Advanced GUI structures

- In this section:
  - More advanced interactive widgets
  - Window control
  - More complex layout management

Advanced GUI structures (some in JFS Appendix A)

- Users are accustomed to a sophisticated GUI (Graphical User Interface) in programs
- We have seen "basic" Swing features like:
  - Windows (`JFrames`)
  - Widgets (`JButtons`, `JSliders`, `JLabels`, `JPanels`)
- We will now look at new kinds of more advanced widget:
  - Check boxes (select or not)
  - Combo boxes (choose from a list)
  - Radio button groups (choose one)
  - Menus
- Also
  - Window control
  - More complex layout management

Simple examples

- Check box:

- Combo box:

- Radio button group:

- Menu widgets:

Check Boxes

- A **check box** is a box that the user can select by clicking
  - The check might look like a cross or a tick on different systems:

- An application can either:
  - React to the event when a check box is clicked
  - Or get the status of the check box when some other event occurs, e.g. a button click
- Check boxes and some other widgets require a new kind of event-handling – for *items* rather than *actions*

Check Box event handling

```
import java.event.*;  // Include event-handling

public class Payment extends JFrame
                     implements ItemListener {
  ...
  public void itemStateChanged (ItemEvent event) {
    ...  // Check item selection, and take action
  }
}
```

- `ItemListener` and `itemStateChanged` correspond to `ActionListener` and `actionPerformed`
  - As before: a program may use both, and `ChangeListener`, …

Using Check Boxes

```
private JCheckBox invoice;     // Declare check box

private void createGUI() {
  ...
  invoice = new JCheckBox("Invoice Me");
                               // Create invoice box
  contentPane.add(invoice);      // Add to layout
  invoice.addItemListener(this); // Listen for click
  ...
}

public void itemStateChanged (ItemEvent event) {
  boolean doInvoice;   // Whether invoicing needed
  if (event.getSource() == invoice)
                               // Invoice clicked?
     doInvoice = invoice.isSelected();
                               // get invoicing state
}
```

- This is very like listening for actions with buttons
  - Except that *item* methods are involved
- The program is informed only that an item has changed
  - So it calls `isSelected` to discover the new state (`true`: checked, `false`: not checked)
- Note: *Any* method/event handler can call `isSelected` to inspect the status
- There can be any number of boxes
  - And any number can be checked (or not!)
  - E.g: Married, Homeowner, Smoker, Vegetarian
- Sometimes only *one* box of a group should be selected
  - This requires a button group consisting of *radio buttons:*

  - For example: Single, Married, Divorced, Widowed

Using Radio Buttons

(note: use `ActionListener`/`actionPerformed`)

```
private JRadioButton
        single, married, divorced, widowed;

private void createGUI() {
  ...
  // Create the radio buttons
  single = new JRadioButton("Single", true);
                      // Single is selected initially
  single.addActionListener(this);

  married = new JRadioButton("Married");
  married.addActionListener(this);

  ... divorced ... widowed ...
```

(continued...)

```
    // Group the radio buttons
    ButtonGroup group = new ButtonGroup();
    group.add(single);
    ... married ... divorced ... widowed ...
    contentPane.add(group);      // Add group to layout
    ...
}
// React to a selection event
public void actionPerformed(ActionEvent event) {
    ...
    if (event.getSource() == single) // Clicked?
      ... action ...
    ...
}
```

Or in any method can use, for example:

```
boolean isSingle = single.isSelected();
```


Combo Boxes

- Sometimes the requirement to allow one choice from many is more neatly handled by a *combo box*
  - In particular when the choices need to be handled *dynamically*
  - For example: the choice of a starter in restaurant ordering
- A **combo box** (**choice box** or **drop-down list**) is like a button group in that only one item may be chosen:

Using Combo Boxes

```
private JComboBox payment;

private void createGUI() {
  payment = new JComboBox();          // Create choices
  // Add options
  payment.addItem("Invoice Me");        // Add invoice
  payment.addItem("Phone Me");          // Add phone
  payment.addItem("Fax Me");            // Add fax
  // Add combo box to layout
  contentPane.add(payment);
  // Listen for payment choice being made
  payment.addItemListener(this);
}                                              (continued...)
```

Note: `addActionListener` can also be used


```
// React to a selection event
public void itemStateChanged (ItemEvent event) {
  if (event.getSource() == payment) {  // Choice made?
    // Yes, get which method selected
    String method;            // To note payment method
    method = (String) payment.getSelectedItem();
    ...     // Code to handle the payment
  }
}
```

Note the *cast*: To assure the compiler that it *will* be a `String`

Note: Can also use

```
int method = payment.getSelectedIndex();
```

There are many other `JComboBox` methods

Layout

- Widgets are presented in the window by a **layout manager**
- The simplest is **flow layout**
  - The widgets appear in the *order* that `add` is used
- We have selected `FlowLayout` in `createGUI` like this:

```
Container window = getContentPane();
window.setLayout(new FlowLayout());
```

