Creating GUI with JFC/Swing

Creating GUI with JFC/Swing

- Java Foundation Classes (JFC)
- Top-level containers
- Layout managers
- Event-based programming

Java Foundation Classes (JFC)

- JFC is a programming framework used for development of GUI (graphical user interface)-based applications in Java
- It encompasses a group of features for building graphical user interfaces:
  - *Abstract Window Toolkit (AWT)*: original, platform-dependent GUI toolkit
  - *Swing*: the primary, platform-independent GUI toolkit
  - *Java 2D API*: enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and applets

GUI components

- A *GUI component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user
  - Examples include buttons, labels, checkboxes, etc.
- A Java GUI component can be *heavyweight* or *lightweight*
  - A heavyweight components is associated with its own native window
  - A lightweight component is one that "borrows" the screen resource of an ancestor
- All AWT components are heavyweight; all Swing components are lightweight, except for the top-level containers

AWT vs. Swing

- AWT is platform-dependent: its heavyweight components usually act as wrappers around native components
- This means that an AWT application runs fast, but
  - Its components use more resources than their respective Swing counterparts
  - The actual number of offered components is quite small, as it includes only a subset of those available on all target platforms
- Swing is platform-independent, written entirely in Java
  - Only the top-level containers need to allocate native resources
  - Ensures that applications deployed across different platforms have the same appearance
  - Can be used together with AWT
  - Later, significant improvements in the overall speed and native look and feel of the toolkit have been achieved

Swing vs. JavaFX

- JavaFX originated as standalone API, later integrated into JDK 8–10, but standalone again from JDK 11 as an open-source effort (openjfx.io)
- Both libraries based on similar concepts (possibly having different names)
  - Components vs. nodes
  - Containers/panels vs. panes
  - Layouts (slightly different handling: containment vs. inheritance)
  - Event-based programming
- JavaFX aimed at a wider variety of devices
- JavaFX supports special effects, animations
- MVC (model-view-controller) pattern easier to achieve in JavaFX
- In general, JavaFX has cleaner design
- Swing has massive 3rd party support
- Many existing applications have no strong reason to depart from Swing

Creating GUI with JFC/Swing

- Java Foundation Classes (JFC)
- Top-level containers
- Layout managers
- Event-based programming

Top-level containers

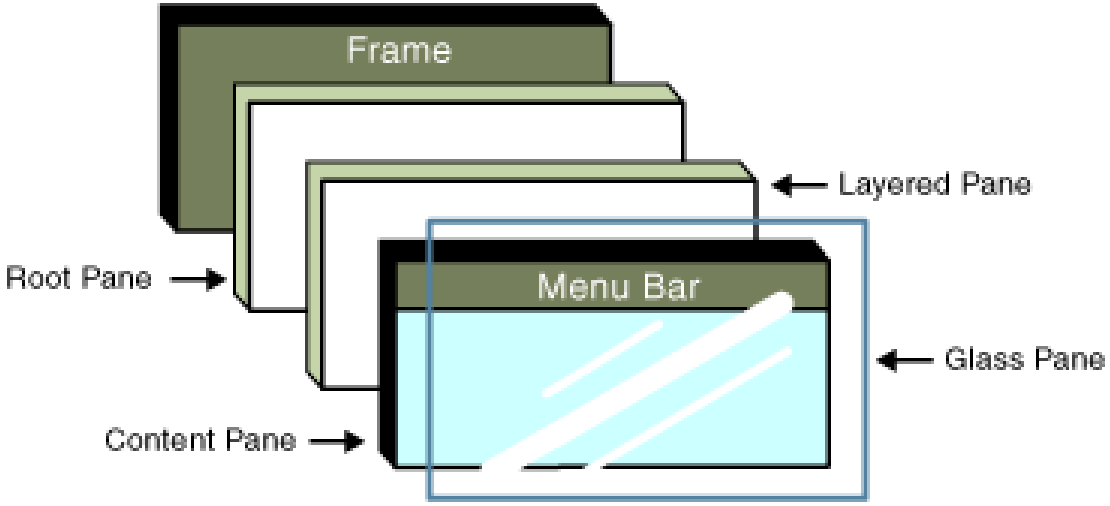
- To appear on screen, every GUI component must belong to a *container*, and be part of a *containment hierarchy*
- A containment hierarchy is a tree of components that has a *top-level container* as its root
- Swing provides three generally useful top-level container classes:
  - *JFrame*, representing windows with titles and borders
  - *JApplet*, representing *applets*, special kind of Java programs that Java-enabled web browsers can download from the internet and run
  - *JDialog*, representing dialogs, e.g. for showing error or warning messages to a user

