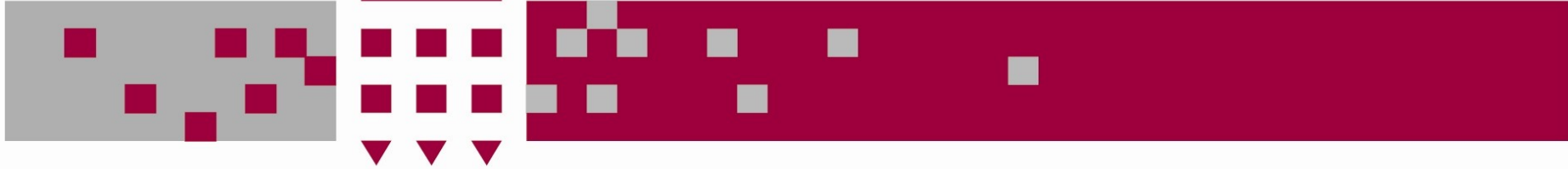


UNIVERSITY OF WESTMINSTER



5COSC019W – Object Oriented Programming Week 8

Dr. Barbara Villarini

b.villarini@westminster.ac.uk



Summary

- Event Handling
- Implementing Event Handler
- File and Stream
- File handling



What is an Event

- GUI components communicate with the rest of the applications through **events**.
- An **event is an object** that describes a state changes in a source
- An event occurs every time the user
 - Types a character or
 - Pushes a mouse button
- The **source** of an event is the component that causes that event to occur.
- The **listener** of an event is an object that receives the event and processes it appropriately.
- That object must:
 - Implement the appropriate interface
 - Be registered as an **event listener** on the appropriate **event source**.

Delegation Event Handling Model

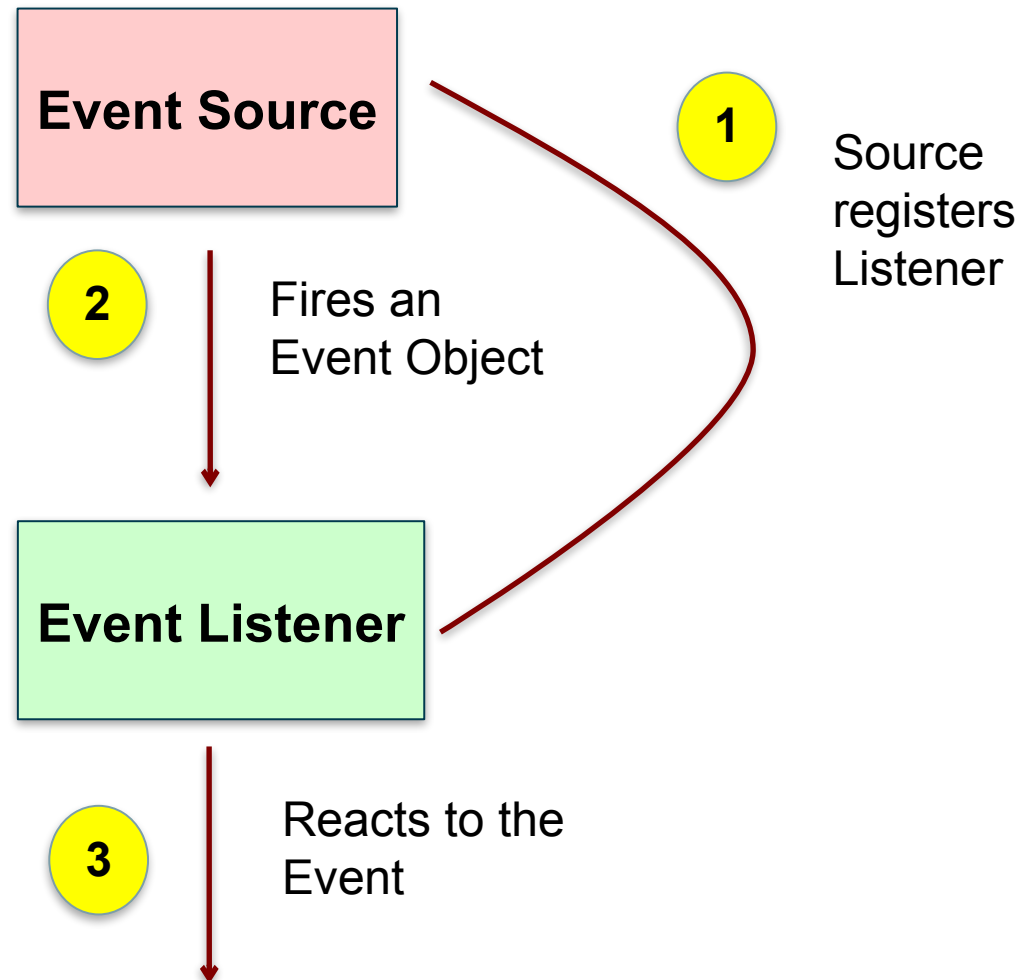
- Three parts
 - Event source
 - GUI component which user interacts with
 - Event object
 - Encapsulates information about event that occurred
 - Event listener
 - Receives event object when notified, then responds
- Programmer must perform two tasks
 1. Register event listener for event source
 2. Implement event-handling method (event handler)



