MORE GUI COMPONENTS

Today

- Last time we looked at some of the components that the:
  - AWT
  - Swing

  libraries provide for building GUIs in Java.
- This time we will look at several more GUI component topics.
- We will also look at:
  - Exception handling

  a topic that isn't directly related to GUIs, but which we need to cover.

BorderTest

- The next few slides refer to the `BorderTest` program that you can download from the class website.

- The `BorderLayout` class allows five things to be arranged in a container:
  - north
  - south
  - east
  - west

  on the borders of the container, and in the:
  - center
- `BorderLayout` doesn't need to be explicitly named, because it is the default.
- Note that `add()` has two parameters.

- If you add a component to `BorderLayout` without specifying what position you want it in, it is positioned in the center.

HTML labels

- The labels in buttons have to be strings, but they can also be strings that implement HTML commands.
- For example:

```
String s1 = "<html><table>" +
            "<tr><td>One</td><tr>Two</td></tr>" +
            "<tr><td>Three</td><tr>Four</td></tr>" +
            "</table></html>"
JButton button = new JButton(s1);
```

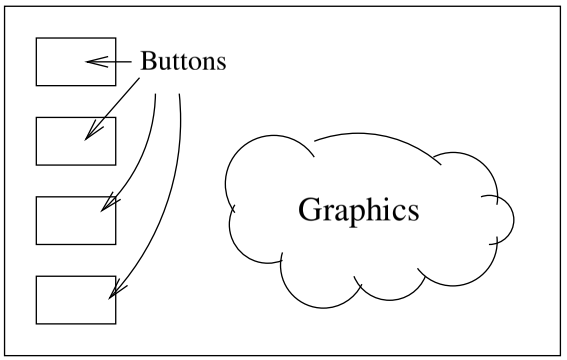
  will display the table in the button.

- HTML can be used in this way to embed graphics in buttons.

Nesting Containers

- The next few slides refer to the `NestContain` program that you can download from the class website.

- Here we want an interface that looks like:



- We can do this by nesting one container in another.

- We put the buttons in a `GridLayout` in one container.
- We then put this container in a second container along with the graphics.
- We use `Borderlayout` for ths outer container.
  - Make the container with the buttons the `WEST` component.
  - Make the graphics the `CENTER` component.

Resize

- The next few slides refer to the `Resize` program that you can download from the class website.

- If you run `NestContain` you will notice that if you resize the window, the buttons behave as you would like:
  - They change size as the window grows and shrinks.
- The graphics component, however, does not change in size.
- This is because we fixed its size with RADIUS.
- `RESIZE` shows a trick that allows us to resize graphics.

- It hinges on the following function:

```
public void setBounds(int x, int y,
      int width, int height){
   radius = Math.min(width, height) / 2;
   super.setBounds(x, y, width, height);
   repaint();
}
```

- This over-rides the `setBounds()` function but calls the superclass version.
- This does not interfere with what setBounds does, but allows us access to its parameters.

- `setBounds()` is called by the layout manager to tell a container where it is on the screen and how big it is.
- `x` and `y` are the location within the screen.
- `width` and `height` are the window dimensions.
- Here we use them to set the size of the star.

Menus

- The main GUI component that we have not yet covered is the menu.
- The next few slides refer to the `SimplePaintMenu` program that you can download from the class website.

  This program was also given out in class.
- This is an extension of the `SimplePaint` program from the last lecture.

- We start by adding a menu bar, which is where menus live:

```
JFrame frame = new JFrame("SimplePaint");
Container pane = frame.getContentPane();

JMenuBar menuBar = new JMenuBar();
frame.setJMenuBar(menuBar);
```

- Note that we add the menu bar to the `JFrame`, not to the `ContentPane`.

- Once we have a menu bar, then we can add menus:

```
JMenu fileMenu = new JMenu("File");
JMenu optionsMenu = new JMenu("Options");
menuBar.add(fileMenu);
menuBar.add(optionsMenu);
```

- And we can add shortcuts, keystrokes that will activate the menu options:

```
fileMenu.setMnemonic('F');
optionsMenu.setMnemonic('O');
```

  (To use these, hold down the Alt key while pressing the relevant letter).

- We can then add items to the menus.
- Here we add an itemn to the **File** menu:

```
JMenuItem exit = new JMenuItem("Exit", 'x');
fileMenu.add(exit);
exit.addActionListener(new GoodBye());
```

- The `'x'` is the keyborad shortcut here.
- The action listener will be called when the menu item is selected.