Containment hierarchies

- A program that uses Swing components has at least one top-level container, representing the root of a containment hierarchy
  - For example, a standalone application that shows a dialog has two containment hierarchies: one with a JFrame as its root, and one with a JDialog as its root
- A GUI component can be contained only once
  - If a component is already in a container and you try to add it to another container, the component will be removed from the first container and then added to the second

Panes

- Every top-level container has several, hierarchically organized *layers*, or *panes*, each serving a different purpose



Panes

- *Root* pane: root of the hierarchy, it contains and controls other panes
- *Layered* pane: a container with *depth* such that overlapping components can appear one on top of the other; it also contains the menu bar, if any
- *Content* pane: the pane that should, as a rule, contain all the non-menu components
- *Glass* pane: on top of all other panes and transparent by default, this pane is used to intercept input messages (such as mouse clicks), to paint over all components, etc.

Creating GUI with JFC/Swing

- Java Foundation Classes (JFC)
- Top-level containers
- Layout managers
- Event-based programming

Layout managers

- Components are added to a container through the container's *add()* method
- The size, position, and alignment of each component within its container is handled by a *layout manager*
- The components can provide size and alignment "hints", but a container's layout manager has the final say on the size and position of the components within the container
- A number of layouts is available in the JFC:
  - BorderLayout
  - CardLayout
  - FlowLayout
  - GridBagLayout
  - GridLayout
  - SpringLayout
  - BoxLayout

Setting up layout managers

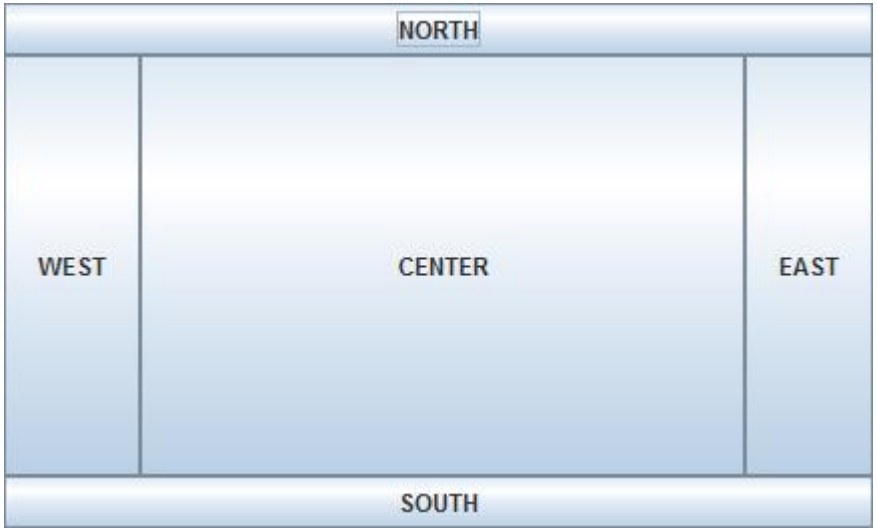
- A layout manager for a container is set as an argument of the container's *setLayout()* method
- Passing the *null* value removes the layout manager, forcing the container to use *absolute positioning*
- With absolute positioning, developers have to manually specify position and size of every component
- Layout managers should be used instead of absolute positioning whenever possible
  - Without a layout manager, components don't respond well to different font sizes, to a container's changing size, and to different locales

BorderLayout

- Default layout manager for content panes of *JFrame* and *JApplet*
- It divides the region into five areas: NORTH, SOUTH, WEST, EAST, and CENTER
- When adding a component, the second argument of the container's *add()* method is the target region (default: CENTER)
- A component is stretched across the entire target region
  - A region can contain no more than a single component

