LAYOUT MANAGERS

- so far controls have been positioned by the default layout manager.
- a layout manager automatically arranges your controls within a window
- it is possible to lay out Java controls by hand, too, you generally won't want to, for two main reasons. First, it is very tedious to manually lay out a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized.
- Each **Container** object has a layout manager associated with it.
- A layout manager is an instance of any class that implements the **LayoutManager** interface.
- The layout manager is set by the **setLayout( )** method.
- If no call to **setLayout( )** is made, then the default layout manager is used.
- The **setLayout( )** method has the following general form:

  void setLayout(LayoutManager *layoutObj*)

- Here, *layoutObj* is a reference to the desired layout manager.
- The layout managers are
  1. FLOW LAYOUT
  2. GRID LAYOUT
  3. BORDER LAYOUT
  4. CARD LAYOUT
  5. GRIDBAG LAYOUT

FlowLayout

- **FlowLayout** is the default layout manager.
- **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor.
- The direction of the layout by default, is left to right, top to bottom.
- components are laid out line-by-line beginning at the upper-left corner.
- In all cases, when a line is filled, layout advances to the next line.
- A small space is left between each component, above and below, as

well as left and right.

- Here are the constructors for **FlowLayout**:

  - FlowLayout( ) --- This form creates the default layout, which centers components and leaves five pixels of space between each component.
  - FlowLayout(int *how*)--------- This form lets you specify how each line is aligned. Valid values for *how* are as follows:

FlowLayout.LEFT	FlowLayout.CENTER
FlowLayout.RIGHT	FlowLayout.LEADING
FlowLayout.TRAILING	

These values specify left, center, right, leading edge, and trailing edge alignment, respectively.

- FlowLayout(int *how*, int *horz*, int *vert*)--- The third constructor allows you to specify the horizontal and vertical space left between components in *horz* and *vert,* respectively.

```
//PROGRAM
import  java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*<applet code="FlowLayoutDemo" width=500 height=600>
</applet>
*/
public class FlowLayoutDemo extends Applet
{
  Button b1,b2,b3;
  public void init()
{
  b1=new Button("1");
  b2=new Button("2");
  b3=new Button("3");
  setLayout(new FlowLayout(FlowLayout.CENTER));
  add(b1);
  add(b2);
  add(b3);
}}
```

BorderLayout

- It has four narrow,fixed-width components at the edges and one large area in the center.
- The four sides are referred to as north,south, east, and west. The middle area is called the center.

Here are the constructors defined by **BorderLayout**:

BorderLayout( )----creates a default border layout
BorderLayout(int *horz*, int *vert*)----

allows you to specify the horizontal and vertical space left between components in *horz* and *vert,* respectively.

BorderLayout defines the following constants that specify the regions:

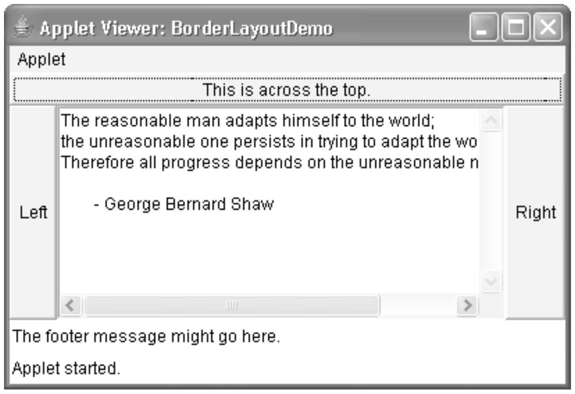
BorderLayout.CENTER	BorderLayout.SOUTH
BorderLayout.EAST	BorderLayout.WEST
BorderLayout.NORTH	

When adding components, you will use these constants with the following form of **add( )**, which is defined by **Container**:

void add(Component *compObj*, Object *region*)

Here, *compObj* is the component to be added, and *region* specifies where the component will be added.

```
// Demonstrate BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;
 /*
<applet code="BorderLayoutDemo" width=400 height=200> </applet>
*/

public class BorderLayoutDemo extends Applet
{ public void init()
{
setLayout(new BorderLayout());


add(new Button("across the top."), BorderLayout.NORTH);
add(new Label("footer message "), BorderLayout.SOUTH);
add(new Button("Right"), BorderLayout.EAST);
add(new Button("Left"), BorderLayout.WEST);

String msg =  "ONE WHO ASKS A QUESTION \n" +"IS A FOOL FOR 5 MINUTES\n" +
"ONE WHO NEVER ASKS A QUESTION\n" + "IS A FOOL FOREVER\n"+

"        - ANONYMOUS \n\n";

add(new TextArea(msg), BorderLayout.CENTER); }
}
```



This is across the top.

The reasonable man adapts himself to the world;
the unreasonable one persists in trying to adapt the wo
Therefore all progress depends on the unreasonable n

- George Bernard Shaw

The footer message might go here.

Applet started.