What is an Event Listener?

- An event listener is an **object**
 - It "listens" for events from a specific GUI component (itself an object)
- When an event is generated by the GUI component a method in the listener object is invoked to respond to the event

Recap the Event Handling process





Implement an Event Handler

Every event handler requires three bits of code:

1. Code that specifies that the class either
 1. Implements a listener interface or
 2. Extends a class that implements a listener interface.

```
public class MyClass implements ActionListener {...
```

2. Code that implements the methods in the listener interface.

```
public void actionPerformed(ActionEvent e)  
{ ...//code that reacts to the action... }
```

3. Code that registers an instance of the event handler class as a listener upon one or more components.

```
someComponent.addActionListener(instanceOfMyClass) ;
```



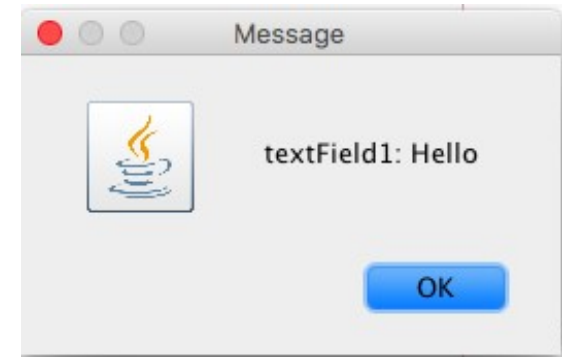
Example using JTextField

- **JTextField**
 - Single-line area in which user can enter text
- **JPasswordField**
 - Extends **JTextField**
 - Hides characters that user enters

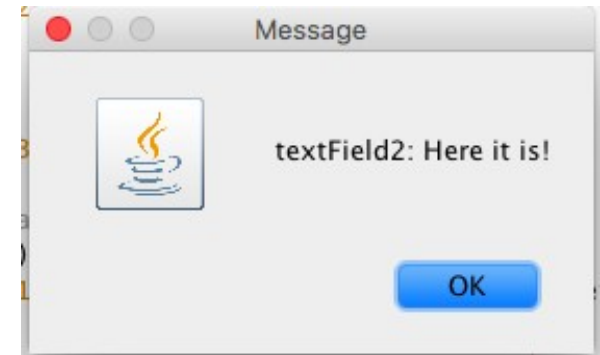
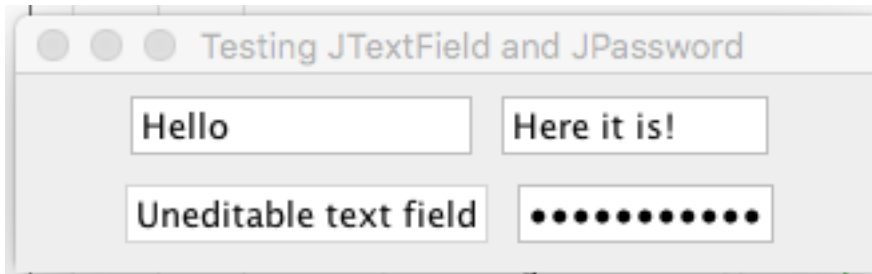
What we visualise:



