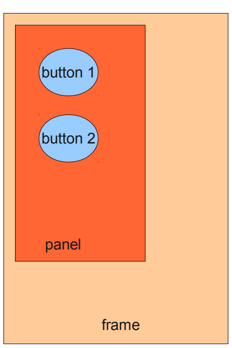
GUIs

- Using Java Swing is easy
  - In terms of creating components, adding them to a panel, listening for events, and so on
- But Swing is also tedious
  - In terms of laying things out
  - The layout managers save us a lot of trouble, but they also make it difficult for us to be precise in how things are arranged

Layout Managers

- A layout manager is associated with a particular component, usually a background component that contains other components
- If a frame contains a panel, and the panel contains a button, the layout manager of the panel controls the size and placement of the button, and the frame controls the size and placement of the panel

Layout Managers



•The frame layout manager has nothing to say about the placement of the buttons. It only has control of the things it directly contains
•The panel's layout manager has control of the buttons.
•Buttons don't need a layout manager, since they won't contain anything else

Layout Managers

Here is a bit of the code to create a frame and a panel with two buttons

```
JFrame myframe = new JFrame();  // make a new JFrame object
final int F_WIDTH = 300;        // 300 pixels wide
final int F_HEIGHT = 400;       // 400 pixels high
myframe.setSize(F_WIDTH, F_HEIGHT);
myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel panelA = new JPanel();
panelA.setBackground(Color.white);
panelA.add(new Button("hi there"));
panelA.add(new Button("bye"));
myframe.getContentPane().add(BorderLayout.EAST, panelA);

myframe.setVisible(true);
```

Layout Managers

Here is a bit of the code to create a frame and a panel with two buttons

```
JFrame myframe = new JFrame();  // make a new JFrame object
final int F_WIDTH = 300;        // 300 pixels wide
final int F_HEIGHT = 400;       // 400 pixels high
myframe.setSize(F_WIDTH, F_HEIGHT);
myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel panelA = new JPanel();
panelA.setBackground(Color.white);
panelA.add(new Button("hi there"));
panelA.add(new Button("bye"));
myframe.getContentPane().add(BorderLayout.EAST, panelA);

myframe.setVisible(true);
```

We'll set the background of the panel to white so we can see its boundaries clearly

Layout Managers

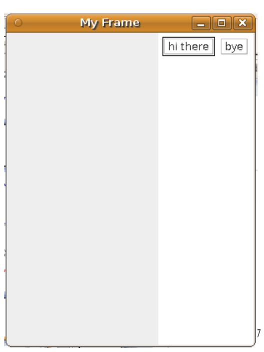
Here is a bit of the code to create a frame and a panel with two buttons

```
JFrame myframe = new JFrame();  // make a new JFrame object
final int F_WIDTH = 300;        // 300 pixels wide
final int F_HEIGHT = 400;       // 400 pixels high
myframe.setSize(F_WIDTH, F_HEIGHT);
myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel panelA = new JPanel();
panelA.setBackground(Color.white);
panelA.add(new Button("hi there"));
panelA.add(new Button("bye"));
myframe.getContentPane().add(BorderLayout.EAST, panelA);

myframe.setVisible(true);
```

We'll add the panel to the EAST portion of the border layout

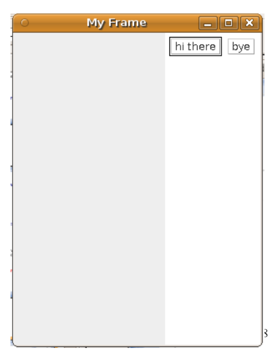
Layout Managers

- The panel is arranged according to the border layout of the frame containing it
- The buttons are arranged according to the flow layout of the panel containing them



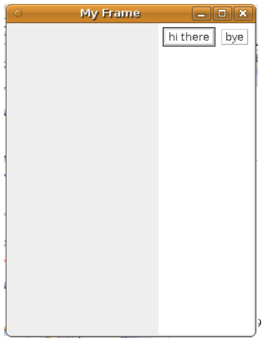
Layout Managers

- Notice that we can click the buttons but that nothing happens.
- Why is that?