BorderLayout example

```
Container cp = frame.getContentPane();

cp.add(new JButton("NORTH"), BorderLayout.NORTH);

cp.add(new JButton("WEST"), BorderLayout.WEST);

cp.add(new JButton("SOUTH"), BorderLayout.SOUTH);

cp.add(new JButton("EAST"), BorderLayout.EAST);

cp.add(new JButton("CENTER"));
```

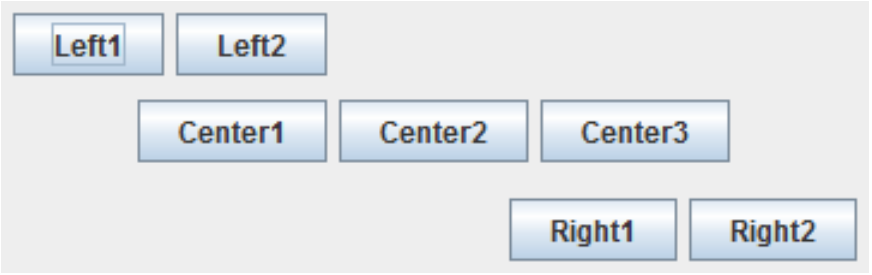


FlowLayout

- Default layout manager for *JPanel*, it puts components in a row, sized at their preferred size
- If the horizontal space in the container is too small to put all the components in one row, the FlowLayout class uses multiple rows
- By default, the row of components is centered vertically within the container
- Method **void setAlignment(int align)**, or the constructor **FlowLayout(int align)** can be used to alter this behavior
  - The alignment parameter can be one of the following: FlowLayout.LEFT, FlowLayout.CENTER, or FlowLayout.RIGHT
- By default, horizontal and vertical gaps between components are set to 5 pixels

FlowLayout example

```
Container cp = getContentPane();
// left-aligned row
JPanel panel = new JPanel();
((FlowLayout)panel.getLayout()).setAlignment(FlowLayout.LEFT);
cp.add(panel, BorderLayout.NORTH);
panel.add(new JButton("Left1"));
panel.add(new JButton("Left2"));
// right-aligned row
panel = new JPanel();
((FlowLayout)panel.getLayout()).setAlignment(FlowLayout.RIGHT);
cp.add(panel, BorderLayout.SOUTH);
panel.add(new JButton("Right1"));
panel.add(new JButton("Right2"));
// centered row
panel = new JPanel();
cp.add(panel, BorderLayout.CENTER);
panel.add(new JButton("Center1"));
panel.add(new JButton("Center2"));
panel.add(new JButton("Center3"));
```



GridLayout

- A GridLayout object places components in a grid of cells
- Each component takes all the available space within its cell, and each cell is exactly the same size
  - The cells are as large as possible, given the space available to the container
- The number of rows and columns is specified in the layout's constructor
- One number, but not both, of rows and columns can be zero, which means that any number of objects can be placed in a row or in a column

GridLayout example

```
// use two columns, and as much rows as needed
GridLayout grid = new GridLayout(0, 2);
Container cp = getContentPane();
cp.setLayout(grid);
for (int i = 0; i < 10; i++)
  cp.add(new JButton("Button" + i));
```

BoxLayout

- Allows multiple components to be laid out either vertically (from top to bottom) or horizontally (from left to right)
- The component layout is specified in the layout's constructor, as one of the following values:
  - X_AXIS: horizontally from left to right
  - Y_AXIS: vertically from top to bottom
  - LINE_AXIS: horizontally; orientation is based on the container's *ComponentOrientation* property
  - PAGE_AXIS: vertically; orientation is based on the container's *ComponentOrientation* property
- By default, components are pressed against one another, so in order to insert spaces, invisible components can be used:
  - *Rigid area*: a fixed-size space between two components
  - *Glue*: controls where the excess space should go; for example, a horizontal glue between two components in a horizontal box will insert any extra space between those components, instead of to the right of all the components

