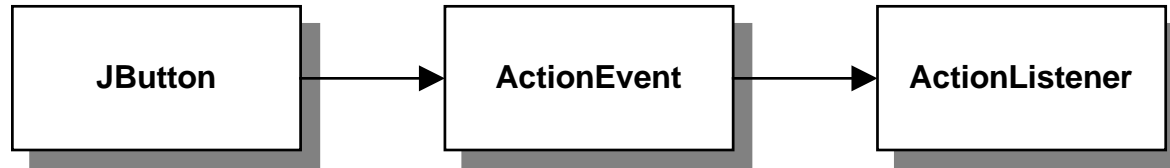
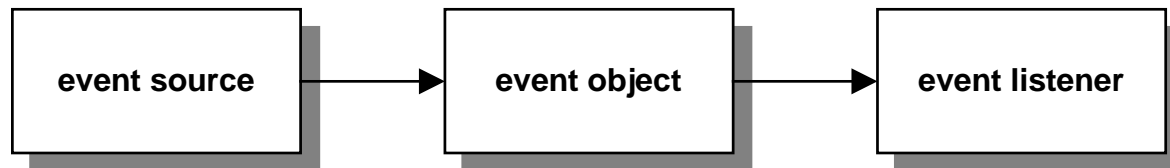


GUI Event Handling and validate data

What happens when a button is pressed



What happens when any event occurs



The Java event model

- GUI applications depend on *events* that represent user interactions such as clicking a button or selecting an item from a list.
- All events are represented by an *event object* that derives from the `EventObject` class. The event object contains information about the event that occurred.
- An *event listener* is an object that responds to an event.
- The class that defines an event listener must implement an event listener interface.
- A component that generates an event is called an *event source*.
- To respond to an event, an application must *register* an event listener object with the event source that generates the event.
- The class for the event source provides a method for registering event listeners. Then, when the event occurs, the event source creates an event object and passes it to the event listener.

Semantic events

Action	Event object	Listener interface
Button clicked	ActionEvent	ActionListener
Combo box item selected	ActionEvent	ActionListener
	ItemEvent	ItemListener
List item selected	ListSelectionEvent	ListSelectionListener
Text component changed	DocumentEvent	DocumentListener
Radio button selected	ActionEvent	ActionListener
	ItemEvent	ItemListener
Check box selected	ActionEvent	ActionListener
	ItemEvent	ItemListener
Scroll bar repositioned	AdjustmentEvent	AdjustmentListener

Low-level events

Action	Event object	Listener interface
Window changed	WindowEvent	WindowListener
Focus changed	FocusEvent	FocusListener
Key pressed	KeyEvent	KeyListener
Mouse moved or clicked	MouseEvent	MouseListener

The two types of Java events

- Two types of events exist in Java: *semantic events* and *low-level events*.
- A semantic event is related to a specific component such as clicking a button or selecting an item from a list.
- Low-level events are less specific, like clicking a mouse button, pressing a key on the keyboard, or closing a window.
- Most events and listeners are stored in the `java.awt.event` package, but some of the newer events and listeners are stored in the `javax.swing.event` package.
- Some user actions create more than one event. You can use listeners to respond to any of them.

Two steps to handle any event

1. Create a class that implements the appropriate listener interface. In this class, you must code an implementation of the appropriate listener interface method to respond to the event.
2. Register an instance of the listener class to the event source by calling the appropriate *addEvent*Listener method

Four options for implementing the listener interface

- Implement it in the panel itself
- Implement it in a separate class
- Implement it in an inner class within the panel
- Implement it in an anonymous inner class

Two options for handling multiple event sources

- Create one listener that handles all events for the panel
- Create a separate listener for each event

Code for a panel that implements the ActionListener interface

```
class FutureValuePanel extends JPanel
    implements ActionListener
{
    private JButton calculateButton;
    private JButton exitButton;

    public FutureValuePanel()
    {
        calculateButton = new JButton("Calculate");
        calculateButton.addActionListener(this);
        this.add(calculateButton);

        exitButton = new JButton("Exit");
        exitButton.addActionListener(this);
        this.add(exitButton);
    }
}
```

Code for a panel that implements the ActionListener interface (continued)

```
public void actionPerformed(ActionEvent e)
{
    Object source = e.getSource();
    if (source == exitButton)
        System.exit(0);
    else if (source == calculateButton)
        calculateButton.setText("Clicked!");
}
}
```

Notes

- The easiest way to implement a listener interface is in the class that defines the panel or frame that contains the components that generate the events.
- When the panel or frame class itself implements the listener, you can specify the `this` keyword as the parameter to the method that registers the listener.

Code for a panel that uses a separate listener class: The panel class

```
class FutureValuePanel extends JPanel
{
    public JButton calculateButton;
    public JButton exitButton;

    public FutureValuePanel()
    {
        ActionListener listener =
            new FutureValueActionListener(this);
        calculateButton = new JButton("Calculate");
        calculateButton.addActionListener(listener);
        this.add(calculateButton);

        exitButton = new JButton("Exit");
        exitButton.addActionListener(listener);
        this.add(exitButton);
    }
}
```

Code for a panel that uses a separate listener class: The listener class

```
class FutureValueActionListener implements ActionListener
{
    private FutureValuePanel panel;

    public FutureValueActionListener(FutureValuePanel p)
    {
        this.panel = p;
    }

    public void actionPerformed(ActionEvent e)
    {
        Object source = e.getSource();
        if (source == panel.exitButton)
            System.exit(0);
        else if (source == panel.calculateButton)
            panel.calculateButton.setText("Clicked!");
    }
}
```

How to implement an event listener as a separate class

- If you implement a listener as a separate class, you'll need to provide a way for the listener class to access the source components and any other panel components that are required to respond to the event.
- One way to do that is to pass the panel to the constructor of the listener class and declare the components that need to be referred to as public.