- In the above example, the connection between listener and menu is simple — there is one listener for one menu item.
- Here is a more complex example from `SimplePaintMenu`

```
JMenu penAdjustMenu = new JMenu("Pen Size");
penAdjustMenu.setMnemonic('P');
JMenuItem smallPen = new JMenuItem("Small", 'S');
penAdjustMenu.add(smallPen);
JMenuItem mediumPen = new JMenuItem("Medium", 'M');
penAdjustMenu.add(mediumPen);
JMenuItem largePen = new JMenuItem("Large", 'L');
penAdjustMenu.add(largePen);
optionsMenu.add(penAdjustMenu);
```


- We have one listener for all these menu items:

```
PenAdjuster penAdjust = new PenAdjuster(listener);
smallPen.addActionListener(penAdjust);
mediumPen.addActionListener(penAdjust);
largePen.addActionListener(penAdjust);
```


- In the listener we define:

```
public void actionPerformed(ActionEvent e){
   painter.setPenSize(e.getActionCommand());
}
```

- The `getActionCommand()` returns the string in the menu item so that we can write:

```
public void setPenSize(String size){
   if (size.equals("Small")){
      radius = 0;
      diameter = 1;
}
```

Other GUI items

- Two useful components are check boxes and radio buttons.
- Radio buttons need to be grouped. Here's an example:
- Given:

```
JPanel myPanel = new JPanel();
```

we can add a button to the panel using:

```
ButtonGroup myGroup = new ButtonGroup();
JRadioButton radioButton = new JRadioButton("Java");
radioButton.setActionCommand("Java");
myPanel.add(radioButton);
myGroup.add(radioButton);
```

- And we would repeat for all the other radio buttons we want grouped together.

- In the listener we then ask the `ButtonGroup` what the `actionCommand` of the selected button is:

```
string lang =
   myGroup.getSelection().getActionCommand();
```

- Note that in this case we are assuming that the listener is an object inside the one that defines the interface, allowing direct access to `myGroup`.

- Check boxes are easier to set up since we don't have an object to group them, though we will typcially handle them by adding them to their own panel:

```
JPanel mypanel2 = new JPanel();
myPanel2.add(new JCheckBox("Java"));
```

and we repeat the second part for other check boxes that we want to group together.

- This panel is then added to a container that is part of the interface.

- Since we can have several check boxes selected, we need to handle them something like this in the listener:

```
Component[] components = myPanel2.getCOmponents();

for(Component c : components){
   JCheckBox cb = (JCheckBox) c;
  if(cb.isSelected()){
   <whatever action we want>
}
```

- Again we assume that the listener has direct access to the variables in the interface.
- Note that if `myPanel2` contained other kinds of component, we would have to pick out the check boxes before the `if` construct, for example by using `instanceof`.

Exception Handling

- The next few slides refer to the `TextInputWExceptH` program that you can download from the class website.
- This is an extension of the `TextInput` program from the last lecture.

- Exception handling is one of the nice features of Java.
  - Provides an elegant way to handle runtime problems with programs.
  - Allows programs to start running even when the compiler knows there are problems.
- It is a legacy of Java's origins in embedded systems.
  - Now exploited in Android

- Exceptions are handled using:

  `try`
  `catch`
  `finally`
- Also we may use:

  `throw`

  when we want to generate exceptions.

- When we expect code to generate exceptions that we will handle, we wrap the code in a `try` construct.

```
try{

  // This will run normally, except when there
  // is an exception.
}
```

- To handle an exception, we add a `catch` to the `try`.

```
try{

}
catch(SomeException e1){

   // If the try generates an exception, Java will
   // try to match it against SomeException.
   //
   // If there is a match, code here will be excuted.
   //
   // That code can refer to the exception as e1.
}
```


- We can have multiple `catch` constructs for any `try`.
- It is typical to have different `catch`es to handle the different exceptions that might arise.
- If an execption in a given `try` does not match a following `catch`, the interpreter looks for an enclosing `try` and its corresponding `catch`es.
- If these don't exist, or don't match the exception, it tries the calling method.
- This process will eventually (if nothing catches the exception) percolate up to `main()`.
- If this doesn't catch the exception, the interpreter prints a stack trace on the console and exits.

  (You will likely have seen this already.)

- After the last `catch`, we may have a `finally`.
- The code in a `finally` construct is always called after the `try` is executed.
- This happens whether or not there is an exception, and whether or not the exception is handled.
- The only time `finally` does not run is if `try` calls `System.exit()`.
- `finally` is useful for doing housekeeping things.

- Note that all exceptions are (of course) objects.
- The base class for exceptions is `java.lang.Throwable`
- `Throwable` has two subclasses:
  - `Error`
  - `Exception`
- It is rare to want to try to handle `Error`s. They are usually fatal and best left alone.

- Sometimes we want to generate our own exception.
  - That is we want to force one to happen.
- This allows us to use exception handling as a way of doing error checking.

```
if (d2 < 0){
   throw new IllegalArgumentException();
}
```

- The above example is from `TextInputWExceptH2`.

- The exception that is thrown must be a bona fide exception object.
- But, of course, we can create our own exception (extending an existing exception) if we need to.

Summary

- This lecture looked very quickly at a number of interface components in Java:
  - Buttons
  - Reading from text fields
  - Writing to text fields
  - Listeners for multiple components
  - Simple drawing
  - Mouse events