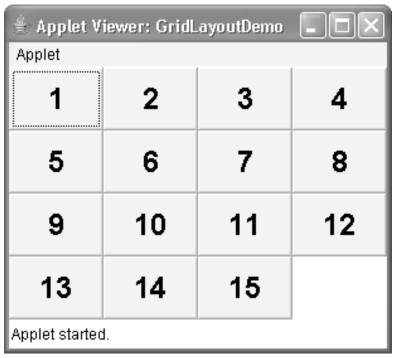
GridLayout

- **GridLayout** lays out components in a two-dimensional grid.
- When you instantiate a **GridLayout**, you define the number of rows and columns.
- The constructors supported by **GridLayout** are shown here:

- GridLayout( )---creates single column grid layout
- GridLayout(int *numRows*, int *numColumns*)---- creates a grid layout with the specified number of rows and columns.
- GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)----This form allows you to specify the horizontal and vertical space left between components in *horz* and *vert,* respectively.
- Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns.
- Specifying *numColumns* as zero allows for unlimited-length rows.

```
//PROGRAM in NOTEPAD
// Demonstrate GridLayout
import java.awt.*;
import java.applet.*;
/*

<applet code="GridLayoutDemo" width=300 height=200> </applet>
*/

public class GridLayoutDemo extends Applet
{ static final int n = 4;
public void init()
```

```
 { setLayout(new GridLayout(n, n));
setFont(new Font("SansSerif", Font.BOLD, 24));
for(int i = 0; i < n; i++)
 { for(int j = 0; j < n; j++)
{
int k = i * n + j; if(k > 0)
add(new Button("" + k)); }
} }
}
```



CardLayout

- The **CardLayout** class is unique among the other layout managers in that it stores several different layouts.
- Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time.

CardLayout provides these two constructors:

- CardLayout( )---creates default card layout
- CardLayout(int *horz*, int *vert*)

This form allows you to specify the horizontal and vertical space left between components in *horz* and *vert,* respectively.

- Use of a card layout requires a bit more work than the other layouts.
- The cards are typically held in an object of type **Panel**.
- This panel must have **CardLayout** selected as its layout manager.
- The cards that form the deck are also typically objects of type **Panel**.
- Thus, you must create a panel that contains the deck and a panel for each card in the deck.

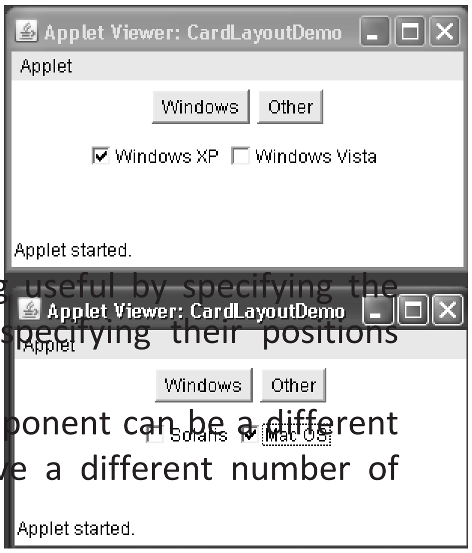
- Next, you add to the appropriate panel the components that form each card.
- You then add these panels to the panel for which **CardLayout** is the layout manager.
- Finally, you add this panel to the window.
- Once these steps are complete, you must provide some way for the user to select between cards.
- One common approach is to include one push button for each card in the deck.
- When card panels are added to a panel, they are usually given a name.
- Thus, most of the time, you will use this form of **add( )** when adding cards to a panel:

➢ void add(Component *panelObj*, Object *name*)

```
// Demonstrate CardLayout.
 import java.awt.*;
import java.awt.event.*;

import java.applet.*;
 /*
<applet code="CardLayoutDemo" width=300 height=100> </applet>
*/

public class CardLayoutDemo extends Applet implements ActionListener,
MouseListener
{

Checkbox winXP, winVista, solaris, mac;
 Panel osCards;
CardLayout cardLO;
 Button Win, Other;

public void init() {

Win = new Button("Windows");
```

```
 Other = new Button("Other");
add(Win);
add(Other);

cardLO = new CardLayout();
osCards = new Panel();
osCards.setLayout(cardLO); // set panel layout to card layout

winXP = new Checkbox("Windows XP", null, true);
winVista = new Checkbox("Windows Vista");
solaris = new Checkbox("Solaris");
mac = new Checkbox("Mac OS");

// add Windows check boxes to a panel
Panel winPan = new Panel();
winPan.add(winXP);
winPan.add(winVista);

// add other OS check boxes to a panel
Panel otherPan = new Panel();
otherPan.add(solaris);
otherPan.add(mac);

// add panels to card deck panel
osCards.add(winPan, "Windows");
osCards.add(otherPan, "Other");

// add cards to main applet panel
 add(osCards);

// register to receive action events
Win.addActionListener(this);
Other.addActionListener(this);

// register mouse events
 addMouseListener(this);
}
```

```
// Cycle through panels.

public void mousePressed(MouseEvent me)
 { cardLO.next(osCards);
}

// Provide empty implementations for the other MouseListener methods.
 public void mouseClicked(MouseEvent me) {
}
public void mouseEntered(MouseEvent me) { }
public void mouseExited(MouseEvent me) { }
public void mouseReleased(MouseEvent me) { }
public void actionPerformed(ActionEvent ae) {
if(ae.getSource() == Win)
 {cardLO.show(osCards, "Windows");
 }
else {cardLO.show(osCards, "Other");
}}}
```



GridBagLayout

- **GridBagLayout** class makes the grid bag useful by specifying the relative placement of components by specifying their positions within cells inside a grid.
- The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number of columns.

