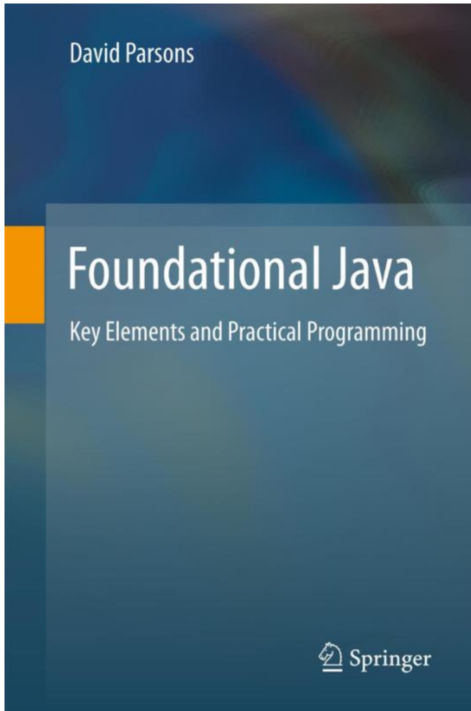


Chapter 18
Event Driven Programming

Foundational Java

Key Elements and Practical Programming

Foundational Java by David Parsons © 2012

Event Driven GUIs

- In a GUI various events may occur
  - Press a button
  - Type into a text field
  - Click on a radio button
  - Close a window
  - etc.
- Which events occur, and in which order, is largely unpredictable
- Code must be ready to respond to the various events that may be triggered by the user's actions

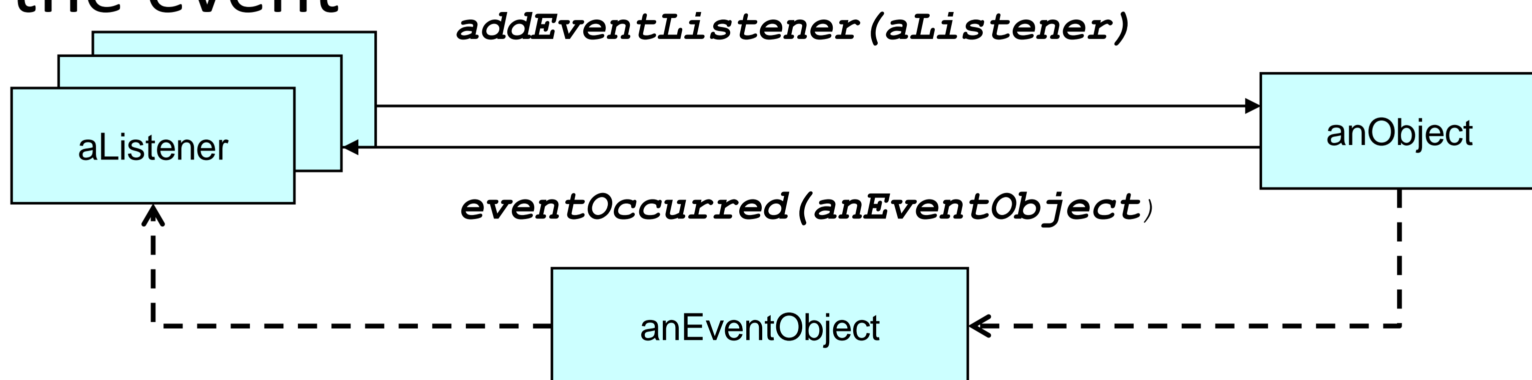
Foundational Java by David Parsons © 2012

Event Listeners

- Listener objects respond to events in the user interface
  - There can be multiple listeners for each event
  - Listeners are registered with event sources
- Listeners implement interfaces of event handler methods
- Event objects are passed to these methods
  - e.g. mouse button is clicked
  - Event object informs listeners about the location of the mouse when the button click occurred

