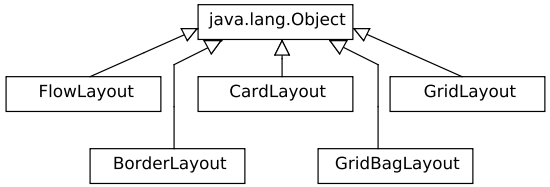
cis3120
design and implementation of software applications I
spring 2015
lecture # II.2: Java GUI API

topics:

- Java GUI API
  - Layout Managers
  - Event Handlers
- *Reference: A Programmers Guide to Java Certification by Mughal and Rasmussen*

Layout Managers. (1)

- Java AWT provides five layout managers:



Layout Managers. (2)

- a *layout manager* describes where the components are laid out within a given container
- each container is associated with a "default" layout manager
- for `Frame` and default is `BorderLayout`
- for `Panel` and `Applet` default is `FlowLayout`
- you can "set" and "get" the layout manager for each container using :
  - `void setLayout(LayoutManger mgr)`
  - `LayoutManger getLayout()`
- you can "nest" containers (and their layour managers)
- even if components use `setSize()`, layout managers may not honor it, they will be treated more like "preferred" size of the components

FlowLayout.

- `FlowLayout` places the components in *row-major* order, growing from left to right and rows top to bottom.
- honors preferred size of the components
- C'tors:
  - `FlowLayout()`
  - `FlowLayout(int alignment)`
  - `FlowLayout(int alignment, int horizontalGap, int verticalGap)`
- `static` data fields to set alignment:
  - `public static final int LEFT`
  - `public static final int CENTER`
  - `public static final int RIGHT`

BorderLayout.

- `BorderLayout` places components in the four compass directions in addition to its center:

	north	
west	center	east
	south	

- only one component can be placed in each region. if you add more than one, only the last one is shown
- not all regions need to be occupied
- C'tors
  - `BorderLayout()`
  - `BorderLayout(int horizontalGap, int verticalGap)`
- components can be explicitly added to one of the regions by using the `contraints` argument (`NORTH, SOUTH, EAST, WEST`) in add method of the container.
- e.g. `add(okButton, BorderLayout.NORTH)`
- will attempt to honor the preferred height of components in NORTH and SOUTH regions and preferred width in WEST and EAST regions

CardLayout.

- `CardLayout` handles containers like a stack of indexed cards
- only the top component is visible and fills the whole region
- C'tors
  - `CardLayout()`
  - `CardLayout(int horizontalGap, int verticalGap)`
- components can be accessed using:
  - `void first(Container parent)`
  - `void next(Container parent)`
  - `void previous(Container parent)`
  - `void last(Container parent)`
- `void show(Container parent, String name)` shows the component with "name"

GridLayout.

- `GridLayout` divides the region of a container into a rectangular grid
- only one component can be placed in each cell
- all the cells in the grid have the same height and width
- ignores the component's preferred size, components are resized to fill the cell
- cell size depends on the container's size and the number of cells
- C'tors
  - `GridLayout()`
  - `GridLayout(int rows, int columns)`
  - `GridLayout(int rows, int columns,`
    `int horizontalGap, int verticalGap)`

GridBagLayout.

- `GridBagLayout` places components in the container in "rows" and "columns"
- you can specify the number of rows and columns
- you can specify the spacing between each row and/or column
- you can specify how a component is placed within its row/column, if it is smaller than the space allocated
- note that the height of an entire row is uniform, even if the components in each column are of different heights
- and the same for the width of a column
- all these are specified using a `GridBagConstraints` object
- component is then added using the `GridBagConstraints` object (`gbc`):
  `add(button, gbc)`

GridBagLayout. (2)