CardLayout

- The CardLayout class manages two or more components that share the same display space
- Conceptually, each component that a CardLayout manages is like a playing card in a stack, where only the top card is visible at any time
- The ordering of cards is determined by the container's own internal ordering of its component objects
  - Usually, the first component that was added is at the top; the last one is at the bottom of the stack

CardLayout methods

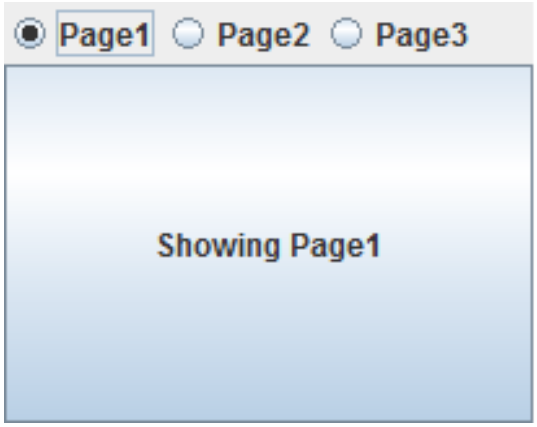
- The cards can be managed using the following methods:
  - **first(Container parent)**: flips to the first card of the container
  - **next(Container parent)**: flips to the next card of the container; if the currently visible card is the last one, this method flips to the first card in the layout
  - **previous(Container parent)**: flips to the previous card of the container; if the currently visible card is the first one, this method flips to the last card in the layout
  - **last(Container parent)**: flips to the last card of the container
  - **show(Conainer parent, String name)**: flips to the component that was added to this layout with the specified name
- When using CardLayout, components have to be added to the container using method **void add(Component c, Object constraints)**, where *constraints* represents unique String name of the component

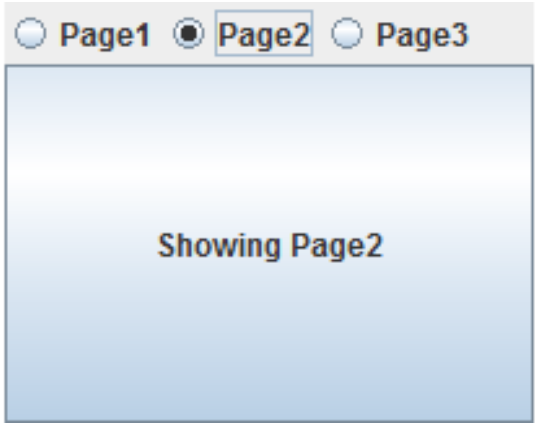
CardLayout example

```
private CardLayout cards;
private JPanel main;
private String[] names = { "Page1", "Page2", "Page3" };

public CardLayoutSample() {
  Container cp = getContentPane();
  // setup CardLayout
  main = new JPanel();
  cp.add(main);
  cards = new CardLayout();
  main.setLayout(cards);
  // add 3 cards (JButtons)
  for (String name : names)
    main.add(new JButton("Showing " + name), name);
  // put 3 radio buttons at the top, for choosing a card
  JPanel top = new JPanel();
  cp.add(top, BorderLayout.NORTH);
  top.setLayout(new BoxLayout(top, BoxLayout.X_AXIS));
  ButtonGroup group = new ButtonGroup();
  for (int i = 0; i < names.length; i++) {
    JRadioButton jr = addRadioButton(top, names[i], i == 0);
    group.add(jr);
  }
}
```

CardLayout example

```
private JRadioButton addRadioButton(JPanel panel,
    String text, boolean selected) {
  JRadioButton jr = new JRadioButton(text);
  panel.add(jr);
  jr.setSelected(selected);
  jr.addActionListener(new ActionListener() {
    @Override public void actionPerformed(ActionEvent e) {
      cards.show(main, ((JRadioButton)e.getSource()).getText());
    }
  });
  return jr;
}
```





GridBagLayout and SpringLayout

- GridBagLayout is a more powerful version of GridLayout
  - It allows cells of different sizes
  - One component can stretch over many cells
- SpringLayout is a very flexible layout manager (added in JDK 1.4) that can emulate many of the features of other layout managers – such as BoxLayout, GridLayout, etc.
- However, these layout managers can be quite complicated for manual coding and should rather be used with a GUI builder