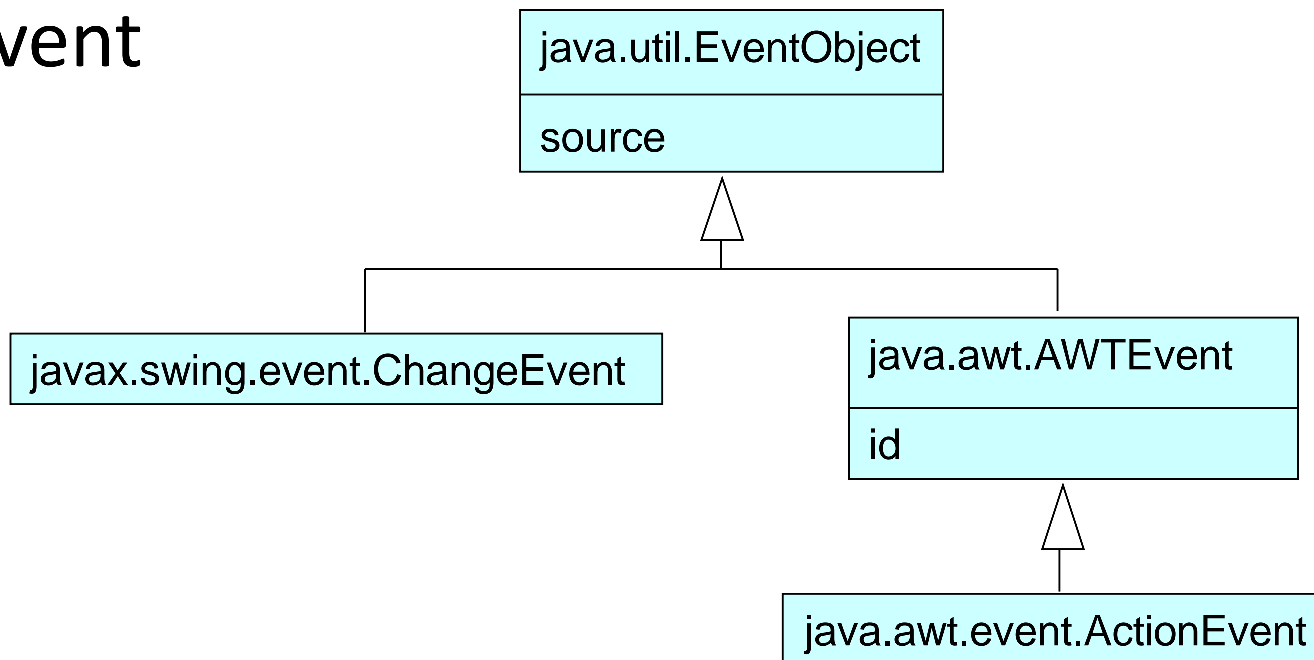
Listeners and Events

- ‘anObject’ is able to fire events
- Listener objects are added to its collection of listeners
- If an event is fired, all the listeners are sent an event object that contains information about the event



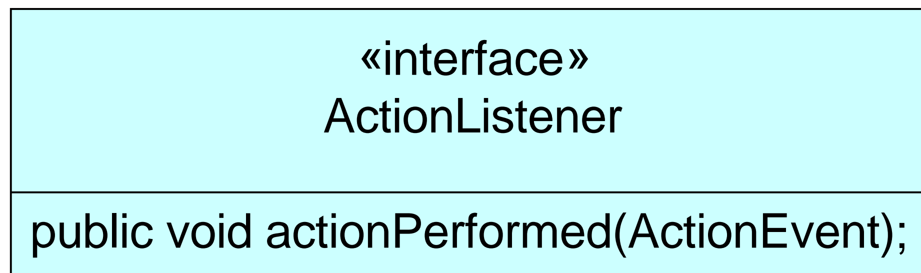
Event Classes

- Event classes are in a hierarchy that spans a number of packages
- Each class represents a particular type of event



EventListener Interfaces

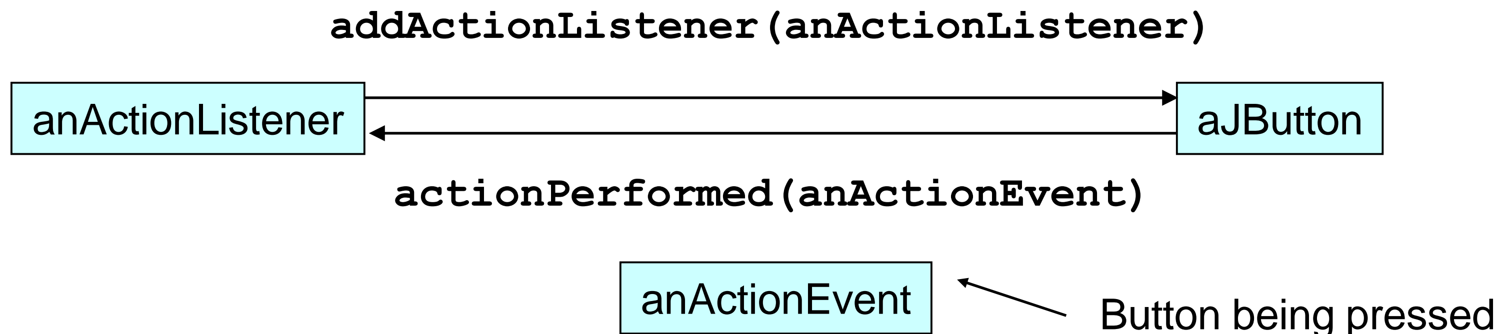
- For each EventObject type there are one or more event listener interfaces defined
  - Each interface declares methods which must be implemented by listeners to handle specific events
- e.g. ActionListener interface declares the method ‘actionPerformed’, which handles an ActionEvent
- MouseListener declares several methods, and handles MouseEvents



e.g. Button being pressed

JButton ActionEvent

- Many components can fire an ActionEvent
- JButton is commonly used
  - An ActionListener can register with a JButton via the 'addActionListener' method
  - This ActionListener will have its 'actionPerformed' method invoked each time the button is pressed.



Closing a JFrame

- This CloseButtonListener implements ActionListener
  - The constructor takes a reference to a JFrame
  - When 'actionPerformed' is invoked the JFrame is disposed of

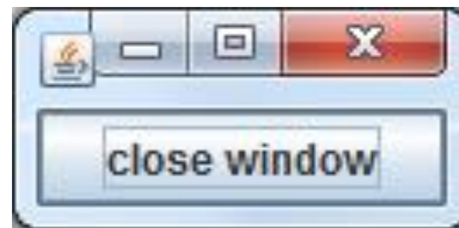
```
public class CloseButtonListener implements ActionListener {
  private JFrame target;
  public CloseButtonListener(JFrame aFrame)  {
    target = aFrame;
  }
  public void actionPerformed(ActionEvent event)  {
    target.dispose();
  }
}
```

Adding the ActionListener

- A JButton as the source of the event
- An instance of CloseButtonListener is added to it

```
JFrame frame = new JFrame();
JButton closeButton = new JButton("close window");
closeButton.addActionListener(new CloseButtonListener(frame));
```

- When the 'close window' button is pressed, the window closes



Multiple Action and Focus Listeners

- Event sources and listeners form many-to-many relationships
- Multiple listeners can listen to the same event from the same source
- One listener can listen for events from multiple sources