- This is why the layout is called a *grid bag.*
- It's a collection of small grids joined together.
- The location and size of each component in a grid bag are determined by a set of constraints linked to it. The constraints are contained in an object of type **GridBagConstraints**.
- Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.
- The general procedure for using a grid bag is to first create a new **GridBagLayout** object and to make it the current layout manager.
- Then, set the constraints that apply to each component that will be added to the grid bag.
- Finally, add the components to the layout manager.

GridBagLayout defines only one constructor, which is shown here:

GridBagLayout( )

GridBagLayout defines one method **setConstraints( )**.

void setConstraints(Component *comp*, GridBagConstraints *cons*)

Here, *comp* is the component for which the constraints specified by *cons* apply. This method sets the constraints that apply to each component in the grid bag.

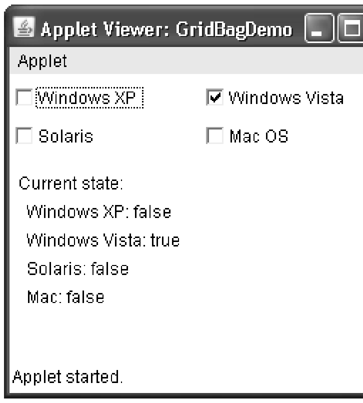
The key to successfully using **GridBagLayout** is the proper setting of the constraints, which are stored in a **GridBagConstraints** object. **GridBagConstraints** defines several fields that you can set to govern the size, placement, and spacing of a component.

Field	Purpose
int anchor	Specifies the location of a component within a cell. The default is GridBagConstraints.CENTER.

int fill	Specifies how a component is resized if the component is smaller than its cell. Valid values are GridBagConstraints.NONE (the default), GridBagConstraints.HORIZONTAL, GridBagConstraints.VERTICAL, GridBagConstraints.BOTH.
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
int gridx	Specifies the X coordinate of the cell to which the component will be added. The default value is GridBagConstraints.RELATIVE.
int gridy	Specifies the Y coordinate of the cell to which the component will be added. The default value is GridBagConstraints.RELATIVE.

Insets insets	Specifies the insets. Default insets are all zero.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0.
int ipady	Specifies extra vertical space that surrounds a component within a cell. The default is 0.
double weightx	Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is
double weighty	Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is

[illegible]



```
Current state:
 Windows XP: false
 Windows Vista: true
 Solaris: false
 Mac: false

Applet started.
```

```
// Use GridBagLayout.
import java.awt.*;
 import java.awt.event.*;
import java.applet.*;
 /*
<applet code="GridBagDemo" width=250 height=200> </applet>
*/

public class GridBagDemo extends Applet implements ItemListener {

String msg = "";

Checkbox winXP, winVista, solaris, mac;

public void init() {

GridBagLayout gbag = new GridBagLayout();
```

```
 GridBagConstraints gbc = new GridBagConstraints();
setLayout(gbag);

// Define check boxes.

winXP = new Checkbox("Windows XP ", null, true);
winVista = new Checkbox("Windows Vista");
solaris = new Checkbox("Solaris");
mac = new Checkbox("Mac OS");

// Define the grid bag.

// Use default row weight of 0 for first row.
gbc.weightx = 1.0;
// use a column weight of 1
gbc.ipadx = 200; // pad by 200 units
gbc.insets = new Insets(4, 4, 0, 0); // inset slightly from top left

gbc.anchor = GridBagConstraints.NORTHEAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
 gbag.setConstraints(winXP, gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(winVista, gbc);

// Give second row a weight of 1. gbc.weighty = 1.0;
gbc.gridwidth                =                GridBagConstraints.RELATIVE;
gbag.setConstraints(solaris, gbc);

gbc.gridwidth                =                GridBagConstraints.REMAINDER;
gbag.setConstraints(mac, gbc);

//   Add   the   components.   add(winXP);   add(winVista);   add(solaris);
add(mac);

//    Register    to    receive    item    events.    winXP.addItemListener(this);
```

```
winVista.addItemListener(this); solaris.addItemListener(this);
mac.addItemListener(this);
}

// Repaint when status of a check box changes. public void
itemStateChanged(ItemEvent ie) {
repaint(); }

// Display current state of the check boxes. public void paint(Graphics g) {
msg = "Current state: "; g.drawString(msg, 6, 80);
msg = " Windows XP: " + winXP.getState(); g.drawString(msg, 6, 100);
msg = " Windows Vista: " + winVista.getState(); g.drawString(msg, 6,
120);
msg = " Solaris: " + solaris.getState(); g.drawString(msg, 6, 140);
msg = " Mac: " + mac.getState(); g.drawString(msg, 6, 160);
} }
```