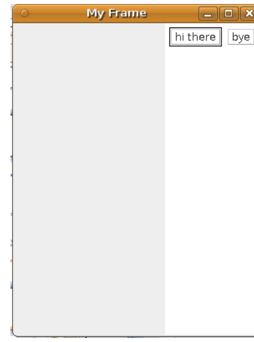
Layout Managers

- Why are the buttons different sizes?
- In this example the buttons have different text field lengths and end up with different sizes



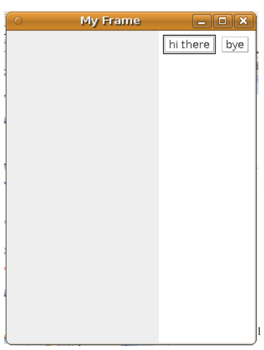
Layout Managers

- Can't we just tell the layout manager how big we want them to be?
- There are methods for setPreferredSize, setMaximumSize and setMinimumSize
- However...



Layout Managers

- Behaviour varies according to layout manager
- Some layout managers will respect your preferences, some will respect part of your preferences, and sometimes a layout manager will not respect your preferences



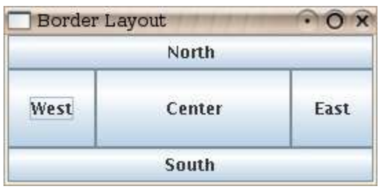
Layout Scenario

- We made a panel and added two buttons to it
- The panel's layout manager asks each button how big that button prefers to be (in this case, the preferred size is based on the amount of text)
- The panel's layout manager uses its layout policies to decide whether it should respect all, part or none of the buttons' preferences
- We added the panel to the frame
- The frame's layout manager asks the panel how big it prefers to be (in this case, the panel needs to be big enough to hold the two buttons)
- The frame's layout manager uses its layout policies to decide whether it should respect all, part or none of the panel's preferences

Border Layout

Border Layout (default for a JFrame's content pane):

- has north, south, east, west, center regions



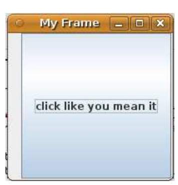
Border Layout

```
import javax.swing.*;
import java.awt.*;

public class GUIStuff {
public static void main(String[] args) {
JFrame myframe = new JFrame(); // make a new JFrame object
myframe.setSize(200, 200);
myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JButton myButton = new JButton("click like you mean it");
myframe.getContentPane().add(BorderLayout.EAST, myButton);

myframe.setVisible(true);
}
}
```

Let's add a button, this time with more text, and add it directly to the frame, in the EAST section

Border Layout

- Because the button has more text, it has a larger preferred size
- Since the button is in the EAST region, the layout manager respects its preferred width
- No matter it's preferred height, it will be as tall as the frame



Border Layout

```
import javax.swing.*;
import java.awt.*;

public class GUIStuff {
public static void main(String[] args) {
JFrame myframe = new JFrame(); // make a new JFrame object
myframe.setSize(200, 200);
myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JButton myButton = new Jbutton("it's lonely at the top");
myframe.getContentPane().add(BorderLayout.NORTH, myButton);

myframe.setVisible(true);
}
}
```

Let's change the code to add the button to the NORTH sector

Border Layout

- Now the button in the NORTH sector gets its desired height
- But it will be as wide as the frame, no matter its preferred width

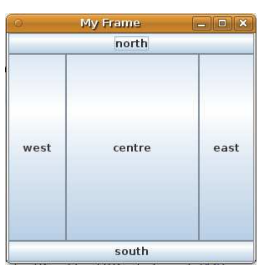
Border Layout

Let's put a button in each sector

```
JButton north = new JButton("north");
JButton east = new JButton("east");
JButton south = new JButton("south");
JButton west = new JButton("west");
JButton centre = new JButton("centre");

myframe.getContentPane().add(BorderLayout.NORTH, north);
myframe.getContentPane().add(BorderLayout.EAST, east);
myframe.getContentPane().add(BorderLayout.WEST, west);
myframe.getContentPane().add(BorderLayout.SOUTH, south);
myframe.getContentPane().add(BorderLayout.CENTER, centre);
myframe.setVisible(true);
```


Border Layout

- Components in the EAST and WEST get their preferred width
- Components in the NORTH and SOUTH get preferred height
- Components in the CENTER get what's left
- Note: components in N and S go all the way across frame, so the things in E and W aren't as tall as they would be otherwise



Flow Layout

- Flow Layout:
  - places components from left to right, with their centres aligned
  - can specify row alignment (LEFT, CENTER, RIGHT)
  - default for JPanel

Flow Layout

- Flow layout is the default layout manager of JPanels
- Let's add an empty JPanel to a JFrame
- Remember, JFrame arranges its components via border layout
- And the JPanel arranges its components via flow layout

Flow Layout

```
JFrame myframe = new JFrame(); // make a new JFrame object

myframe.setSize(200, 200);
myframe.setTitle("My Frame"); // this is optional
myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
myframe.getContentPane().add(BorderLayout.EAST, panel);

myframe.setVisible(true);
}
}
```


Flow Layout

- The panel doesn't contain anything, so it doesn't need much width in the EAST sector
- But let's add something to the panel...