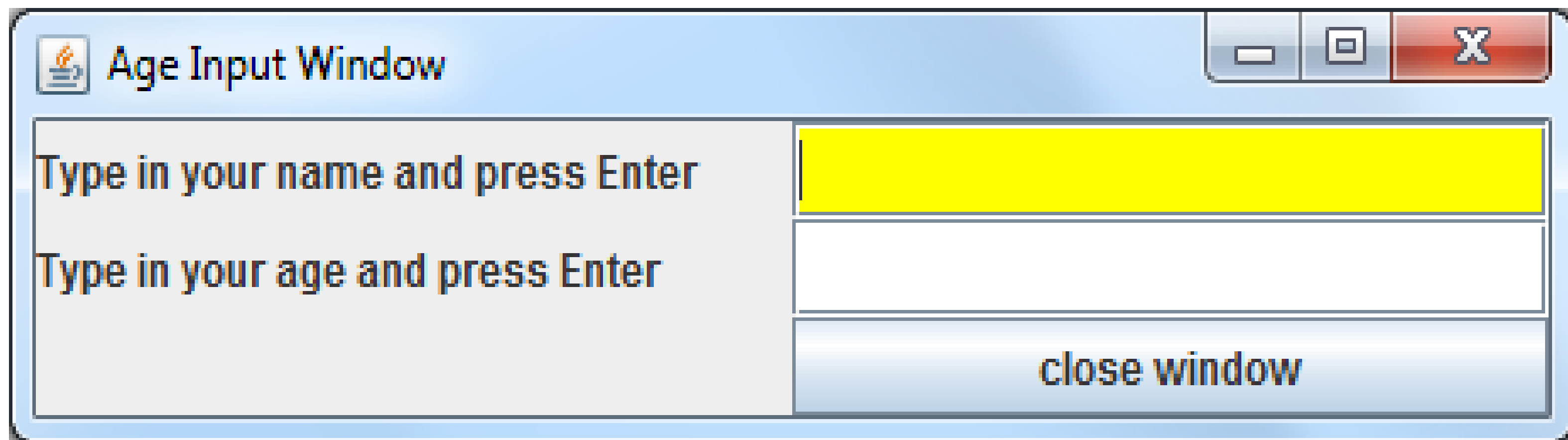
Foundational Java by David Parsons © 2012

JTextField Events

- JTextFields can trigger many different types of event
  - e.g. ActionEvent, FocusEvents
- Focus events occur when the field gains or loses focus
- An action event is triggered by pressing Enter when the JTextField has focus

Example: Validating Text Fields

- Same FocusListener is used with two different JTextFields
- FocusListener and ActionListener applied to a single JTextField



The FocusListener

- This listener will change the background from white to yellow when the text field has focus, and back to white again when it loses focus

```
public class TextFieldFocusListener implements FocusListener  {
  public void focusGained(FocusEvent e)
  {
    JTextField field = (JTextField)e.getSource();
    field.setBackground(Color.YELLOW);
  }
  public void focusLost(FocusEvent e)
  {
    JTextField field = (JTextField)e.getSource();
    field.setBackground(Color.WHITE);
  }
}
```

Uses 'getSource' to access the JTextField that fired the event

ActionListener

- The second text field has an ActionListener
- Checks if the value typed in can be parsed as an integer using this method

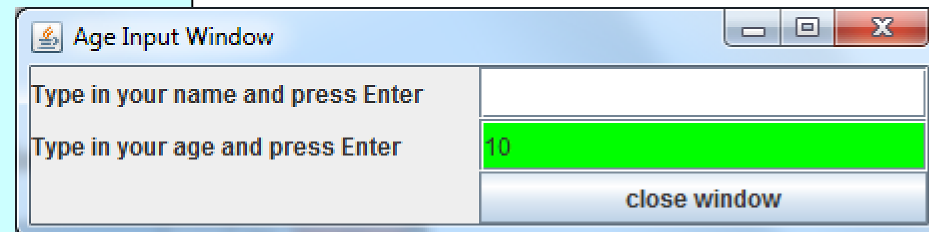
```
private boolean isValidAge(String ageValue)
  {
    try
    {
      Integer.parseInt(ageValue);
      return true;
    }
    catch (NumberFormatException e)
    {
      return false;
    }
  }
```

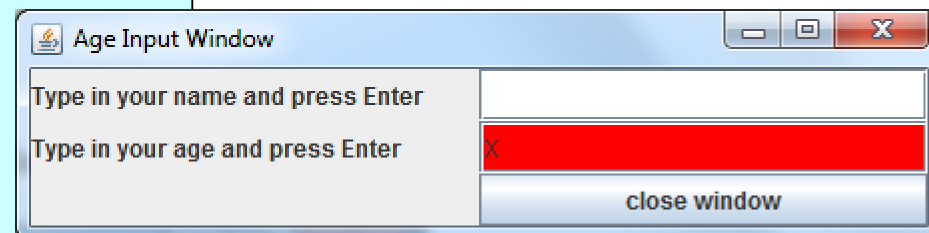
ActionListener

– Text that cannot be parsed as an integer causes the text field background to turn red

– Valid numbers turn it green

```
public void actionPerformed(ActionEvent e) {
    JTextField field = (JTextField) e.getSource();
    if (isValidAge(field.getText()))
    {
      field.setBackground(Color.green);
    }
    else
    {
      field.setBackground(Color.red);
    }
  }
```





Exercise 18.1

- Write a simple GUI using a JFrame that contains a text box, a combo box and a button
- When the button is pressed, the current contents of the text box should be added to the combo box

Foundational Java by David Parsons © 2012

Exercise 18.2

- Add a second button to your JFrame that will close the frame when it is pressed

