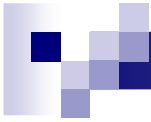




Basics of programming 3

Java GUI and SWING



GUI basics



GUI basics

- Mostly window-based applications
- Typically based on widgets
 - small parts (buttons, scrollbars, etc)
 - built on a windowing framework
- Abundance of frameworks
 - AWT, SWING, SWT, etc
- Logic is to be defined
- GUI builders help



Abstract Windowing Toolkit

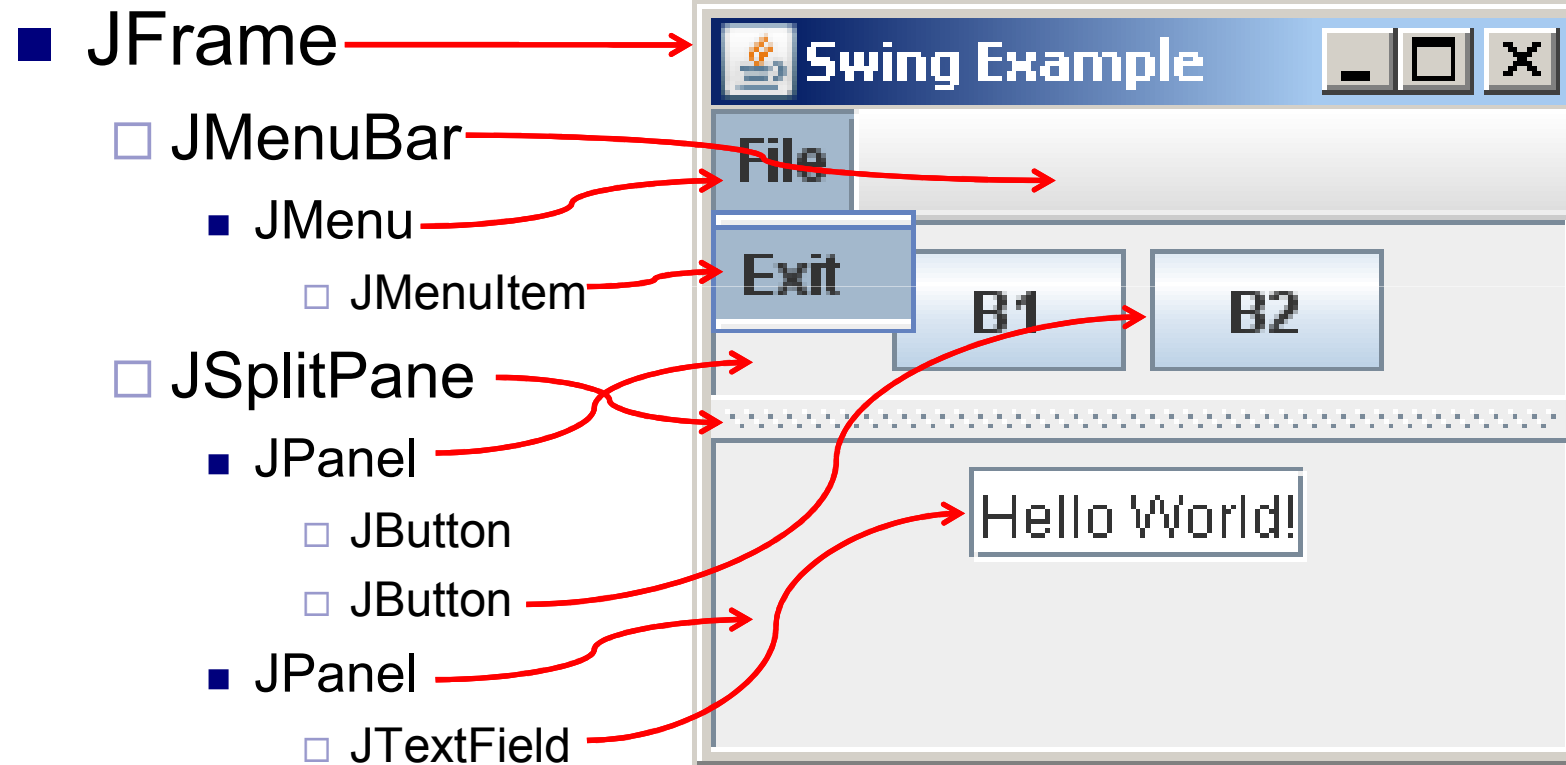
- Heavy-weight widgets
 - relies on OS features
- Abstract facade
 - same look and feel on all platforms
- Different implementations
 - incompatibility issues
- Most features still in use
 - events, layoutmanagement, etc
- `java.awt.*`