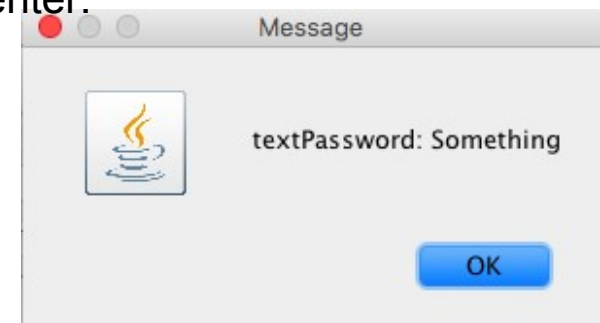
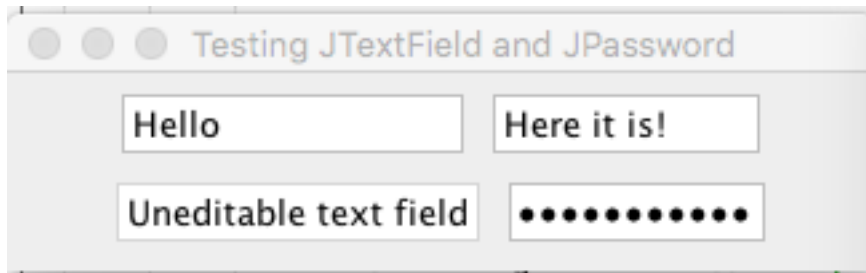
If you enter “Hello!” in the text field 1 and press enter:



If you enter “Here it is!” in the text field 2 and press enter:



If you enter “Something” in the text Password and press enter:





Part (1)

```
package textfieldframe;

import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JOptionPane;

public class TextFieldFrame extends JFrame{
```

Import Swing Packages

← We define our class that extends JFrame

```
    private JTextField textField1;
    private JTextField textField2;
    private JTextField textField3;
    private JPasswordField passwordField;

    // TextFieldFrame constructor adds JTextFields to JFrame
    public TextFieldFrame()
    {
        super("Testing JTextField and JPassword");
        setLayout(new FlowLayout());

        // construct textfield with 10 columns
        textField1 = new JTextField(10);
        add(textField1); // add the textfield to JFrame

        // construct textfield with default text
        textField2 = new JTextField("Enter text here");
        add(textField2); // add the textfield to JFrame
```

Constructor and we define the layout adding the components

Part (2)



```
// construct textfield with default text and 21 columns
textField3 = new JTextField("Uneditable text field");
textField3.setEditable(false);
add(textField3); // add the textfield to JFrame

// construct passwordfield with the default text
passwordField = new JPasswordField("Hidden text");
add(passwordField);

// register event handlers
TextFieldHandler handler = new TextFieldHandler();
textField1.addActionListener(handler);
textField2.addActionListener(handler);
textField3.addActionListener(handler);
passwordField.addActionListener(handler);
} // End TextFieldFrame constructor
```

textField3 will not be editable!

Register event handler to our components:
addActionListener(handler)

```
//private inner class for event handling
private class TextFieldHandler implements ActionListener
{
```