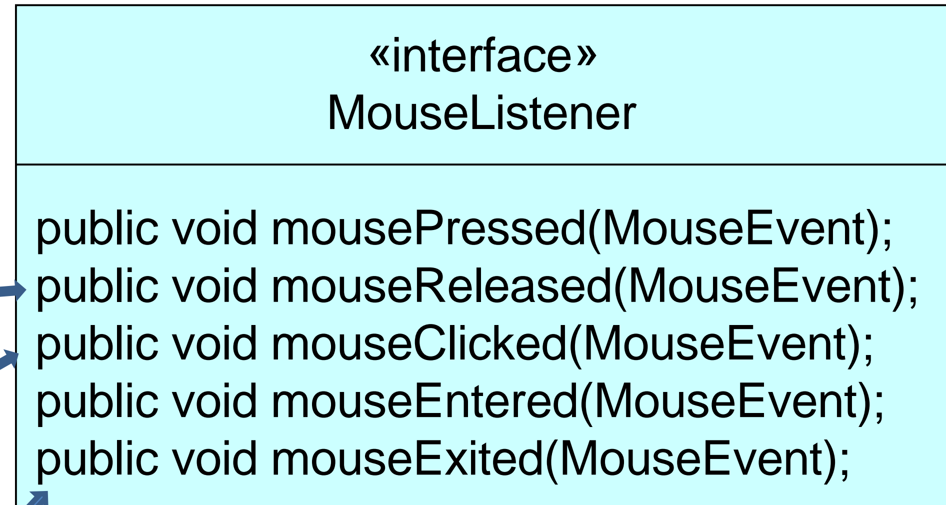
Foundational Java by David Parsons © 2012

Responding to Mouse Events

- There are different kinds of mouse events
  - Mouse movement, mouse buttons, mouse wheel, dragging…
- To handle the various mouse events there are several different Listener interfaces
- MouseListener
  - Mouse button presses
  - Crossing the bounds of a component
- MouseMotionListener
  - Mouse movement
  - Mouse dragging (moving with button down)

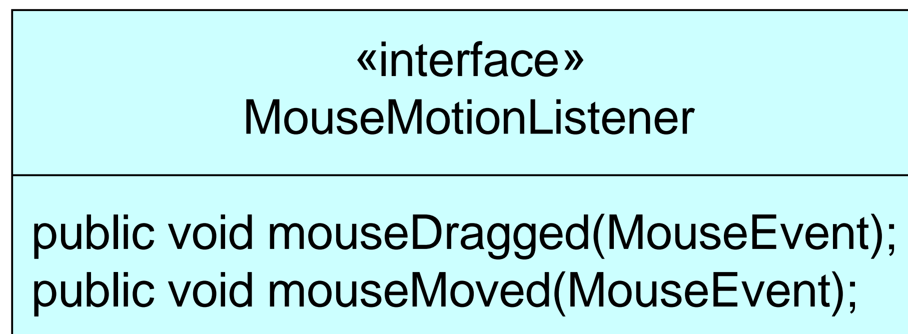
MouseListener interface

- button down
- button up
- pressed and released
- mouse moving in or out of the area covered by a component



MouseMotionListener

- In addition we can trace the motions of the mouse, either moving (no button pressed) or dragging (a button being pressed down), by implementing the MouseMotionListener interface



MouseEvents

- The MouseEvent object passed as a parameter to MouseListener methods contains a Point
  - The mouse position when a button was pressed

```
position = event.getPoint();
```

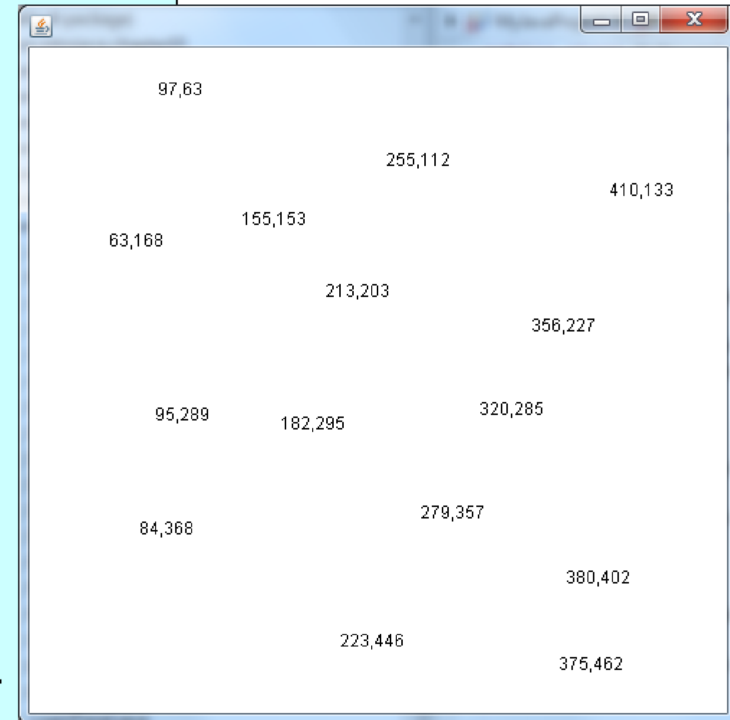
- Can use this information in the event handler

```
frame.getGraphics().drawString(position.x + "," + position.y, position.x, position.y);
```

MouseListener

```
public class MyMouseListener implements MouseListener
{
  private Point position;
  private JFrame frame;
  public MyMouseListener (JFrame frame)
  {
    this.frame  = frame;
  }

  @Override
  public void mousePressed(MouseEvent event)
  {
    position = event.getPoint();
    frame.getGraphics().drawString(position.x + "," +
        position.y, position.x, position.y);
  }
```