Java GUI development

- Java Foundation Classes (JFC)
- Top-level containers
- Layout managers
- Event-based programming

Event-based programming

- Event-based programming is a programming paradigm in which the flow of the program is determined by *events*, such as user actions (mouse clicks, key presses), messages from other programs, sensory signals, etc.
- In Java, three important components of an event can be identified:
  - *Event source*: component that generates the event (e.g. button, mouse, keyboard, etc.)
  - *Event listener (handler)*: an object that receives news of events and processes them
  - *Event object*: contains all necessary information about the event that has occurred (type, source, etc.)

Java event model

- Any number of event listener objects can listen for all kinds of events from any number of event source objects
- In general, Java event model follows the next algorithm:
  - A listener is registered with a source; once registered, it waits until an event occurs
  - When an event occurs, an event object is first created, and then “fired” by the source to the registered listeners
  - Once the listener receives an event object from the source, it extracts the necessary information and processes the event

Registration of listeners

- Listeners are registered with a source by calling its method in form of:
  - **void add*Type*Listener(*Type*Listener listenerObject)** where *Type* depends on the type of event source: *Key*, *Mouse*, *Focus*, *Component*, *Action*, etc.
- A listener can be unregistered by invoking the following method on the source object:
  - **void remove*Type*Listener(*Type*Listener listenerObject)**
- In the above declarations, *listenerObject* represents the actual listener: an object that implements the appropriate interface, such as *ActionListener*, *MouseListener*, etc.
  - These interfaces contain methods which are invoked when the event occurs

Event objects

- Every event-listener method has a single argument: an object that inherits from the EventObject class
- Although the argument always descends from EventObject, its type is generally specified more precisely
  - For example, the argument for methods that handle mouse events is an instance of MouseEvent
- The EventObject class defines one very useful method:
  - **Object getSource()**: returns the object that fired the event

Action listener

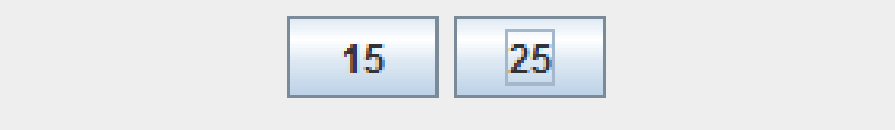
- An action event occurs whenever an action is performed by the user
  - For example, when the user clicks a button, chooses a menu item, presses Enter in a text field
- ActionListener interface:
  - **void actionPerformed(ActionEvent e)**
- ActionEvent class:
  - **String getActionCommand()**: the string associated with this action. Most objects that can fire action events support a method called *setActionCommand()* that lets developers set this string
  - **int getModifiers()**: returns an integer representing the modifier keys the user was pressing when the action event occurred; the following ActionEvent-defined constants can be used to determine which keys were pressed: SHIFT_MASK, CTRL_MASK, META_MASK, and ALT_MASK

Action listener example

```
public class ActionListenerExample
  extends JFrame implements ActionListener {

  public static void main(String[] args) {
    JFrame frame = new ActionListenerExample();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 200);
    frame.setVisible(true);
  }

  public ActionListenerExample() {
    getContentPane().setLayout(new FlowLayout());
    addButton(10);
    addButton(11);
  }

  private void addButton(int number) {
    JButton button = new JButton(Integer.toString(number));
    button.addActionListener(this);
    getContentPane().add(button);
  }
```

Action listener example

```
  @Override
  public void actionPerformed(ActionEvent e) {
    // shift + click increases the number in the source
    if ((e.getModifiers() & ActionEvent.SHIFT_MASK) != 0) {
      JButton source = (JButton)e.getSource();
      int n = Integer.parseInt(source.getText());
      source.setText(Integer.toString(n + 1));
    }
  }
}
```