Class for event handling

Override actionPerformed(ActionEvent event)

```
    // process text field events
    public void actionPerformed(ActionEvent event)
    {
        String string = ""; // declare string to display

        // user pressed Enter in JtextField1
        if(event.getSource() == textField1)
            string = String.format("textField1: %s", event.getActionCommand());
```

getSource() is specified by the
EventObject class that ActionEvent is
a child of (via java.awt.AWTEvent)

getActionCommand() gives you a String representing the action command

Part (3)



```
// user pressed Enter in JTextField2
if(event.getSource() == textField2)
    string = String.format("textField2: %s", event.getActionCommand());

// user pressed Enter in JTextField3
if(event.getSource() == textField3)
    string = String.format("textField3: %s", event.getActionCommand());

// user pressed Enter in JTextField passwordField
if(event.getSource() == passwordField)
    string = String.format("textField1: %s", new String(passwordField.getPassword()));

// display the string in a dialog box
JOptionPane.showMessageDialog(null, string);
} // end method action Performed
} // end private inner class

public static void main(String[] args) {

    TextFieldFrame textFieldFrame = new TextFieldFrame();
    textFieldFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    textFieldFrame.setVisible(true);
    textFieldFrame.setSize(325,100);
}
```

Show the string in dialog box

Steps required to set up and event handler

- Several coding steps are required for an application to respond to events
 - Create a class for the event handler
 - Implement an appropriate event-listener interface
 - Register the event handler



Using a Nested Class to Implement an Event Handler

- Top-level classes
 - Not declared within another class
- Nested classes
 - Declared within another class
 - Non-static nested classes are called inner classes
 - Frequently used for event handling
- Note: An inner class is allowed to directly access its top-level class's variables and methods, even if they are private.



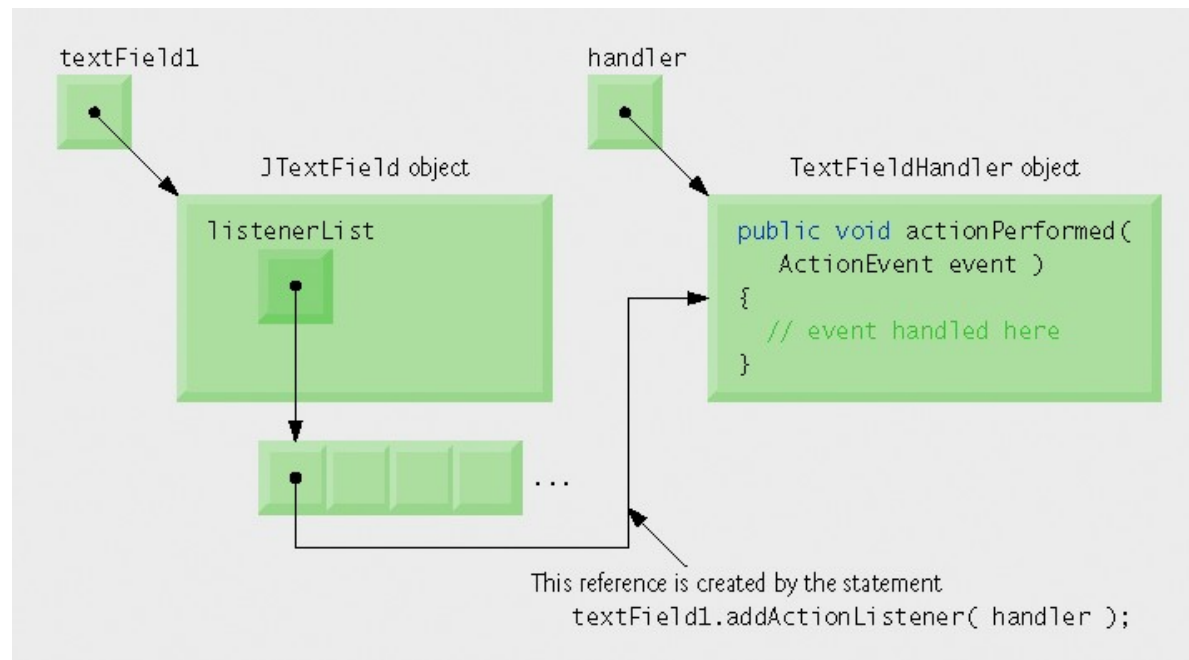
Types of Event

Act causing Event	Listener Type
User clicks a button, presses Enter, typing in text field	ActionListener
User closes a frame	WindowListener
Clicking a mouse button, while the cursor is over a	MouseListener
User moving the mouse over a component	MouseMotionListener
Component becomes visible	ComponentListener
Table or list selection changes	ListSelectionListener