Mouse Motion Events

- The next example demonstrates the MouseMotionListener interface
- Implements the MouseMotionListener in the DrawMouseListener class
- Only the ‘mouseDragged’ method has been given a meaningful implementation
- The other method, ‘mouseMoved’ is empty

‘mouseDragged’

- This method responds to the left button being pressed (and held)

```
public class DrawMouseListener implements MouseMotionListener
{
  @Override
  public void mouseDragged(MouseEvent e)
  {
    position = e.getPoint();
    frame.getGraphics().fillRect(position.x, position.y, 5, 5);
  }

  @Override
  public void mouseMoved(MouseEvent e)
  {    // empty implementation    }
}
```

Exercise 18.3

- Write a program that draws a line between two points within a frame
- Implement a subclass of MouseListener that captures 'mouseClicked' events, and draw a line between two consecutive mouse clicks
- Use the 'drawLine' method of the Graphics class
  - This method draws a line, using the current color, between the points (x1, y1) and (x2, y2)

```
drawLine(int x1, int y1, int x2, int y2)
```

Event Listener Adapter Classes

- Each EventListener interface corresponds to a group of related events
  - The event handlers must implement each method in the interface
  - Each event in the group must be handled whether interesting or not
- Event listener adapter classes implement all the methods of a particular event listener interface
  - Implements a stub for each method
  - Handlers can subclass the adapter class and override only those methods of interest

Exercise 18.4

- Create a subclass of MouseAdapter that overrides the ‘mouseClicked’ event in the same way as your previous MouseListener class
- Modify the program you wrote for the previous exercise so that it uses your new class rather than the previous MouseListener

Event Handlers as Inner Classes

- Separate event handler classes may be useful
  - For generic operations
- Other handlers are closely tied to a particular application
- Event handlers as separate classes means passing objects between the event handlers and other objects
- A more elegant solution is to write event handlers as inner classes

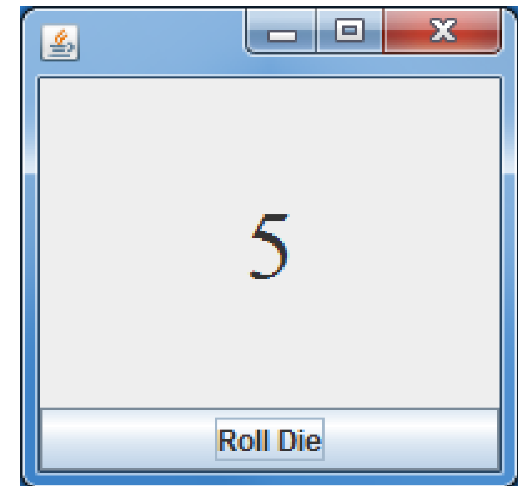
DieListener as Separate Class

```
public class DieListener implements ActionListener
{
  private Die die;
  JLabel dieScore;

  public DieListener(Die die, JLabel dieScore)
  {
    this.die = die;
    this.dieScore = dieScore;
  }

  @Override
  public void actionPerformed(ActionEvent e)
  {
    dieScore.setText(String.valueOf(die.getRoll()));
  }
}
```

References passed around



Inner Classes

- A listener is just a class that provides listener behavior
  - Classes for listeners may seem a little excessive
- Inner classes are lightweight classes, defined inside other classes
- They can directly access all fields and methods of the enclosing instance including private fields and methods

Foundational Java by David Parsons © 2012

Inner Class

- Declared within the body of another class
  - Direct access all the fields and methods of that enclosing class
  - No external listener class, no need to pass parameter references around between objects

```
class DieListener implements ActionListener {
  public void actionPerformed(ActionEvent e)
  {
    dieScore.setText(String.valueOf(die.getRoll()));
  }
};
```

Local Inner Class

- Declared within the body of a method
- Only visible within that method
- Could be used multiple times inside the method

```
ActionListener dieListener = new ActionListener()
 {
   public void actionPerformed(ActionEvent e)
   {
     dieScore.setText(String.valueOf(die.getRoll()));
   }
 };
 rollButton.addActionListener(dieListener);
```

Note the final semicolon

Anonymous Inner Class

- ActionListener implementation applied directly to the ‘addActionListener’ method
- No class name and no reference name

```
rollButton.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        dieScore.setText(String.valueOf(die.getRoll()));
      }
    });
```

Note the final parenthesis and semicolon

Event Handlers, Visibility and Reusability

Type of event handler class	Visibility	Level of Reusability
Separate class	public	Across different applications
Inner class	Within a class (like a field)	In any method of the class (e.g. multiple handler objects of the same class can be created and applied to multiple components in different methods)
Local inner class	Within a method (locally named, like a local variable)	In the same method of the class (e.g. the same handler object can be applied to multiple components in the same method)
Anonymous inner class	Within a method (no local name)	A single instance of the handler class can be used with a single component

JPanel Subclasses and Multi Panel Layouts

- Sometimes we need panels that have application-specific roles
- JFrame’s layout manager can host different panels that have their own layouts
- We can define any number of subclasses of JPanel to manage different components or display graphical output

ScaledImagePanel Example

- Maintains properties (fields, with getters and setters) for the height and width of the image
- Draws an image from an external file

```
public Image getImage()
{
  Toolkit kit = Toolkit.getDefaultToolkit();
  return kit.getImage(getImageFilename());
}
```

- 'paint' method displays the scaled image
  - final parameter is 'this' ImageObserver

```
public void paint(Graphics g)
{
  g.fillRect(0, 0, this.getWidth(), this.getHeight());
  g.drawImage(getImage(), 0, 0, getImageWidth(), getImageHeight(), this);
}
```