Mouse listener

- Mouse events notify when the user uses the mouse (or similar input device) to interact with a component
- Mouse events occur when the cursor enters or exits a component's onscreen area, and when the user presses or releases one of the mouse buttons
- MouseListener interface:
  - **mouseClicked(MouseEvent)**: the user has clicked the component
  - **mouseEntered(MouseEvent)**: the cursor has entered the bounds of the component
  - **mouseExited(MouseEvent)**: the cursor has exited the bounds of the component
  - **mousePressed(MouseEvent)**: the user has pressed a mouse button while the cursor is over the component
  - **mouseReleased(MouseEvent)**: the user has released a mouse button after a mouse press over the component

The MouseEvent class

- Parameter of all mouse listener methods; its most commonly used methods are:
  - **int getClickCount()**: the number of consecutive clicks the user has made (e.g. returns 2 for a double click)
  - **int getButton()**: returns which mouse button, if any, has a changed state; one of the following constants is returned: NOBUTTON, BUTTON1, BUTTON2, or BUTTON3
  - **int getX(), int getY(), Point getPoint()**: return the (x,y) position at which the event occurred, relative to the component that fired the event

Adapter classes

- Some listener interfaces, such as MouseListener, contain more than one method
  - But, even if the application is interested only in mouse clicks, all 5 methods of MouseListener have to be implemented
- To make the code easier to develop, read, and maintain, the Swing API includes an *adapter* class for each listener interface with more than one method
- An adapter class implements empty versions of all its interface's methods
  - This enables the developer to override only the necessary set of methods
- (Adapter classes are necessary since Swing is much older than Java 8, otherwise empty default methods in listeners could have been used)

Inner classes

- However, extending an adapter class is not always possible, or desirable
  - A window that needs to respond to mouse clicks cannot extend both JFrame and MouseAdapter classes, since the Java language does not permit multiple inheritance
- Inner classes are used to solve this problem:

```
public class MyClass extends JFrame {
  class MyAdapter extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
      // event listener implementation goes here...
    }
  }

  public MyClass() {
    someObject.addMouseListener(new MyAdapter());
  }
}
```

Anonymous inner classes

- An inner class can be defined without specifying its name; this is known as an *anonymous inner class*
- Anonymous inner classes can make the code easier to read because the class is defined where it is referenced
- An inner class can refer to instance fields and methods of the enclosing class just as if its code is in the containing class (as long as these fields and methods are not static)
  - To refer to the enclosing instance, you can use *EnclosingClass.*this

Anonymous MouseAdapter example

```
public class MouseAdapterExample extends JFrame {

  public MouseAdapterExample() {
    final JLabel label = new JLabel("Click anywhere");
    getContentPane().add(label);
    label.setHorizontalAlignment(SwingConstants.CENTER);
    // use anonymous inner class
    label.addMouseListener(new MouseAdapter() {
      @Override
      public void mouseClicked(MouseEvent e) {
        if ((e.getButton() == MouseEvent.BUTTON1) &&
            (e.getClickCount() == 1)) {
          String msg = String.format("Single left click at (%d, %d)",
            e.getX(), e.getY());
          label.setText(msg);
        }
      }
    });
  }

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

Remaining mouse listeners

- Tracking the cursor's motion involves significantly more system overhead than tracking other mouse events, so mouse-motion events are separated into *MouseMotionListener* type
  - As a convenience, the *MouseInputListener* interface implements both *MouseListener* and *MouseMotionListener* interfaces
- Additionally, mouse-wheel events, which are fired when the wheel on the mouse rotates, are handled by the *MouseWheelListener*
- The API offers two mouse-related adapter classes:
  - *MouseAdapter*: implements all 3 base mouse listeners
  - *MouseInputAdapter*: implementation of the *MouseInputListener*'s methods

Other listener types

- The Swing API offers numerous other listeners, such as:
  - Key listeners: handle key events, which are fired by the component with the keyboard focus when the user presses or releases keyboard keys
  - Component listeners: invoked when the component's size, location, or visibility changes
  - Window listeners: handle window activities or states, such as opening, closing, iconifying/deiconifying, focusing, activating/deactivating, maximizing, etc.
  - Item listeners: available for components that maintain on/off state (check boxes, radio buttons, etc)
  - And many more