Flow Layout

```
myframe.setSize(200, 200);
myframe.setTitle("My Frame"); // this is optional
myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("shock me");
panel.add(button);
myframe.getContentPane().add(BorderLayout.EAST, panel);

myframe.setVisible(true);
```

Flow Layout

- The panel's flow layout controls the button, and the frame's border layout controls the panel
- The panel expanded because it now contains something
- The button got its preferred size in both dimensions because it is part of the panel now and the panel uses flow layout

Flow Layout

Let's try to add another button below this one

```
Frame myframe = new JFrame(); // make a new JFrame object
myframe.setSize(200, 200);
myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("shock me");
JButton buttonTwo = new JButton("hi");
panel.add(button);
panel.add(buttonTwo);
myframe.getContentPane().add(BorderLayout.EAST, panel);

myframe.setVisible(true);
```


Flow Layout

- Hmm, we wanted the second button below the first, but they got placed side by side
- We can use a different layout manager
- Notice that the “hi” button is smaller. With flow layout, the button gets the size it needs and no more

Box Layout

```
myframe = new JFrame(); // make a new JFrame object
myframe.setSize( 300, 300 );
myframe.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
panel = new JPanel();
panel.setBackground( Color.darkGray );
panel.setLayout( new BoxLayout( panel, BoxLayout.Y_AXIS ) );
button = new JButton( "shock me" );
buttonTwo = new JButton( "hi" );
panel.add( button );
panel.add( buttonTwo );
myframe.getContentPane().add( BorderLayout.EAST, panel );
myframe.setVisible( true );
```

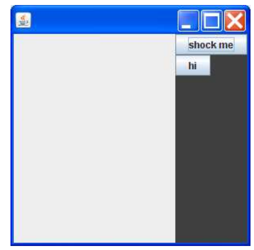
Box Layout

```
myframe = new JFrame(); // make a new JFrame object
myframe.setSize( 300, 300 );
myframe.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
panel = new JPanel();
panel.setBackground( Color.darkGray );
panel.setLayout( new BoxLayout( panel, BoxLayout.Y_AXIS ) );
button = new JButton( "shock me" );
buttonTwo = new JButton( "hi" );
panel.add( button );
panel.add( buttonTwo );
myframe.getContentPane().add( BorderLayout.EAST, panel );
myframe.setVisible( true );
```

The BoxLayout constructor needs to know the component it is laying out, and it needs to know the axis on which it is stacking things (in this case, vertical).

Box Layout

- Now the buttons are stacked
- The panel has shrunk because it doesn't need as much width as before



Components

- That's a bit more information on layout
- Let's look at some other types of components

JTextField

```
JTextField field = new JTextField("Your name");
// create a text field
System.out.println(field.getText());
// get its contents
field.setText("whatever");
// set it to something else
field.setText(""); // clear the field
field.addActionListener(myActionListener);
// register for key events
field.selectAll();
// highlight the text in the field
field.requestFocus();
// put cursor in the field
```

Scrolling Panes

- Often we instead us a JTextArea, which can have more than one line
- We associate a JTextArea with a ScrollPane so that the text will wrap and scroll
- We then add the ScrollPane to the Panel
- And add the Panel to the Frame
- Let's walk through that...

Scroll Pane

```
myframe = new JFrame(); // make a new JFrame object
myframe.setSize( 300, 300 );
myframe.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
panel = new JPanel();
JTextArea text = new JTextArea(10, 20);
text.setLineWrap( true );
```

We create a text area (10 rows by 20 columns) and turn on line wrapping

Scroll Pane

```
myframe = new JFrame(); // make a new JFrame object
myframe.setSize( 300, 300 );
myframe.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
panel = new JPanel();
JTextArea text = new JTextArea(10, 20);
text.setLineWrap( true );
JScrollPane scroller = new JScrollPane(text);

scroller.setVerticalScrollBarPolicy( ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS );

scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
```

We create a JScrollPane object and associate it with the text. We also indicate whether we want it to scroll horizontally or vertically. In this case, we just want vertical scrolling.

Scroll Pane

```
myframe = new JFrame(); // make a new JFrame object
myframe.setSize( 300, 300 );
myframe.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
panel = new JPanel();
JTextArea text = new JTextArea(10, 10);
text.setLineWrap( true );
JScrollPane scroller = new JScrollPane(text);

scroller.setVerticalScrollBarPolicy( ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS );

scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
panel.add(scroller);
myframe.getContentPane().add(panel);
myframe.setVisible(true);
```

We add the scroller to the panel and the panel to the frame.