ImageControlPanel

- Contains two JSliders that the user can control to change the width and height of the scaled image
- The movement of a JSlider triggers ChangeEvents, captured in the ‘stateChanged’ method of the ChangeListener interface
- In this case, a local inner class is used

Foundational Java by David Parsons © 2012

ChangeListener

- Single event handler used by both JSliders

```
ChangeListener sliderListener = new ChangeListener()
{
  @Override
  public void stateChanged(ChangeEvent e)
  {
    if(e.getSource().equals(heightSlider))
    {
      scaledImagePanel.setImageHeight(heightSlider.getValue());
    }
    else
    {
      scaledImagePanel.setImageWidth(widthSlider.getValue());
    }
  }
};
heightSlider.addChangeListener(sliderListener);
widthSlider.addChangeListener(sliderListener);
```

ImageScaler

- The scale of an image on one panel is controlled by components (in a GridLayout) on a separate panel
- Both panels are added to the frame's BorderLayout

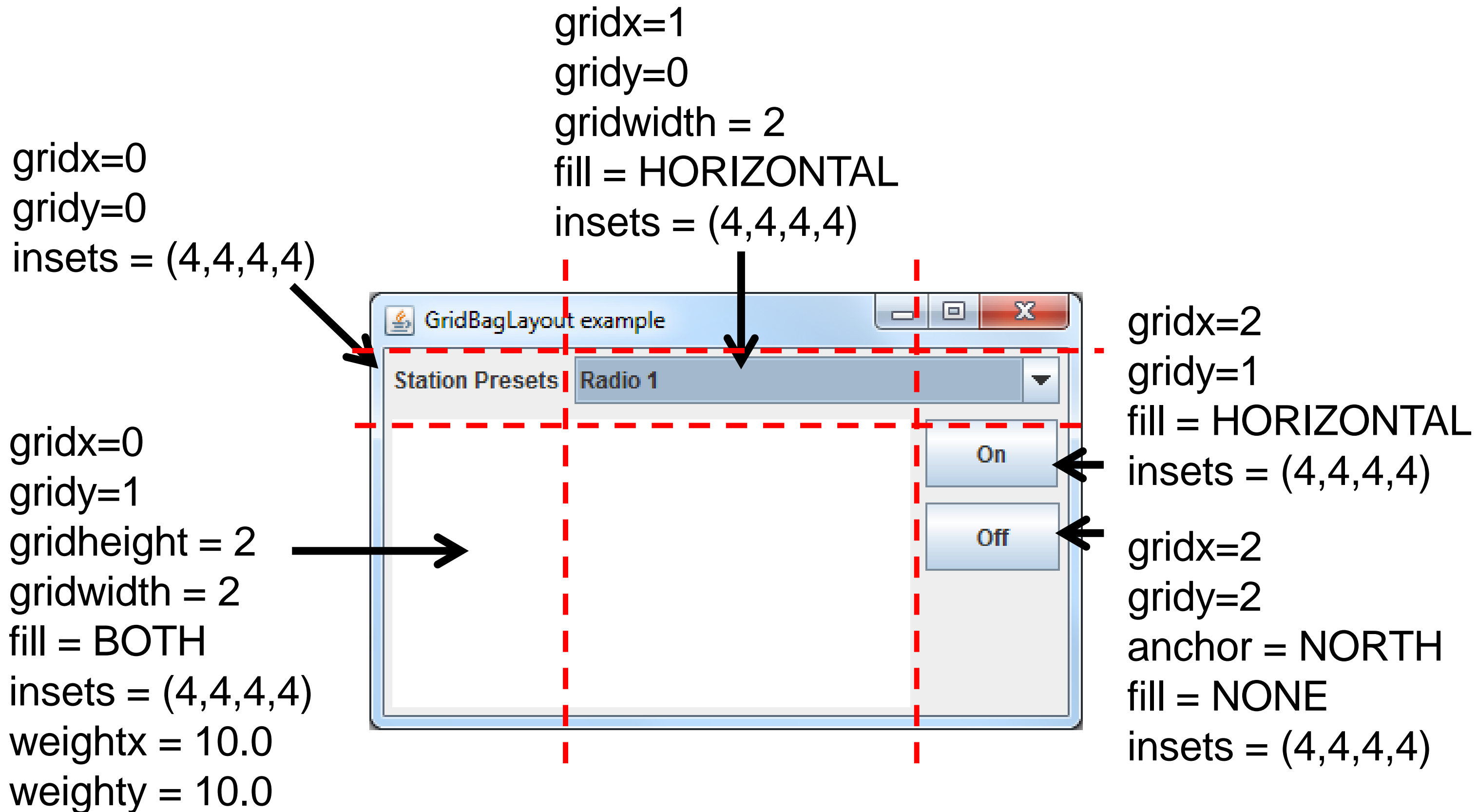
GridBagLayout Manager

- The most sophisticated of the original AWT layout managers
- Has two aspects
  - The layout grid, on which components are positioned using x and y coordinates, starting at 0, 0 for the top left hand cell of the grid
  - The way that each component is managed inside the cells of the grid is specified by a GridBagConstraints object

GridBagConstraints

Constraint	Manages
gridx and gridy	Position on the grid - top left is 0, 0
gridwidth and gridheight	Number of rows or columns spanned
fill	How space is filled within a cell
insets	Margins around a component
ipadx and ipady	Increases the size of the component
anchor	Position within the cell
weightx and weighty	Weight relative to other components

GridBagLayout Example



Resize Behaviour

- GridBagLayouts provide good resizing behaviour, allowing the weighted components to take up the slack without disrupting other components.