SWING

- Rich set of GUI elements
 - light-weight widgets
 - implemented in Java
- Heavy use of model-view-controller pattern
 - complex widgets have separate model classes
 - parts of rendering is implemented separately
- Configurable look and feel
 - via new classes, config files and config parameters
- `javax.swing.*`

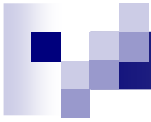
Basic window architecture





Window Application Lifecycle

- Build up window
 - create JFrame or other window
 - put on containers and components
- Register event handlers
 - separate objects for handling user events
- Make it visible
 - window is shown and can be used
- *Use it*
- Close window
 - release resources



Components and containers



Components and containers

- Basic widgets

- JButton
- JTextField
- JPanel
- JFrame

- Complex widgets

- MVC
- JList + JScrollPane



Widget basics

- Rendering is automatic
- Common functionality
 - size (minimum, maximum, preferred)
 - visibility
 - enabled/disabled
 - event handling
- Can be added to a container
- Containers are also widgets
 - component-hierarchy



Simple widgets: JButton

- Classic button
 - can be clicked
 - *event handling later*
- Text and image can be set
- Size is calculated automatically
 - shrinks and grows as the container asks
- Important methods
 - `setEnabled(boolean)`
 - `get/setText()`



Simple widgets: JTextField

- Classic textfield
 - textual input can be entered
- Initial text and size can be set
- Size is calculated automatically if needed
 - shrinks and grows as the container asks
- Important methods
 - `setEditable(boolean)`
 - `get/setText(String)`
 - `setCaretPosition(int)`
 - `setSelectionStart/End(int)`



Simple widgets: JPanel

- Basic container
 - components can be put on it
- Responsible for layout of components
- Size is calculated automatically if needed
 - shrinks and grows as the parent container asks
- Important methods
 - `add(Component[, param])`
 - `setLayout(LayoutManager)`
 - `getComponentAt(int, int)`



Basic window: JFrame

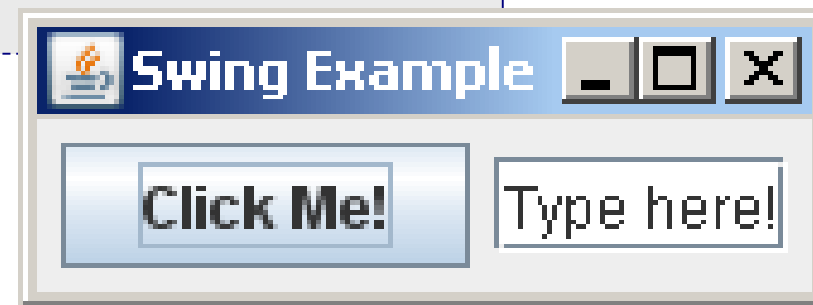
- Basic window with frame and title
- All window operations are supported
 - some has to be explicitly enabled
- Size is calculated automatically
 - based on the size of its contents
- Important methods
 - `add(Component, int where)`
 - `pack()`
 - `setVisible(boolean)`
 - `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`

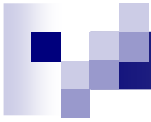
Example: building a GUI

```
JFrame f = new JFrame("Swing Example");
JPanel p = new JPanel();
JButton b = new JButton("Click Me!");
JTextField t = new JTextField("Type here!");

p.add(b);
p.add(t);
f.add(p, BorderLayout.NORTH);

f.pack();
f.setDefaultCloseOperation(f.EXIT_ON_CLOSE);
f.setVisible(true);
```





Event handling



Event handling basics

- Listener (observer) pattern
 - two roles: *listener* and *subject*
 - subject receives event ...
 - ... and forwards it to all registered listeners
 - single *event handler thread* for all events
- Event types for all kinds of events
 - `java.util.EventObject`
 - `java.awt.AWTEvent`
 - `java.awt.event.MouseEvent`
 - `java.awt.event.WindowEvent`
- Events and listeners can be separated
 - responsibility can be dedicated



Event handling interfaces

- *XEvent* → *XListener*
 - interface (implementation is needed)
 - implementation has to be registered at the component
 - `addXListener(XListener e1)`
 - *XEvent* must be processed
 - `MouseEvent`, `KeyEvent`, `AdjustmentEvent`, `FocusEvent` stb.
 - it is usual to use anonymous inner classes
 - looks convenient but is hardly maintainable



Event handling adapters

- `XAdapter`

- default implementation of `XListener`
 - empty methods
- convenience class
 - if only a subset of methods is to be implemented
- use them for a clearer looking code
- consider single inheritance!