JTextArea

- What other things can we do with the text area?
- Replace the text that's in it
  - text.setText("now that's something else");
- Append to the text that's in it
  - text.append("and another thing!");
- Select/highlight the text in the field
  - text.selectAll();
- Put the cursor back in the field
  - text.requestFocus();

JTextArea Example

- Here's a longer example

```
public class TextExample implements ActionListener
{
    JFrame myframe;
    JButton button;
    JPanel panel;
    JTextArea text;

    public static void main( String[] args )
    {
        TextExample newDemo = new TextExample();
        newDemo.go();
    }
```



```
public void go()
    {   myframe = new JFrame(); // make a new JFrame object
        myframe.setSize( 300, 300 );
        myframe.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        panel = new JPanel();
        button = new JButton("Just click it");
        button.addActionListener(this);
        text = new JTextArea(10, 20);
        text.setLineWrap( true );
        JScrollPane scroller = new JScrollPane(text);

scroller.setVerticalScrollBarPolicy( ScrollPaneConstants.VERTICAL_SCROLL
BAR_ALWAYS );
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCR
OLLBAR_NEVER);
        panel.add(scroller);
        myframe.getContentPane().add(BorderLayout.CENTER, panel);
        myframe.getContentPane().add(BorderLayout.SOUTH, button);
        myframe.setVisible( true );
    }
```



```
public void go()
    {   myframe = new JFrame(); // make a new JFrame object
        myframe.setSize( 300, 300 );
        myframe.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        panel = new JPanel();
        button = new JButton("Just click it");
        button.addActionListener(this);
        text = new JTextArea(10, 20);
        text.setLineWrap( true );
        JScrollPane scroller = new JScrollPane(text);

scroller.setVerticalScrollBarPolicy( ScrollPaneConstants.VERTICAL_SCROLL
BAR_ALWAYS );
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCR
OLLBAR_NEVER);
        panel.add(scroller);
        myframe.getContentPane().add(BorderLayout.CENTER, panel);
        myframe.getContentPane().add(BorderLayout.SOUTH, button);
        myframe.setVisible( true );
    }
```

Create the text area and its scroll pane.

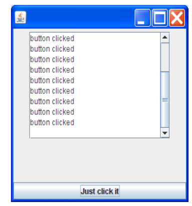
```java
public void go()
    {   myframe = new JFrame(); // make a new JFrame object
        myframe.setSize( 300, 300 );
        myframe.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        panel = new JPanel();
        button = new JButton("Just click it");
        button.addActionListener(this);
        text = new JTextArea(10, 20);
        text.setLineWrap( true );
        JScrollPane scroller = new JScrollPane(text);

scroller.setVerticalScrollBarPolicy( ScrollPaneConstants.VERTICAL_SCROLL
BAR_ALWAYS );
scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCR
OLLBAR_NEVER);
        panel.add(scroller);
        myframe.getContentPane().add(BorderLayout.CENTER, panel);
        myframe.getContentPane().add(BorderLayout.SOUTH, button);
        myframe.setVisible( true );
    }
```

Create a button and associate it with a listener. Notice that we made this class implement a listener type, rather than using an inner class.

The listener

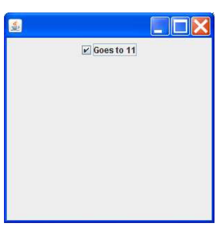
```java
public void actionPerformed(ActionEvent ev)
    {
        text.append("button clicked \n");
    }

}
```

So each time the button is clicked, we append text to the text area.



JCheckBox

- Create a simple check box
- Do something when it is checked or unchecked



JCheckBox

```java
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class CheckBoxTester implements ItemListener
{
    JFrame myframe;
    JCheckBox checker;
    JPanel panel;

    public static void main(String[] args)
    {
        CheckBoxTester cb = new CheckBoxTester();
        cb.go();
    }
```

JCheckBox

```
public void go()
    {
        myframe = new JFrame(); // make a new JFrame object
        myframe.setSize( 300, 300 );
myframe.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        panel = new JPanel();
        checker = new JCheckBox("Goes to 11");
        checker.addItemListener(this);
        panel.add(checker);
        myframe.getContentPane().add(panel);
        myframe.setVisible( true );
    }
```