- Some extra flexibility:

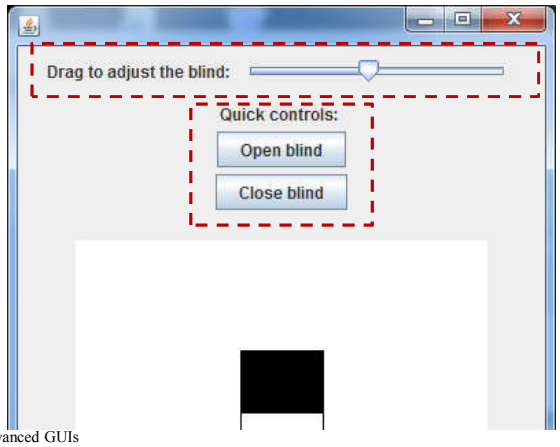
```
new FlowLayout(align, hgap, vgap)
```

  - Alignment is `FlowLayout.LEFT`/`CENTRE`/`RIGHT`
  - with horizontal and vertical gaps (numbers of pixels)
- The problem with flow layout is that there is little control
  - Widgets are positioned sequentially according to the window size

Panels are not only for drawing

- We have used `JPanel`s as drawing canvasses
  - Convenient empty rectangular areas
- **But** they are `Container`s
  - Like the main `JFrame` window
  - *And can have other components added to them*
- Important fact:
  - When a `JPanel` is moved around in a layout
  - … it's contents are *unaffected*
- So we can use `JPanel`s for *grouping*
  - For example, to keep a label and a text field next to each other
- See example on next slide

- The basic idea of grouping:
  - Create a `JPanel` (by default it uses `FlowLayout` *internally)*
  - Add widgets to it
  - Add the panel to the main window layout



- Code for the example:

```
Container window = getContentPane();
window.setLayout(new FlowLayout());  // Main window

// Group the slider and its label
JPanel sliderPanel = new JPanel();   // New panel

sliderLabel = new JLabel("Drag to adjust blind: ");
sliderPanel.add(sliderLabel);        // Add label

slider = new JSlider(JSlider.HORIZONTAL,0,100,50);
sliderPanel.add(slider);             // Add slider

window.add(sliderPanel);     // Add group to window
...
```

- Panels allow even more control:
  - Their dimensions can be set (as we have seen): e.g.

```
panel.setPreferredSize(
                    new Dimension(100, 100));
```

  - Their *internal* layout manager can be set: e.g.

```
JPanel panel =
       new JPanel(new GridLayout(3,2));
```

    Or

```
panel.setLayout(new GridLayout(3,2));
```

- Example: QuickBlind with the label and slider grouped, and the buttons grouped with a label in a *narrow panel*
  - The main window is still flow layout
  - (No `GridLayout` yet)

Demo

WindowBlind with sub panels

Another Simple Layout Manager: `GridLayout`

- Using, for example:

```
container.setLayout(new GridLayout(2, 3));
```

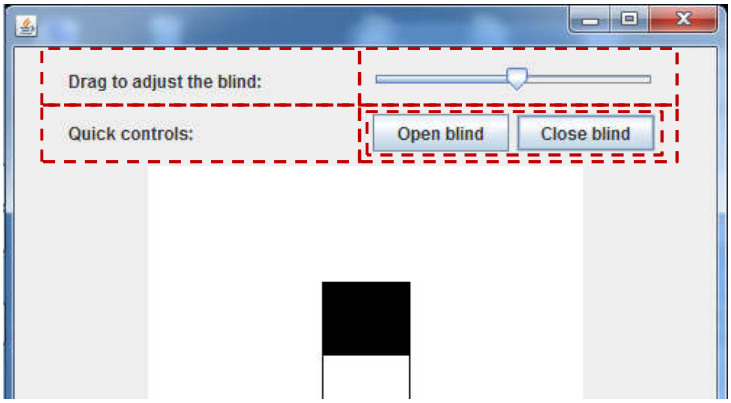
  - Widgets are laid out in a grid 2 rows x 3 cols
  - Added across the columns and row by row
  - All columns are same width – fitting the widest item
  - All rows are same height – fitting the tallest item

1 xxx	2 xxxxxxx	3
4 aaa bbb	5	6 c

- Useful variant:

```
new GridLayout(rows, cols, hgap, vgap)
```

- Another layout for QuickBlind:
  - All the controls in a subpanel with grid layout
  - Labels in left column, controls in right
  - (and the two buttons are grouped into a panel)