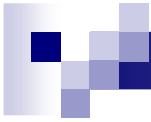
Event handling example

- Modify previous example:
 - when pressing button, put current time in textfield!
- An `ActionListener` implementation is needed
 - new class, take care of visibility
 - *`actionPerformed(ActionEvent ae)`* method
- How to identify the button
 - `ActionEvent`'s *`getSource`* → `Component`
 - `ActionEvent`'s *`getActionCommand`* → `String`
 - name of button or set by *`setActionCommand(String c)`*

Event handling example

```
final class MyActionListener implements ActionListener {
    JTextField t;
    public MyActionListener(JTextField tt) { t = tt;}
    public void actionPerformed(ActionEvent ae) {
        if (ae.getActionCommand().equals("date")) {
            t.setText((new Date()).toString());
        }
    }
}
```

```
...
JButton b = new JButton("Click Me!");
b.setActionCommand("date");
ActionListener al = new MyActionListener(t);
b.addActionListener(al);
...
```



Layout handling



Where are components placed?

■ Problems

- positioning

 - *this page is optimized for 800x600 resolution*

- what happens when resizing

■ Solution: Layout Managers

- every container has a default layout manager

- layout managers can be changed (setLayoutManager)

- responsible for placement

- recursive calculation

- when resizing recalculation occurs



Layout Managers

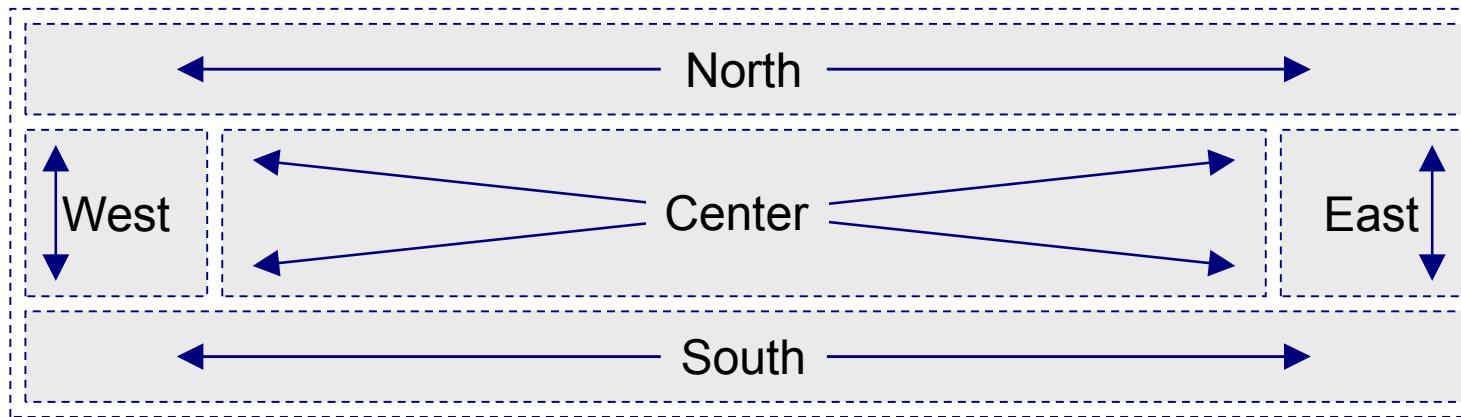
■ Container

- void setLayout(LayoutManager mgr)
- LayoutManager getLayout()
- void validate()
 - recalculates placement and arranges components (recursive)
- Component add(Component comp [*, int index*])
- void add(Component c, Object constraint, int index)
 - adds the component to the container
 - optional parameter specifies special placement demands

■ LayoutManager

- long story...