JCheckBox

```
public void go()
    {
        myframe = new JFrame(); // make a new JFrame object
        myframe.setSize( 300, 300 );
myframe.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        panel = new JPanel();
        checker = new JCheckBox("Goes to 11");
        checker.addItemListener(this);
        panel.add(checker);
        myframe.getContentPane().add(panel);
        myframe.setVisible( true );
    }
```

Create the check box and associate it with an item listener

JCheckBox

```
public void itemStateChanged(ItemEvent ev)
    {
        String onOrOff = "off";
        if (checker.isSelected()) onOrOff = "on";
        System.out.println("Checkbox is "+onOrOff);
    }
```


JCheckBox

```
public void itemStateChanged(ItemEvent ev)
    {
        String onOrOff = "off";
        if (checker.isSelected()) onOrOff = "on";
        System.out.println("Checkbox is "+onOrOff);
    }
```

Now each time somebody clicks the checkbox we will print out whether it is checked or unchecked.

JList

- Sometimes we want to present the user with a list
- The user can select items in the list

JList

```
String[] listEntries = {"giant", "cerevelo", "trek",
"specialized"};
Jlist mylist = new JList(listEntries);
mylist.setVisibleRowCount( 2 );

mylist.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
mylist.addListSelectionListener(this);
```


JList

```
String[] listEntries = {"giant", "cerevelo", "trek",
"specialized"};
Jlist mylist = new JList(listEntries);
mylist.setVisibleRowCount( 2 );

mylist.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
mylist.addListSelectionListener(this);
```

JList constructor takes an array of any object type. It doesn't have to be String, but a String representation will appear in the list.

JList

```
String[] listEntries = {"giant", "cerevelo", "trek",
"specialized"};
Jlist mylist = new JList(listEntries);
mylist.setVisibleRowCount( 2 );

mylist.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
mylist.addListSelectionListener(this);
```

We indicate how many rows should be visible before scrolling starts

JList

```
String[] listEntries = {"giant", "cerevelo", "trek",
"specialized"};
Jlist mylist = new JList(listEntries);
mylist.setVisibleRowCount( 2 );

mylist.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
mylist.addListSelectionListener(this);
```

We indicate that a user can only make a single selection at a time.

JList

```
String[] listEntries = {"giant", "cerevelo", "trek",
"specialized"};
Jlist mylist = new JList(listEntries);
mylist.setVisibleRowCount( 2 );

mylist.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
mylist.addListSelectionListener(this);
```

We associate a listener with the list.

Let's add the list to a scroll pane

```
JScrollPane scroller = new JScrollPane(mylist);

scroller.setVerticalScrollBarPolicy( ScrollPaneConstants.VERTICA
L_SCROLLBAR_ALWAYS );

scroller.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZO
NTAL_SCROLLBAR_NEVER);
panel.add(scroller);
myframe.getContentPane().add(panel);
myframe.setVisible( true );
```


ListSelectionListener

- We'll make our outer class implement ListSelectionListener
- Which means we need to implement this method:

```
public void valueChanged(ListSelectionEvent lse)
    {
        if (!lse.getValueIsAdjusting())
            {
            String selection = (String)
mylist.getSelectedValue();
            System.out.println(selection);
            }
    }
```

ListSelectionListener

- We'll make our outer class implement ListSelectionListener
- Which means we need to implement this method:

```
public void valueChanged(ListSelectionEvent lse)
    {
        if (!lse.getValueIsAdjusting())
        {
            String selection = (String)
mylist.getSelectedValue();
            System.out.println(selection);
        }
    }
```

We just need this check so that we don't get things printed twice

ListSelectionListener

- We'll make our outer class implement ListSelectionListener
- Which means we need to implement this method:

```
public void valueChanged(ListSelectionEvent lse)
    {
        if (!lse.getValueIsAdjusting())
        {
            String selection = (String)
mylist.getSelectedValue();
            System.out.println(selection);
        }
    }
```

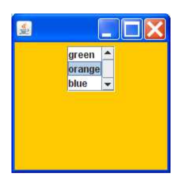
When somebody selects an item from the list, we print it out.
We get the selected value from the event that JList passes when it calls valueChanged(). It passes an Object so we have to cast it to String.