- Code for the example:

```
window.setLayout(new FlowLayout());  // Overall

// Panel to group the controls and their labels
JPanel controlPanel = new JPanel(new GridLayout(2,2));

sliderLabel = new JLabel("Drag to adjust blind: ");
controlPanel.add(sliderLabel);

slider = new JSlider(JSlider.HORIZONTAL,0,100,50);
controlPanel.add(slider);

buttonLabel = new JLabel("Quick controls: ");
controlPanel.add(buttonLabel);

(buttons subpanel still to be added…)
```

```
// Now panel to group the buttons
JPanel buttonPanel = new JPanel();

open = new JButton("Open blind");
buttonPanel.add(open);

close = new JButton("Close blind");
buttonPanel.add(close);

controlPanel.add(buttonPanel);   // Add button panel
                                                to controls

window.add(controlPanel);        // Add controls to main

// And finally the drawing panel
panel = new JPanel();
window.add(panel);
```

Demo

WindowBlind with grid layout

Layout

- Overall:
  - Any frame/window and panel can have its own layout manager
  - Any component added to a layout can be a panel - containing further components
- There are more other, more complex but more flexible, layout managers
  - See the Java online reference
  - For example: **`BorderLayout`**, **`BoxLayout`**, **`SpringLayout`**, **`GroupLayout`**, **`GridBagLayout`**, …
- Some IDEs have window builders to help with coding the layout

Windows (`JFrame`s)

- Our GUI applications run in their own window
  - We have seen that an application creates, configures and then makes visible its window
- The user can cause events that affect the window:
  - An application may have to listen for those events
  - …and react accordingly
- A `JFrame` that needs to react to window events must have:

```
implements WindowListener
```

  - And must contain suitable event handler methods

- `WindowListener` event handlers include:

```
public void windowOpened(WindowEvent e)
```

- Called the first time a window is made visible

```
public void windowClosing(WindowEvent e)
```

- Called when the user attempts to close the window from the window's system menu

```
public void windowActivated(WindowEvent e)
```

- Called when the window is set to be the active window

```
public void windowDeiconified(WindowEvent e)
```

- Called when a window is changed from a minimized to a normal state

- The `windowClosing` event can be handled instead of setting the default close operation
  - For example

```
public void windowClosing(WindowEvent event) {
  System.exit(0);
}
```

  - Or, more sophisticated:

```
public void windowClosing(WindowEvent event) {
  int resp = JOptionPane.showConfirmDialog(
                                    "OK to quit?");
  if (resp == JOptionPane.YES_OPTION)
    System.exit(0);
}
```


Menus

- A `JFrame` can have a **menu bar**
- Menus are defined at several levels:
  - A **menu bar** gives the menus available
  - A **menu** gives the menu items available
  - A **menu item** defines a specific menu choice

- All three kinds of information have to be defined, so the code can become lengthy
- When a menu item is selected by the user
  - An `ActionEvent` occurs (like a button)

- Suppose that a payments application needs the following menus:
  - File
    - Open
    - Save
  - Pay
    - Card
    - Cash
- The file menu and its items would be defined as follows:
  - See next slide…


```
JMenu file = new JMenu("File");  // For file menu

JMenuItem open = new JMenuItem("Open");
file.add(open);                  // Add open option
open.addActionListener(this); // Listen for open

JMenuItem save = new JMenuItem("Save");
file.add(save);                  // Add save option
save.addActionListener(this); // Listen for save
```

- The payment menu and its items would be defined similarly:

```
JMenu pay = new JMenu("Pay"); // For pay menu

JMenuItem card = new JMenuItem("Card");
pay.add(card);                  // Add card option
card.addActionListener(this); // Listen for card

JMenuItem cash = new JMenuItem("Cash");
pay.add(cash);                 // Add cash option
cash.addActionListener(this); // Listen for cash
```


- Finally, the complete menu bar would be defined as:

```
JMenuBar menus = new JMenuBar();
menus.add(file);          // Add file menu
menus.add(pay);           // Add pay menu
setJMenuBar(menus); // Add completed menu bar
                                    to frame
```

Menu event handling

- Selection of a menu item causes an action event that is checked as usual:

```
public void actionPerformed (ActionEvent event) {
    if (event.getSource() == open)      // Open?
      ...       // Handle open action
    else if (event.getSource() == save) // Save?
      ...       // Handle save action
    else if (event.getSource() == card) // Card?
      ...       // Handle card action
    else if (event.getSource() == cash) // Cash?
      ...       // Handle cash action
}
```


Menus – other features

- Menus can be *cascaded* (a menu item leads to another menu):

```
file.add(pay); // Cascade pay off file menu
```

- If desired, a menu *accelerator* (*shortcut key*, not in JFS) can be defined for a menu item
  - For example:

```
JMenuItem open = new JMenuItem("Open");
// set Control-O for open item
open.setAccelerator(
    KeyStroke.getKeyStroke(
      KeyEvent.VK_O, ActionEvent.CTRL_MASK));
```

End of section