BorderLayout



- Five fields
 - north, south, west, east, center
- Default layout for JFrames
- Resize in the arrows' direction possible
- One field – one component

FlowLayout



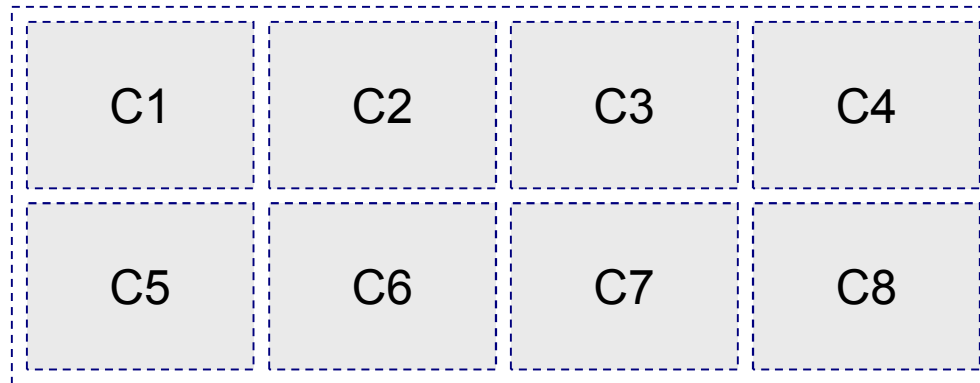
- Default layout for `JPanel`
- Puts components beside each other
 - if there is no more place, starts new row
- Does not resize them individually
- Direction depends on container
 - `ComponentOrientation.LEFT_TO_RIGHT`,
`RIGHT_TO_LEFT`
- Alignment can be set:
 - `LEFT`, `RIGHT`, `CENTER`, `LEADING`, `TRAILING`



CardLayout

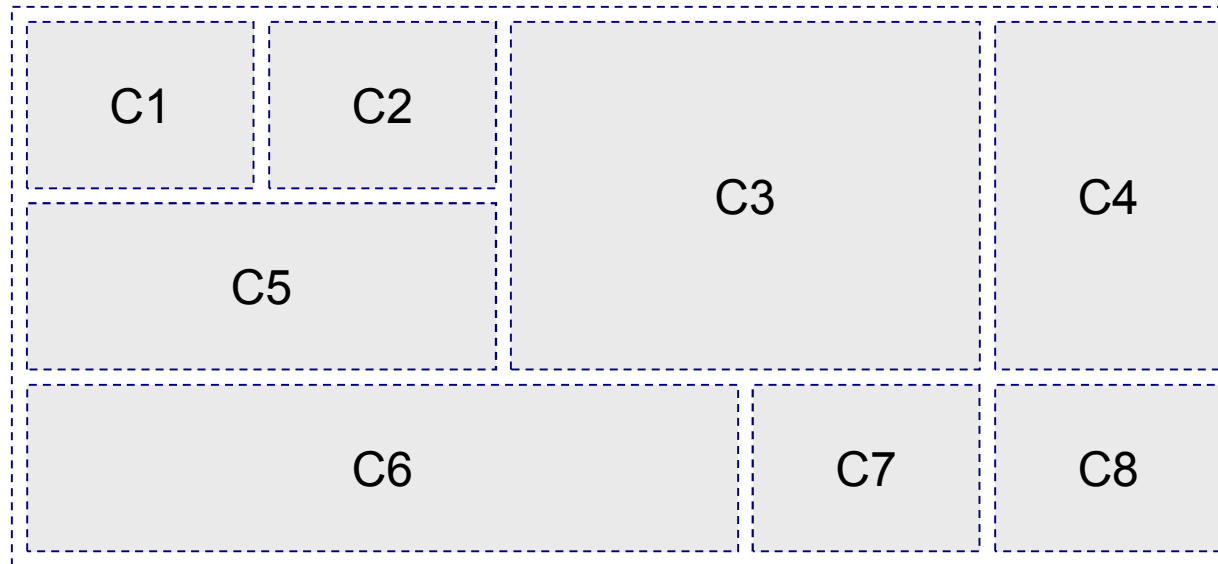
- Components are placed below each other
 - like a deck of cards
 - one component – one card
- Always the topmost is visible
- Deck can be reordered by methods of the managers
- Components can be named for faster access