JList

- We can see two items at a time, and have to scroll to see the others
- Users can select one at a time and it will print out the item they selected

In-Class Exercise I

- Write a program to present the user with a list of colors, and when they select a color it changes the panel background to that color
- Start with green, orange and blue



Recursive Methods

- Reading
  - 2nd Ed: Chapter 18
  - 3rd Ed: Chapter 13
- Other Resources
  - •http://www.iol.ie/~jmchugh/csc302/
  - •http://www2.hawaii.edu/~qzhang/ToyProject-TowerOfHanoi.htm

Learning Objectives

- trace code that uses recursion to determine what the code does
- draw a recursion tree corresponding to a recursive method call
- draw a stack trace of code that uses single and multi-branch recursion
- write recursive methods
- replace a recursive implementation of a method with an iterative solution (may need to use a stack to model the run-time stack)

Recursive Methods

- We have seen that a method can make a call to another method (e.g. a method calling a helper method).
- Many programming languages, including Java, allow a method to make a call to itself – we call this *recursion*.
- A method that makes a call to itself is known as a *recursive method*.
- When a method calls itself, it is essentially repeating itself and so recursion is a form of looping.
- Note that in some programming languages, recursion is the *only* way to loop through a block of code.

Recursive Methods

- Some problems are more naturally solved using recursion than a looping construct such as a *for* loop.
- Problems whose solution can be defined in terms of solutions to *smaller* sub-problems have natural recursive solutions.
- There are also some data structures whose *structure* can be defined recursively (a binary tree, for example). These structures can be processed recursively in a very natural way.
- We'll start with some easy examples.

Real-World Examples

- Shampoo bottle instructions
  - Lather
  - Rinse
  - **Repeat**

  repeat all three steps, *including the repeat step*
- An unhelpful dictionary definition
  - Book (n.) - A bunch of pages that make up a **book**
- Neither of these ever terminate – they keep calling themselves

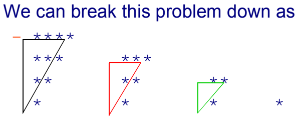
Terminating Conditions

- We need a defined stopping point
  - e.g. “If hair is clean, stop. Otherwise, repeat.”
- Without this, you get infinite recursion, and eventually a memory overload error

Example

- Suppose we want to write a method to draw the following down-triangle (because the pointy end is down) of size 4 on the screen:

  * * * *
  * * *
  * *
  *
- We can break this problem down as follows:



- Hence we define the problem in terms of *smaller* sub-problems.

Example cont'd

- We can therefore write the following *recursive* definition of a triangle of a certain size:

$$\text{drawDownTriangle(size)} = \begin{cases} \text{drawRow(size) } \textit{then} \text{ drawDownTriangle(size - 1)} & \text{if size} > 1 \\ \text{drawRow(1)} & \text{if size} = 1 \end{cases}$$

Example cont'd

- The following method will draw a row of stars:

```
private void drawRow( int size ) {
    for( int count = 0; count < size; count++ )
        System.out.print( '*' );
    System.out.println();
}
```

- This method will be used as a helper to draw our triangle.

Example cont'd

- The following method uses the recursive definition given earlier to draw a down-triangle of a certain size:

```
public void drawDownTriangle( int size ) {
    if( size == 1 )
      drawRow( size );
    else {
      drawRow( size );
      drawDownTriangle( size - 1 );
    }
}
```

Base case

Recursive case

Recursive call

Example cont'd

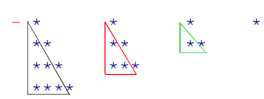
- Trace the following method call: `drawDownTriangle( 4 );`

Example

- Suppose we want to write a method to draw the following up-triangle (up because the pointy end is up) of size 4 on the screen:

```
*
**
***
****
```

- We can break this problem down as follows:



- Hence we define the problem in terms of *smaller* sub-problems.

Example cont'd

- We can therefore write the following *recursive* definition of a triangle of a certain size:

$$\text{drawUpTriangle(size)} = \begin{cases} \text{drawUpTriangle(size - 1) } \textit{then} \text{ drawRow(size)} & \text{if size} > 1 \\ \text{drawRow(1)} & \text{if size} = 1 \end{cases}$$

Example cont'd

- The following method uses the recursive definition on the previous slide to draw a triangle of a certain size:

```
public void drawUpTriangle( int size ) {
    if( size == 1 )
       drawRow( size );
    else {
      drawUpTriangle( size - 1 );
      drawRow( size );
    }
}
```

Base case

Recursive case

Recursive call

Example cont'd

- Trace the following method call: `drawUpTriangle( 4 );`

Example

- Suppose we want to write a method to draw the following ramp of size 3 on the screen:

```
***
**
*
**
***
```

- We can define this problem in terms of smaller ones as follows:

```
– ***
   **        **
   *         *        *
   **        **
   ***
```

Example cont'd

- We can therefore write the following *recursive* definition of a ramp of a certain size:

$$\text{drawRamp(size)} = \begin{cases} \text{drawRow(size) } \textit{then} \text{ drawRamp(size - 1) } \textit{then} \text{ drawRow(size)} & \text{if size} > 1 \\ \text{drawRow(1)} & \text{if size} = 1 \end{cases}$$

