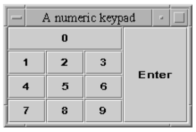
Lecture 1

We review Java's main layout managers — for positioning and sizing components within containers.

We discuss how we might create the following numeric keypad:



Creating the buttons

This code is common to all the programs we look at.

In the constructor:

```
digitB = new JButton[10];
for (int i = 0; i < 10; i++)
{  digitB[i] = new JButton("" + i);
}
enterB = new JButton("Enter");
```

Instance variable declarations:

```
private JButton[] digitB;
private JButton enterB;
```


Absolute positioning I

- Use `null` layout; use `setSize/ setLocation/ setBounds` to position & size components:

  ```
  setLayout(null);

  digitB[0].setBounds(20, 20, 240, 80);
  add(digitB[0]);
  digitB[1].setBounds(20, 100, 80, 80);
  add(digitB[1]);
  etc.
  ```

- Insensitive to container resizing: the keypad will be same size, same position within container
- May not adapt well to other platforms
- Changes of button wording, font, button size, etc. require recalculation and much program revision

Absolute positioning II

- If you're determined to use absolute positioning, used named constants and calculations — for readability & maintainability
- (Take this as far as possible, e.g. calculate widths from font information)

```
setLayout(null);

digitB[0].setBounds(X_MARGIN, Y_MARGIN,
   TREBLE_SIZE, NORMAL_SIZE);
add(digitB[0]);

private static final int NORMAL_SIZE = 80;
private static final int TREBLE_SIZE =
   NORMAL_SIZE * 3;
private static final int X_MARGIN = 20;
private static final int Y_MARGIN = 20;
```


FlowLayout

```
setLayout(new FlowLayout());

add(digitB[0]);

for (int i = 1; i < 10; i++)
{  add(digitB[i]);
}

add(enterB);
```

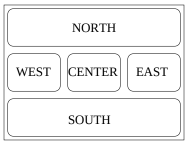
- Components have a preferred size (e.g. for a button: wide enough for its text plus a bit). (You can set preferred sizes.)
- Flow layout places components from left-to-right, starting new rows if necessary; it always uses preferred sizes
- Simple, but little control over positioning; exact arrangement will depend on container width & height

GridLayout

```
setLayout(new GridLayout(4, 3));

add(digitB[0]);

for (int i = 1; i < 10; i++)
{  add(digitB[i]);
}

add(enterB);
```

- Makes the components equal in size (based on the components with widest & highest preferred sizes), then displays them in a grid with the specified number of rows and columns; each component fills its cell in the grid, so the grid always fills its container
- Ideal for grids with equal-sized cells but little else

BorderLayout



- To add, e.g., `enterB` to east:

```
add(enterB, BorderLayout.EAST);
```

- Uses preferred sizes, but. . . if a region is left empty
  - if center is occupied, it takes the empty space
  - if center is empty, east/ west close in &, if east/ west are both also empty, north/ south close in
- If container grows, C gets space, else E/W, else N/S

BorderLayout: using panels

- The restriction to 5 regions is not a problem: the components can themselves be containers (e.g. `JPanel`s), with their own contents and layout
- Using `JPanel` and mixing `BorderLayout`, `GridLayout` and `BoxLayout` gives high flexibility and avoids `GridBagLayout` (see later)

```
setLayout(new BorderLayout());
add(enterB, BorderLayout.EAST);
JPanel digitP = new JPanel();
digitP.setLayout(new BorderLayout());
digitP.add(digitB[0], BorderLayout.NORTH);
JPanel nonZeroDigitP = new JPanel();
nonZeroDigitP.setLayout(new GridLayout(3, 3)
for (int i = 1; i < 10; i++)
{  nonZeroDigitP.add(digitB[i]);
}
digitP.add(nonZeroDigitP, BorderLayout.CENTE
add(digitP, BorderLayout.CENTER);
```


GridBagLayout: for flexibility

- Uses a grid, but components' preferred sizes are used; cells need not be the same size; components may not fill their cells; components may span more than one row or column

```
GridBagLayout gbl = new GridBagLayout();
GridBagConstraints gbc =
   new GridBagConstraints();
setLayout(gbl);

gbc.gridx = 0; // column 0
gbc.gridy = 0; // row 0
gbc.gridwidth = 3; // 3 columns wide
gbc.gridheight = 1; // 1 row high
add(digitB[0], gbc);

gbc.gridx = 3; // column 3
gbc.gridy = 0; // row 0
gbc.gridwidth = 1; // 1 column wide
gbc.gridheigth = 4; // 4 rows high
add(enterB, gbc);
```


Using a helper method

```
addComp(digitB[0], this, gbl, gbc,
   0, 0, 1, 3, 1, 1);
addComp(enterB, this, gbl, gbc,
   0, 3, 4, 1, 1, 1);

// Helper method
private void addComp(Component comp,
   Container cont, GridBagLayout gbl,
   GridBagConstraints gbc, int row,
   int col, int numRows, int numCols,
   double weightx, double weighty)
{  gbc.gridx = col;
   gbc.gridy = row;
   gbc.gridwidth = numCols;
   gbc.gridheight = numRows;
   gbc.weightx = weightx;
   gbc.weighty = weighty;

   cont.add(comp, gbc);
}
```


GridBagConstraint's variables

- `gridx`, `gridy`, `gridwidth`, `gridheight` — row, column, number of rows, number of columns
- `fill` — what to do if cell is bigger than component's preferred size: `NONE` (do nothing), `HORIZONTAL` (expand width only), `VERTICAL` (expand height only) and `BOTH` (But see `weightx/ weighty`)
- `anchor` — when component is smaller than cell, where in cell should it sit; values are `CENTER`, `NORTH`, `NORTHEAST`, `EAST`, etc.
- `weightx` & `weighty` — values 0.0-1.0; how to share spare space; if all are 0.0, space goes outside the grid; if all are 1.0, space is shared equally
- `insets` — the min space between the component and the edge of its cell; `ipadx` & `ipady` — min space within component

Conclusions

- `GridBagLayout` is the most flexible; we can 'easily' lay out forms using it
- Programmers use two other layout managers: `BoxLayout` and `CardLayout`. (You can write your own by implementing Java interface `LayoutManager` or `LayoutManager2`)
- Best to set the layout, but there are defaults: `JFrame` & `JApplet` have `BorderLayout`; `JPanel` has `FlowLayout` and `Box` has `BoxLayout`
- Integrated Development Environments (IDEs) (e.g. Microsoft Visual J++, Symantec Cafe, JBuilder):
  - you place buttons, etc. onto a screen; code is generated for you
  - but mostly use absolute positioning & sizing