GridLayout



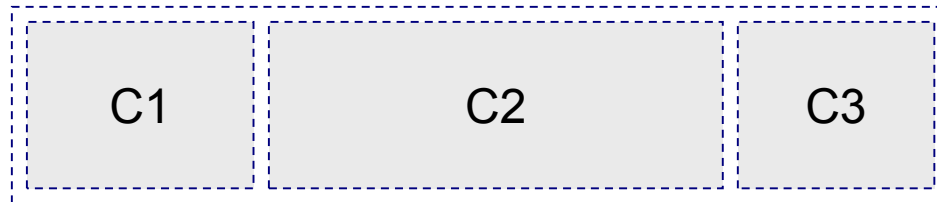
- Components are put into an NxM grid
- Every component has the same size
 - resize if needed
- Direction depends on the container
 - **LEFT_TO_RIGHT** - **RIGHT_TO_LEFT**
- If number of rows is fixed, new columns can be added

GridBagLayout



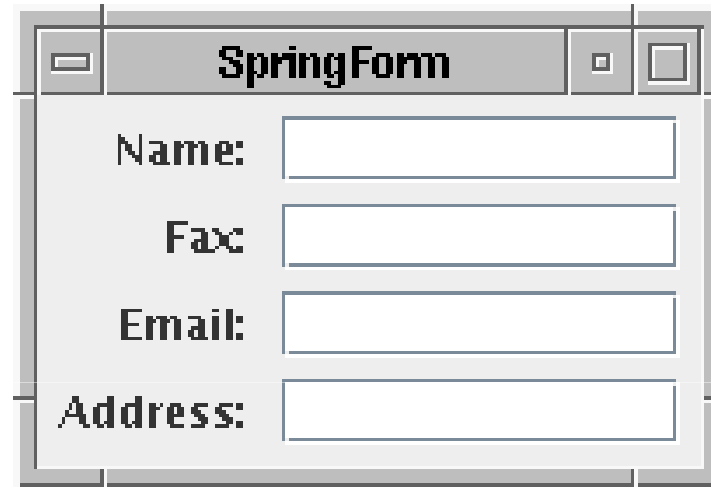
- Advanced variant of *GridLayout*
- Components can occupy more than one square
- *GridBagConstraints* specifies occupancy rules

BoxLayout



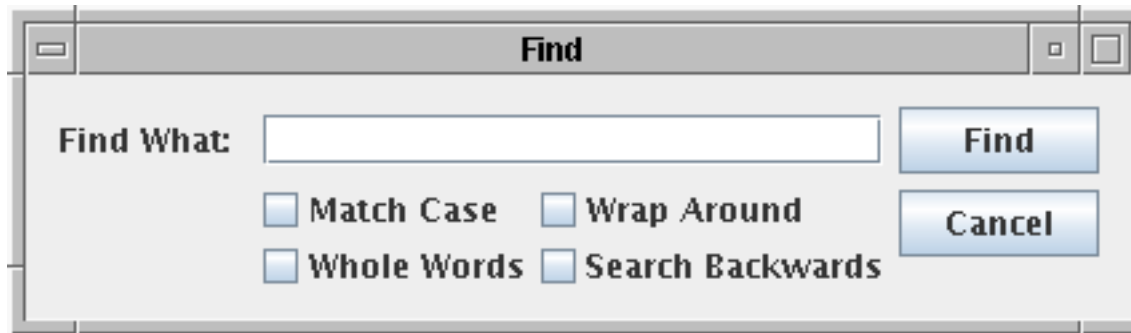
- Vertical or horizontal placement of components
- No new line when full
- Four kinds of directions
 - **X_AXIS – Y_AXIS**: horizontal or vertical
 - **LINE_AXIS – PAGE_AXIS**: considers also **ComponentOrientation**