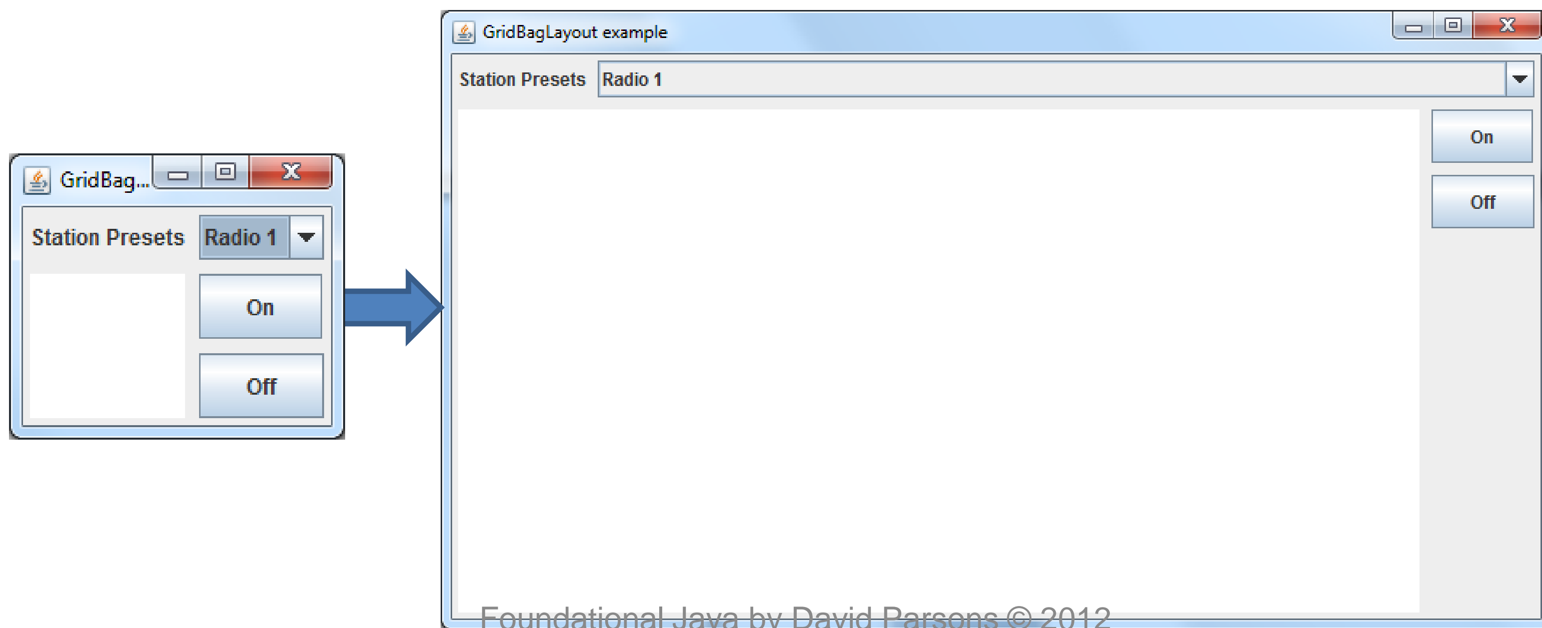
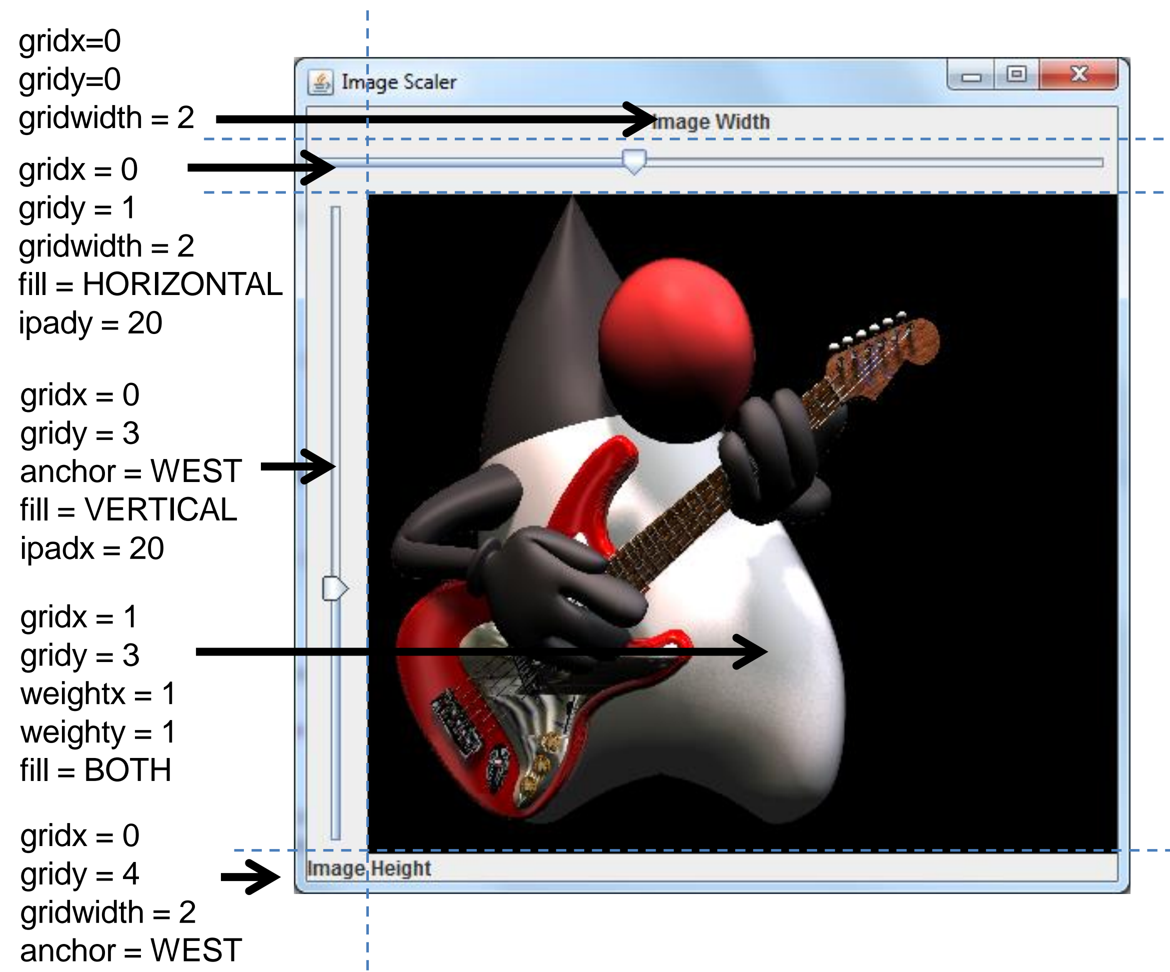