Example cont'd

- The following method uses the recursive definition on the previous slide to draw a triangle of a certain size:

```
public void drawRamp( int size ) {
    if( size == 1 )
       drawRow( size );
    else {
       drawRow( size );
       drawRamp( size - 1 );
       drawRow( size );
    }
}
```

Base case

Recursive case

Recursive call

Example cont'd

- Trace the following method call: `drawRamp( 3 );`

Recursive Method Calls – General Form

- Our `drawRamp` method illustrates the general form of a recursive method call:

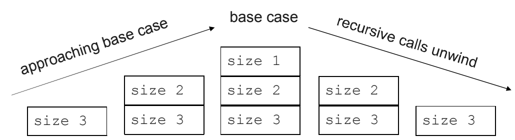
```
type recursiveMethod( type param1, type param2,… )
{
    if( base case )
       // handle base case (code omitted)
    else
    {
         // operations to do before recursive call
         // (code omitted)
         recursiveMethod( … );  // recursive call
         // operations to do after recursive call
         // (code omitted)
    }
}
```

Recursion and Stacks

- There is a strong connection between recursion and stacks.
- Recall that the compiler uses a stack, called the *run-time* stack, in which to store data that is local to each method that is called.
- When a method is called, a new stack frame is generated and pushed onto the run-time stack. This frame stores, among other things, parameters and local variables for the method.
- When the method ends, its stack frame is popped off the run-time stack.

Recursion and Stacks cont'd

- The following is a trace of the run-time stack corresponding to the method call `drawRamp( 3 )`. Note that we show only the parameter `size` in each stack frame.



Recursion and Stacks cont'd

- The trace of the run-time stack gives us an insight into the *space complexity* of the method call. We observe that the method call `drawRamp( N )` will result in the run-time stack holding, at any given point, a maximum of N stack frames corresponding to the method `drawRamp`. Hence the `drawRamp` method has O(N) space complexity.

- Compare this to an iterative solution where we use a loop to draw the ramp in a *single* call to the method `drawRamp`. This implementation has O(1) space complexity.

Recursion and Stacks cont'd

- *Any* recursive method can be converted to an iterative method if we make a stack available. (Note that the use of a stack isn't always necessary.)

- We will now rewrite our `drawRamp` method as an iterative method that uses a stack to mimic the run-time stack maintained by the compiler.

Recursion and Stacks cont'd

```
public void drawRamp( int size ) {
   Stack<Integer> sizeStack = new Stack<Integer>();
   // head towards the base case
   while( size > 1 ) {
      sizeStack.push( size );
      drawRow( size );
      size--;
   }
   drawRow( 1 ); // corresponds to base case
   // unwind the stack
   while( !sizeStack.isEmpty() ) {
      size = sizeStack.peek();
      drawRow( size );
      sizeStack.pop();
   }
}
```


Computing a factorial

- The factorial of a non-negative integer is defined as follows:
  n! = 1 if n=0
  n! = n*(n-1)*(n-2)…*3*2*1 if n>0
- Examples:
  5! = 5*4*3*2*1 = 120
- Can we replace the definition above with a recursive definition?

Example: computing a factorial

- The factorial of a non-negative integer can be *recursively* defined as follows:
- Corresponding recursive method:

```
public int factorial(int n) {
```


Example cont'd

- Trace the following method call:

```
int result = factorial( 3 );
```

Recursive Methods - Checkpoint

- How can you check that a recursive method is correct?
  - check that the base case(s) is (are) correct
  - assuming that the recursive call(s) will return the right answer(s) for the smaller problem(s), show that the recursive step will return the right answer to the original problem
  - make sure that the terminating condition will eventually become true and the recursion will terminate – each recursive step should take you one step closer to reaching the base case
- This is a form of mathematical induction (see CPSC 121)

Example: Fibonacci Numbers

- The Fibonacci sequence is generated as follows:
  - the first two numbers in the sequence are 1
  - all other numbers are generated by adding the previous two numbers

    1, 1, 2, 3, 5, 8, 13, 21, 34, …

  - The following web site contains some interesting facts about Fibonacci and his sequence of numbers: http://plus.maths.org/issue3/fibonacci/

Example: Fibonacci Numbers

- The description of the Fibonacci sequence on the previous page lends itself to the following recursive definition of the Nth Fibonacci number:

$$Fib(N) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ Fib(N-1) + Fib(N-2) & \text{if } n > 2 \end{cases}$$

- Note that the recursive step involves *two* recursive calls to the method – we call this *multi-branch recursion*.

