creation of *data models* for `JTable` and `JList` components:

```
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 way 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);
...
```

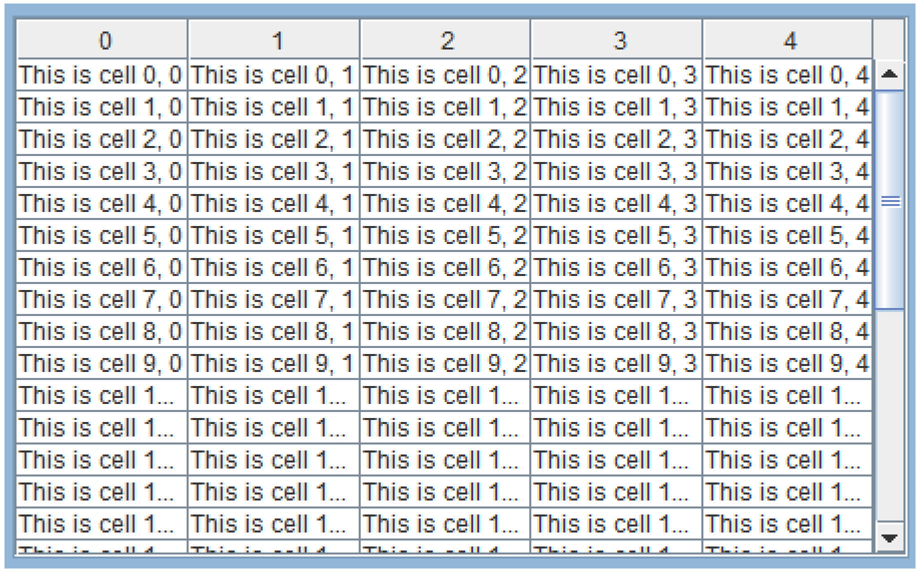
GUI
Tables & anonymous classes

- Remark: table and list components can create events 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

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...

GUI
Pulldown menues & anonymous classes

- In Java >=8, anonymous classes for the purpose of creating functors can often (not always!) be replaced by the more elegant *lambda expressions*... (a form of "anonymous functions")
- More about these in one of the next lectures

- Note that anonymous classes are of course not restricted to specifying classes for functors, they can be used to create other sorts of unnamed classes too.
- Also, anonymous classes are of course not restricted to GUIs or Swing, they can be used in other kinds of Java code also.

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).

USER INTERFACES
Swing and multithreading

- Swing uses an additional thread itself:
  the *Event Dispatch Thread* (EDT).
- This means that every Java Swing program automatically creates at least two threads: the *main thread* (=the thread which executes the main method of your program) and the EDT.
- Swing events are handled in the EDT. This means that listeners (event handlers) run in the EDT, and also the actual painting of Swing components on the screen.
- The EDT thread is started when the first window is shown. It continues running until the last window has been closed - even after the main thread has ended!

USER INTERFACES
Swing and multithreading

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

  JFrame f = new JFrame("Hello World");
  f.setSize(300,200); // initial window size

   … // insert sub-components such as panels into the frame

   f.show(); // this "realizes" the frame and all its sub-components
  //the main thread ends here. The EDT starts here.
}
```

USER INTERFACES
Swing and multithreading

- You cannot directly synchronize Swing methods using "`synchronized`" (simply because you cannot change the implementation of Swing). (*)
- So how to create correct multithreaded Swing programs then?
- Most important: Swing's (in)famous "*Single-thread rule*":

  All Swing components and related classes must be accessed <u>by the EDT only</u> once they are "realized" (~displayed), unless there is a documented exception from this rule.

(*) you can of course synchronize your own code which calls Swing code, but much of the Swing processing happens asynchronously and Swing-internally in reaction on events.

USER INTERFACES
Swing and multithreading

- If the "single-thread rule" is not observed, the program becomes prone to race conditions.
- Assume as an example that your program displays a Swing component which contains several data items, such as a combo box.
- If one of your threads modifies an item in the data model of this combo box, Swing automatically updates the box on the screen.
- Repainting the box costs some time. While Swing repaints the box (on the EDT), your thread removes some data items from the data model of the box.
- But the ongoing Swing painting operation does not know about the new, reduced size of the box's data model, and expects more data in the model than it is actually available --> Painting the combo box fails at runtime!

USER INTERFACES
Swing and multithreading

- In the previous example, all event handling and the repainting of the combo box occurred in the EDT. But the modification of the data content of the component was done in another thread, and this caused the problem.
- To avoid such a problem situation<u>, any kind of modification or any other kind of access of already *realized*(!) Swing component should run in the EDT</u> (unless there is an exception from this rule explicitly stated in the Java Swing API documentation).
- Luckily, code called indirectly by Swing itself automatically runs in the EDT (e.g., code you place in JTable's data model methods)
- For running other code, Swing provides two static methods for this: `EventQueue.invokeLater` and `EventQueue.invokeAndWait`.
- These methods place an operation in the form of an event in Swing's *event queue*. The EDT processes subsequently all events in this queue.
- The class of the posted event must implement interface `Runnable` (just like thread task classes, although no new thread is started by `invokeLater/invokeAndWait`)

USER INTERFACES
Swing and multithreading

- Example (using an anonymous class / a functor):

```
EventQueue.invokeLater(new Runnable() { //post an operation to the event queue

    public void run() { // the posted operation

        …  /* here could be, e.g., code which modifies the data shown by some
           Swing component, such as a table or menu */

    }
});
```

- The operation which is placed in the queue (`run()`) is executed by the EDT when it has its turn in the queue. Not necessarily immediately!
- `invokeLater` returns immediately, but this does NOT mean that the placed operation has already been executed, only that it has been placed in the queue.
- `invokeAndWait` returns after the placed operation has actually been executed by the EDT.

USER INTERFACES

Swing and multithreading

- Letting the EDT perform operations has a shortcoming:
  these operations might block the GUI. Mouse clicks and keyboard events are also processed by the EDT, so these are delayed if you place time-consuming operations in the event queue.
- Therefore, let your own thread(-s) do as much of the work as possible. Only actually GUI-related stuff should be placed on the EDT.
- Do not perform extensive computations in operations placed in the event queue using `invokeLater` or `invokeAndWait`.
  Better let your thread do these calculations on some data structure which is unrelated to Swing components and use the EDT only to copy the result into the Swing component's →data model.
- No need to assign or place listeners explicitly in the EDT - listeners are automatically performed in the EDT.

USER INTERFACES
Swing and multithreading

- Rule of thumb:
  Create a new thread for any non-graphical task which could block the GUI for an amount of time that could annoy the user...
  Such as time-consuming mathematical calculations, file handling or network operations.

  This way, the time-consuming task runs in the background while the user can continue working with the GUI and perform other operations of your software.

  But once the result of a thread might influence a GUI component, you must use the EDT to update the GUI or GUI data model.