SpringLayout



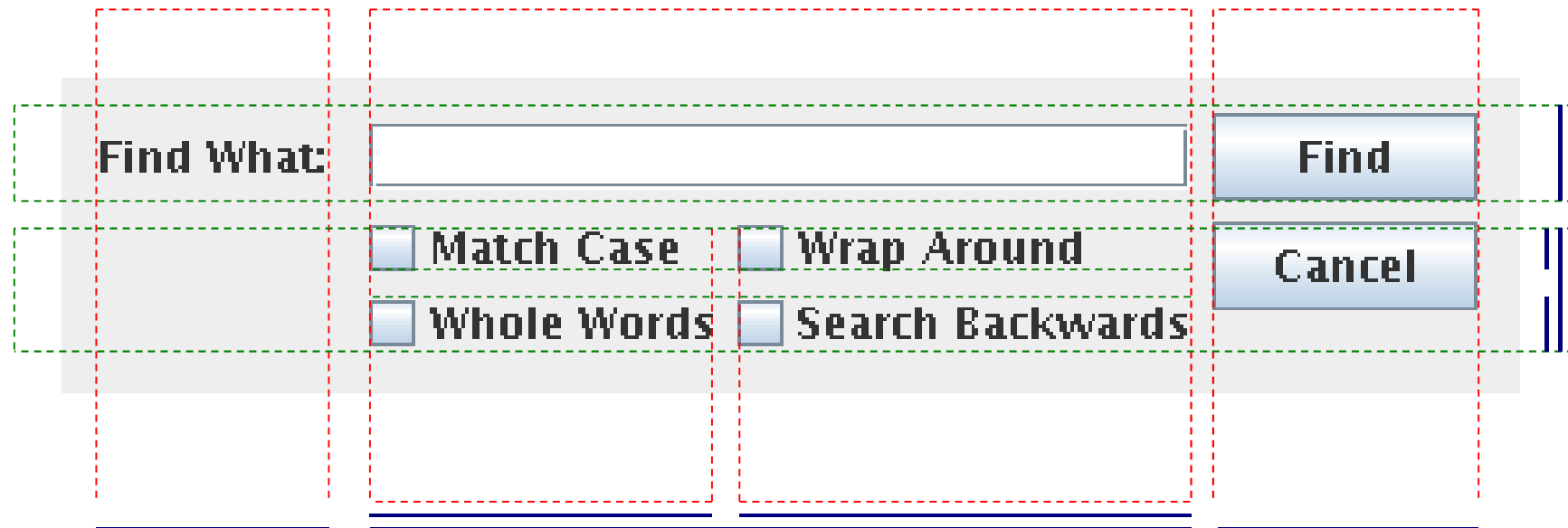
- For flexible table-like layout
- Basically the connection between components' edges must be defined
- Only for GUI builders (designers)
 - hard to program manually

GroupLayout



- Independent specification of horizontal and vertical dimensions
 - each component is added twice
- Hierarchical: groups are placed into each other
- Sequential or parallel placement
- Replacing a component:
 - `void replace(Component oldc, Component newc)`

GroupLayout (swing) 2



----- vertical
----- horizontal



GridLayout (swing) 3

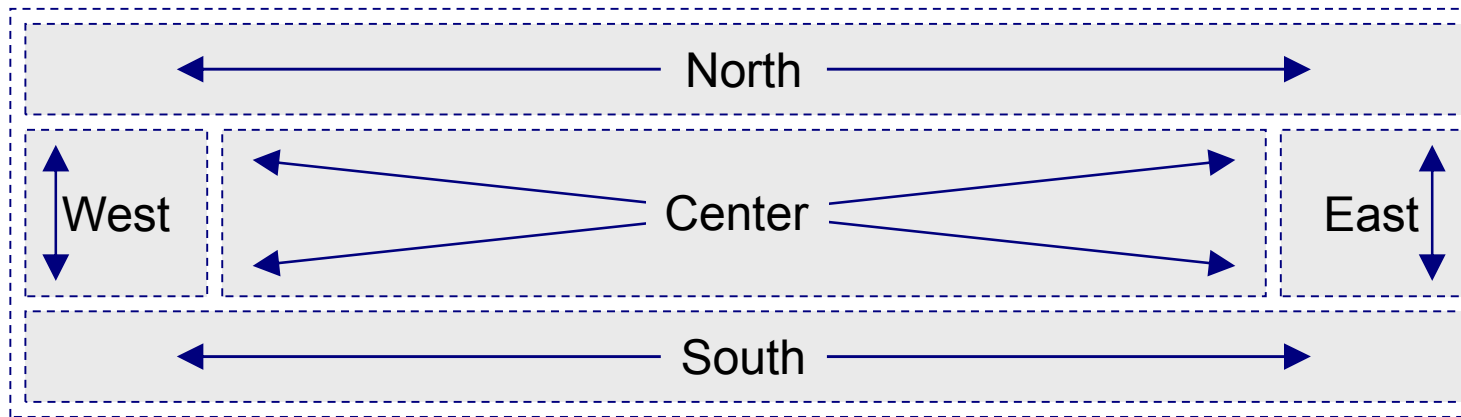
```
layout.setHorizontalGroup(layout.createSequentialGroup()  
    .addComponent(label)  
    .addGroup(layout.createParallelGroup(LEADING)  
        .addComponent(textField)  
        .addGroup(layout.createSequentialGroup()  
            .addGroup(layout.createParallelGroup(LEADING)  
                .addComponent(caseCheckBox)  
                .addComponent(wholeCheckBox))  
            .addGroup(layout.createParallelGroup(LEADING)  
                .addComponent(wrapCheckBox)  
                .addComponent(backCheckBox)))  
    .addGroup(layout.createParallelGroup(LEADING)  
        .addComponent(findButton)  
        .addComponent(cancelButton))  
);
```