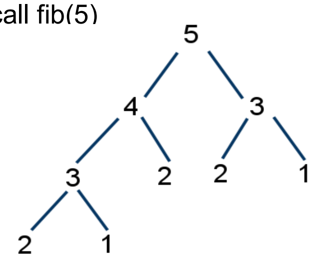
Example: Fibonacci Numbers

- The corresponding recursive method:

```
public int fib( int n ) {
   if( n == 1 || n == 2 )
      return 1;
   else
      return fib( n - 1 ) + fib( n - 2 );
}
```

Recursion Tree

- Let's call fib(5)



- Notice anything inefficient about this?

Recursion and Memory

- Each subroutine invocation creates an **activation record** that holds the values of arguments and local variables for that invocation. These activation records are stored in an area of memory called the **run-time stack**. The diagram below shows the state of the run-time stack when a call to fib(4) is made from main.

			fib(2)		fib(1)					
		fib(3)	fib(3)	fib(3)	fib(3)	fib(3)		fib(2)		
	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	
main()	main()	main()	main()	main()	main()	main()	main()	main()	main()	main()

Recursion and Memory

- By examining the height of the recursion tree we can determine the maximum number of fib activation records on the run-time stack at any given time.
- Memory for the runtime stack is limited. If we attempt to generate more activation records than can be stored on the run-time stack, a stack overflow occurs and the program will crash.
- The height of the recursion tree is useful for telling us how much memory will be required when the function is called

Recursion and Time

- The total number of nodes in the recursion tree is the number of function calls, which can help to determine the amount of time it will take to execute the function.
- The recursive solution of fibonacci uses much more memory and takes longer when done recursively, than iteratively.
- Recursion might seem to work fine for small numbers, but try running fib(40) or higher

Recursion vs Iteration

- Many recursive functions can be easily defined iteratively without using a stack:

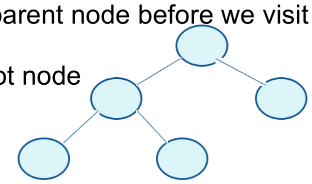
```
int fact(int n) {
  if (n == 0) {
    return 1;
  }

  return n*fact(n-1);
}
```

→

```
int fact(int n) {
   int result = 1;
   while (n > 0) {
      result *= n;
      n = n-1;
   }
   return result;
}
```

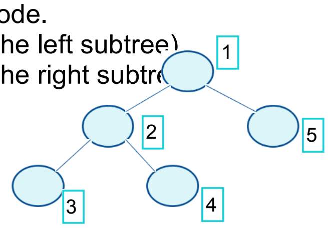

Example: Tree-Traversal

- We have a binary-branching tree structure, and we want to visit each node in the tree exactly once
- Additionally, we have the constraint that we want to visit a parent node before we visit its children
- Begin at the root node



Tree Traversal

- Our traversal instructions, beginning at root node
- Traverse(tree)
  - Visit the node.
  - Traverse(the left subtree)
  - Traverse(the right subtree)



Recursion vs Iteration (cont'd)

- Recursion usually requires more memory than iteration
  - each method call creates a new stack frame in which its parameters and local variables are stored
- Sometimes recursion is more natural so it may take more time to develop an iterative solution.
- Rule of thumb:
  - use iteration when it is easy and natural to do so.
  - use recursion when it is easy and natural to do so.

Tail Recursion

- **Tail Recursion** is when the last line of the subroutine is the recursive call
- Thus there are no deferred operations after the last recursive call
  - unlike our factorial and fibonacci examples
- Many compilers automatically convert tail recursion problems to iterative problems

Indirect Recursion

- Recursion needn't involve a subroutine directly calling itself
- For example, Function A calls Function B which calls Function C, and Function C calls Function A

Conclusion

- Recursion can add simplicity, elegance and readability to a program
- Not always the most efficient method
- Check whether you could solve the problem more efficiently in an iterative fashion
- Check whether your problem naturally lends itself to being solved by solving a number of subproblems
  - e.g. Tree traversal

In-Class Exercise II

- We know how to write a method to take an ArrayList<String> and print out each item using a for-loop or an iterator
- Write a recursive method that does the same thing
  - What is your base case?
  - How do you get closer to your base case?

Learning Goals Review

- trace code that uses recursion to determine what the code does
- draw a recursion tree corresponding to a recursive method call
- draw a stack trace of code that uses single and multi-branch recursion
- write recursive methods
- replace a recursive implementation of a method with an iterative solution (may need to use a stack to model the run-time stack)