How to implement an event listener as an inner class

- An *inner class* is a class that is contained within another class.
- An inner class has access to all of the members of its *containing class*. Because of that, inner classes are often used to implement event listeners.

Code that implements the listener as an inner class

```
class FutureValuePanel extends JPanel
{
    private JButton calculateButton;
    private JButton exitButton;

    public FutureValuePanel()
    {
        ActionListener listener =
            new FutureValueActionListener();
        calculateButton = new JButton("Calculate");
        calculateButton.addActionListener(listener);
        this.add(calculateButton);

        exitButton = new JButton("Exit");
        exitButton.addActionListener(listener);
        this.add(exitButton);
    }
}
```

Code that implements the listener as an inner class (continued)

```
class FutureValueActionListener
    implements ActionListener
{

    public void actionPerformed(ActionEvent e)
    {
        Object source = e.getSource();
        if (source == exitButton)
            System.exit(0);
        else if (source == calculateButton)
            calculateButton.setText("Clicked!");
    }
}
```


How to implement separate event listeners for each event

- You can eliminate the code in the event listener class that determines the event source by creating a separate listener class for each component that raises the event.
- In that case, you simply register an instance of each event listener class with the appropriate event source.

Code that implements separate listeners for each event

```
class FutureValuePanel extends JPanel
{
    private JButton calculateButton;
    private JButton exitButton;

    public FutureValuePanel()
    {
        calculateButton = new JButton("Calculate");
        calculateButton.addActionListener(
            new CalculateListener());
        this.add(calculateButton);

        exitButton = new JButton("Exit");
        exitButton.addActionListener(new ExitListener());
        this.add(exitButton);
    }
}
```

Code that implements separate listeners for each event (continued)

```
class CalculateListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        calculateButton.setText("Clicked!");
    }
}

class ExitListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
}
}
```

The syntax for creating an anonymous class for an event listener

```
new ListenerInterface() { class-body }
```

How to implement event listeners as anonymous inner classes

- An *anonymous inner class* is a class that is both declared and instantiated in one statement.
- Anonymous inner classes are often used as event listeners.
- Anonymous inner classes force you to mix the code that creates a panel with the code that responds to the panel's events. So they should be used for only the simplest event listeners.

Code that implements event listeners as anonymous classes

```
class FutureValuePanel extends JPanel
{
    private JButton calculateButton;
    private JButton exitButton;

    public FutureValuePanel()
    {
        calculateButton = new JButton("Calculate");
        calculateButton.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    calculateButton.setText("Clicked!");
                }
            } );
        this.add(calculateButton);
    }
}
```

Code that implements event listeners as anonymous classes (continued)

```
        exitButton = new JButton("Exit");
        exitButton.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    System.exit(0);
                }
            } );
        this.add(exitButton);
    }
}
```

The showMessageDialog method of the JOptionPane class





Syntax

```
showMessageDialog(parentComponent, messageString,  
                  titleString, messageTypeInt);
```

Arguments

Argument	Description
<code>parentComponent</code>	An object representing the component that's the parent of the dialog box.
<code>messageString</code>	A string representing the message to be displayed in the dialog box.
<code>titleString</code>	A string representing the title of the dialog box.
<code>messageTypeInt</code>	An int that indicates the type of icon that will be used for the dialog box.

Fields used for the message type parameter

Icon displayed	Field
(none)	PLAIN_MESSAGE
	INFORMATION_MESSAGE
	WARNING_MESSAGE
	ERROR_MESSAGE
	QUESTION_MESSAGE

How to display error messages

- The showMessageDialog method is a static method of the JOptionPane class that is commonly used to display dialog boxes with error messages for data validation.
- You can also use the JOptionPane class to accept input from the user.

Code that displays the Invalid Entry dialog box

```
String message = "Monthly Investment is a required field.\n"
    + "Please re-enter.";
JOptionPane.showMessageDialog(this, // assumes "this" is a component
    message, "Invalid Entry",
    JOptionPane.ERROR_MESSAGE);
```

How to validate the data entered into a text field

- Like console applications, Swing applications should validate all data entered by the user before processing the data.
- When an entry is invalid, the program needs to display an error message and give the user another chance to enter valid data.
- To test whether a value has been entered into a text field, you can use the `getText` method of the text field to get a string that contains the text the user entered. Then, you can check whether the length of that string is zero by using its `length` method.
- To test whether a text field contains valid numeric data, you can code the statement that converts the data in a `try` block and use a `catch` block to catch a `NumberFormatException`.

Code that checks if an entry has been made

```
if (investmentTextField.getText().length() == 0)
{
    JOptionPane.showMessageDialog(this,
        "Monthly Investment is "
        + "a required field.\nPlease re-enter.",
        "Invalid Entry", JOptionPane.ERROR_MESSAGE);
    investmentTextField.requestFocusInWindow();
    validData = false;
}
```

Code that checks if an entry is a valid number

```
try
{
    double d = Double.parseDouble(
        investmentTextField.getText());
}
catch (NumberFormatException e)
{
    JOptionPane.showMessageDialog(this,
        "Monthly Investment "
        + "must be a valid number.\nPlease re-enter.",
        "Invalid Entry", JOptionPane.ERROR_MESSAGE);
    investmentTextField.requestFocusInWindow();
    validData = false;
}
```