Image Scaler in a GridBagLayout



Exercise 18.5

- Modify your answer to Exercise 17.5 by applying a GridBagLayout to the panel
- Give each component an appropriate GridBagConstraints object

Separating the ‘Model’ from the ‘View’

- An important design issue in programs with a graphical interface is the separation of ‘model’ (the underlying data in a program) and ‘view’ (the GUI that the user sees)
- In larger programs we need to separate out the underlying program from its interface
- That way we can develop, and test, programs and interfaces separately and make them more independent and maintainable

Foundational Java by David Parsons © 2012

Triangular Array Data Model

- A table of distances between major international cities
- Triangular array - only half of the two dimensional table of values is needed,

Beijing	645								
Cairo	1029	4695							
Cape Town	7329	8044	4479						
London	1138	5070	2183	5987					
Mumbai	7040	2957	2710	5105	4476				
New York	8815	6842	5618	7799	3471	7807			
Rio De Janeiro	7635	1076	6140	3775	5750	8338	4803		
Sydney	1342	5545	8958	6856	10558	6306	9934	8412	
Tokyo	5474	1307	5957	9156	5956	4194	6755	11532	4842
	Auckland	Beijing	Cairo	Cape Town	London	Mumbai	New York	Rio De Janeiro	Sydney

The Model Class: FlightDistances

- Contains data about the distances (in miles) between cities
  - The names of ten cities are held in a static array of Strings.
  - Client code can access this array via the static ‘getCities’ method.
- The class uses a static two-dimensional array of integers to hold the distance data

The 'getDistance' Method

- Two integer parameters
  - The origin and destination of the journey
  - returns the distance between them

```
public static int getDistance(int origin, int destination)
{
  try
  {
    if (origin < destination)
    {   return distanceTable[origin][destination];   }
    else
    {   return distanceTable[destination][origin];   }
  }
  catch (ArrayIndexOutOfBoundsException e)
  { … }
}
```

Handles the triangular array (uses only one half)

Testing the Model

- The model is separate from the UI
- Can be tested separately. e.g.,
  - Test same origin and destination
  - Test distance value returned by the method
  - Test the same distance is returned if indexes of origin and destination are switched
  - Test out of range integer parameter
  - Test that the array of city names is not null
  - Test a valid city name index

The View Class: DistanceViewer

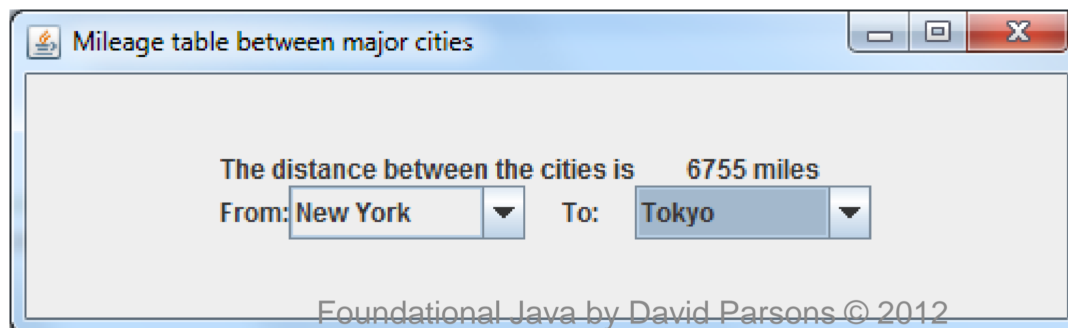
- JComboBoxes for the names of the origin and destination
- ActionEvents triggered when a selection is changed

```
ActionListener cityListener = new ActionListener() {
  public void actionPerformed(ActionEvent e)
  {
    showDistance(fromCity.getSelectedIndex(), toCity.getSelectedIndex());
  }
};
```

The ‘view’

- ‘showDistance’ gets the distance from the model class
  - Sets the text of the ‘resultLabel’.

```
private void showDistance(int fromCity, int toCity)
{
  int distance = FlightDistances.getDistance(fromCity, toCity);
  String measureString = " miles";
  resultLabel.setText(distance + measureString);
}
```



Exercise 18.6

- Modify the DistanceViewer so that it allows the user to choose between displaying the distances in either miles or kilometers, using two radio buttons in a button group
- Use the radio button ActionEvents to change the way that the distance is displayed
- The conversion between miles and kilometers should use the following formula
  - 1 mile = 1.60934 kilometers

Summary

- Events
  - ActionEvents, MouseEvents, ChangeEvents
- Event listeners
  - Separate classes, inner classes, local inner classes, anonymous inner classes
- Multiple panels
- GridBagLayout
- Separating model from view