GridLayout (swing) 4

```
layout.setVerticalGroup(layout.createSequentialGroup()  
    .addGroup(layout.createParallelGroup(BASELINE)  
        .addComponent(label)  
        .addComponent(textField)  
        .addComponent(findButton))  
    .addGroup(layout.createParallelGroup(LEADING)  
        .addGroup(layout.createSequentialGroup()  
            .addGroup(layout.createParallelGroup(BASELINE)  
                .addComponent(caseCheckBox)  
                .addComponent(wrapCheckBox))  
            .addGroup(layout.createParallelGroup(BASELINE)  
                .addComponent(wholeCheckBox)  
                .addComponent(backCheckBox)))  
        .addComponent(cancelButton))  
);
```

BorderLayout via GroupLayout





Border via Group step1

```
GridLayout layout = new GridLayout(getContentPane());
getContentPane().setLayout(layout);
layout.setAutoCreateGaps(true);
layout.setAutoCreateContainerGaps(true);

layout.setHorizontalGroup
    (layout.createParallelGroup(CENTER)
        .addComponent(north)
        .addGroup(layout.createSequentialGroup()
            .addComponent(west)
            .addComponent(center)
            .addComponent(east)
            )
        .addComponent(south)
    );
```



Border via Group step2

```
layout.setVerticalGroup
    (layout.createSequentialGroup()
        .addComponent(north)
        .addGroup(layout.createParallelGroup(CENTER)
            .addComponent(west)
            .addComponent(center)
            .addComponent(east)
        )
        .addComponent(south)
    );

layout.linkSize(SwingConstants.VERTICAL, north, south);
layout.linkSize(SwingConstants.HORIZONTAL, west, east);
```



Inner classes

- Defined within a class
 - outside or inside methods
 - as a parameter when calling a method (*nasty!!!*)
- No limit on nesting
- Has access to members of nesting class
 - methods
 - *final* parameters and *final* local variables only
- Goal: encapsulation
 - e.g. small helper class in a big one: *Map* – *Map.Entry*

Member class

- has a name
- can be accessed outside of nesting class
 - depending on visibility
- declared in class block (not inside a method)

```
class In1 {  
    int k;  
    class In2 {int x = k;}  
    void bar() {  
        In2 i2 = new In2();  
        k = i2.x++;  
    }  
}
```

```
...  
In1 i1 = new In1();  
i1.k++;  
In1.In2 i3 =  
    new In1().new In2();  
...
```


Local class

- has name
- declared inside a block
 - used in the block

```
public class Test {  
    int i = 10;  
    void xxx() { i++; }  
    void foo(final int a) {  
        class In1 {  
            int k = a;  
            int j = i;  
        }  
    }  
}
```

```
void bar() {  
    k = i++;  
    xxx();  
}  
  
In1 i1 = new In1();  
i1.j++;  
}
```

Combined example

```
public class Test {  
    int i = 10;  
    void xxx() { i++; }  
    void foo(final int a) {  
        class In1 {  
            int k = a;  
            int j = i;  
            class In2 {  
                int x = k;  
                int y = i;  
                int z = a;  
            }  
        }  
    }  
}
```

```
void bar() {  
    In2 i2 = new In2();  
    k = i2.z++;  
    xxx();  
}  
}  
In1 i1 = new In1();  
i1.j++;  
In1.In2 i3 =  
    new In1().new In2();  
}
```

Anonymous class

- never *abstract*, never *static*, always *final*
- has no declared constructor

```
public class MyFrame extends JFrame {
    MyFrame() {
        super("MyFrame");
        addWindowListener(new WindowListener() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
            ... // other methods
        }
    );
}
}
```