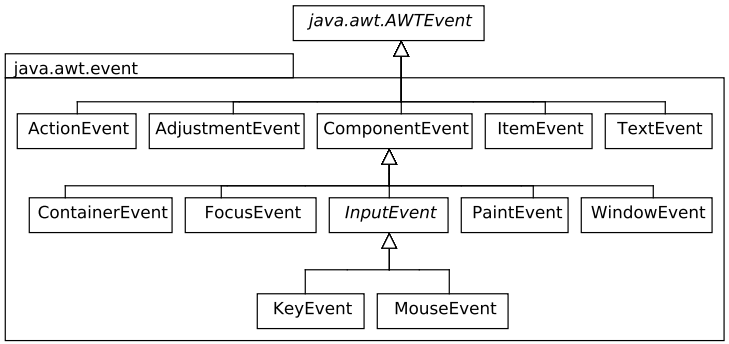
```
GridBagConstraints( int gridx, int gridy, int int,
  gridwidth gridheight, double weightx, double weighty,
  int anchor, int fill, Insets insets, int ipadx, int ipady );
```

- `gridx`, `gridy` specify the location of the component, can be set to RELATIVE
- `gridwidth`, `gridheight` specify how many columns/rows the component occupies. can be set to RELATIVE or REMAINDER
- `weightx`, `weighty` specify how to distribute extra horizontal and vertical space
- `anchor` specifies where to place a component when it is smaller than its display area (e.g., CENTER, NORTH, NORTHEAST, ...)
- `fill` specifies whether to resize a component if it is smaller than its display area (e.g., NONE, HORIZONTAL, VERTICAL, BOTH)
- `insets` specifies minimum amount of space between a component and the edges of its display area (external padding)
- `ipadx`, `ipady` specifies how much space to add to the minimum width and height of the component (internal padding)

Event Handling

- an *event* represents some action on the part of the user
- user-generated events are entered either through the *mouse* or the *keyboard*
- GUI applications are event-driven (mouse clicks, key strokes, moving scrollbar, closing window, etc.)
- principal elements of event handling in Java:
  - event classes encapsulate information about different types of user interaction
  - event source objects inform *event listeners* about events and necessary information about these events
  - event listener objects that are informed by event source objects take appropriate action
- in order to handle events you have to:
  - (1) set event listeners to event sources
  - (2) define appropriate actions in event listeners

Events (1).



Events (2).

- event classes derive from abstract `java.awt.AWTEvent` class package: `java.awt.event`
- `AWTEvent` subclasses can be divided into two groups:
  - Semantic events : high-level (e.g. clicking a button)
    - `ActionEvent` sources : Button, List (double-click), TextField
    - `AdjustmentEvent` sources : Scrollbar
    - `ItemEvent` sources : Checkbox, Choice, List (select/deselect)
    - `TextEvent` sources : TextField (ENTER key)
  - Low-level events : low-level input(e.g. moving mouse) or window operations
    - `KeyEvent` : key press, release or both
    - `MouseEvent` : pressed, release, clicked, dragged, moved, etc.
    - `WindowEvent` : opened, closed, etc.
    - `ComponentEvent` : hidden, shown, moved or resized
    - `ContainerEvent` : handled internally by AWT
    - `PaintEvent` : handled internally by AWT
    - `FocusEvent` : when a component gains or loses focus (can receive keystrokes)

Event Listeners (1).

- *listeners* are interfaces
- for each event type there is a listener interface *XEvent – XListener*
- to register and remove listeners to components use:
  `addXListener()` `removeXListener()`
- then you need to implement each method in the interface
- e.g., for a `KeyListener`, you need:
  - `keyPressed()`
  - `keyTyped()`
  - `keyReleased()`
- the body of a method can be empty, if you don't want to do anything when a given event occurs
- to get the source of the event call `getSource()` method

Event Listeners as Anonymous Inner Classes.

- Anonymous classes allow creating listeners and adding them to event sources:

```
public someMethod() {
   ...
   Button quitButton = new Button("Quit");

   quitButton.addActionListener( new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
         System.exit(0);
      }
   });
   ...
}
```

- the anonymous class *implements* `ActionListener` interface and its method `actionPerformed`
- commonly used in GUI applications if a single listener object will be instantiated

Events Adapters.

- some event listener interfaces have multiple methods (e.g. WindowListener has 7 methods)
- if you only need to specify one or a few of these methods, you still have to define others even if that means leaving their body empty
- java.awt package contains adapters for these interfaces, which implement a listener interface and contain 'blank' definitions for all of its methods (e.g. WindowAdapter)
- you can *extend* an adapter instead of implementing its interface and *override* the method that you need to specify