How Event Handler works

- How did the event handler get registered?
- How does the GUI component know to call **actionPerformed** rather than some other event-handling method?
- Every JComponent has instance variable **listenerList**
 - Object of type **EventListenerList**
 - Maintains references to all its registered listeners





Mouse Event Handling

- Every time an action is performed using a mouse a **MouseEvent** object is created
- Handled by **MouseListener** and **MouseMotionListeners**
- **MouseListener** combines the two interfaces
- Interface **MouseWheelListener** declares method **mouseWheelMoved** to handle **MouseWheelEvents**

Mouse Events



MouseListener and MouseMotionListener interface methods

*Methods of interface **MouseListener***

public void mousePressed(MouseEvent event)

Called when a mouse button is pressed while the mouse cursor is on a component.

public void mouseClicked(MouseEvent event)

Called when a mouse button is pressed and released while the mouse cursor remains stationary on a component. This event is always preceded by a call to **mousePressed**.

public void mouseReleased(MouseEvent event)

Called when a mouse button is released after being pressed. This event is always preceded by a call to **mousePressed** and one or more calls to **mouseDragged**.

public void mouseEntered(MouseEvent event)

Called when the mouse cursor enters the bounds of a component.

public void mouseExited(MouseEvent event)

Called when the mouse cursor leaves the bounds of a component.

*Methods of interface **MouseMotionListener***

public void mouseDragged(MouseEvent event)

Called when the mouse button is pressed while the mouse cursor is on a component and the mouse is moved while the mouse button remains pressed. This event is always preceded by a call to **mousePressed**. All drag events are sent to the component on which the user began to drag the mouse.

public void mouseMoved(MouseEvent event)

Called when the mouse is moved when the mouse cursor is on a component. All move events are sent to the component over which the mouse is currently positioned.



Example – Part(1)

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class MouseTrackerFrame extends JFrame{

    private JPanel mousePanel; // panel in which mouse events will occur
    private JLabel statusBar; // label that displays event information

    // MouseTrackerFrame constructor sets up GUI and
    // registers mouse event handlers

    public MouseTrackerFrame()
    {
        super("Demonstarting Mouse Events");

        mousePanel = new JPanel(); // create panel
        mousePanel.setBackground(Color.white);
        add(mousePanel, BorderLayout.CENTER);

        statusBar = new JLabel("Mouse outside JPanel");
        add(statusBar, BorderLayout.SOUTH);
    }
}
```

Create a panel and a label in
MouseTrackerFrame

Part (2)



```
// create and register listener for mouse and mouse motion events
MouseListener handler = new MouseHandler();
mousePanel.addMouseListener(handler);
mousePanel.addMouseMotionListener(handler);
} // end constructor
```

Register listeners for mouse
and mouse motion

```
private class MouseHandler implements MouseListener,
    MouseMotionListener
```

Definition of our class MouseHandler

```
{
    // MouseListener event handlers
    // handle event when mouse released immediatly after press
    public void mouseClicked(MouseEvent event)
    {
        statusBar.setText (String.format("Clicked at [%d, %d]",
            event.getX(), event.getY()));
    }

    // handle event when mouse pressed
    public void mousePressed(MouseEvent event)
    {
        statusBar.setText (String.format("Clicked at [%d, %d]",
            event.getX(), event.getY()));
    }

    //handle event when mouse released after dragging
    public void mouseReleased(MouseEvent event)
    {
        statusBar.setText (String.format("Clicked at [%d, %d]",
            event.getX(), event.getY()));
    }
}
```

Part (3)



```
// handle event when mouse enters area
public void mouseEntered(MouseEvent event)
{
    statusBar.setText (String.format("Clicked at [%d, %d]",
        event.getX(), event.getY()));
    mousePanel.setBackground(Color.green);
}

// handle event when mouse exits area
public void mouseExited(MouseEvent event)
{
    statusBar.setText ("Mouse outside JPanel");
    mousePanel.setBackground(Color.white);
}

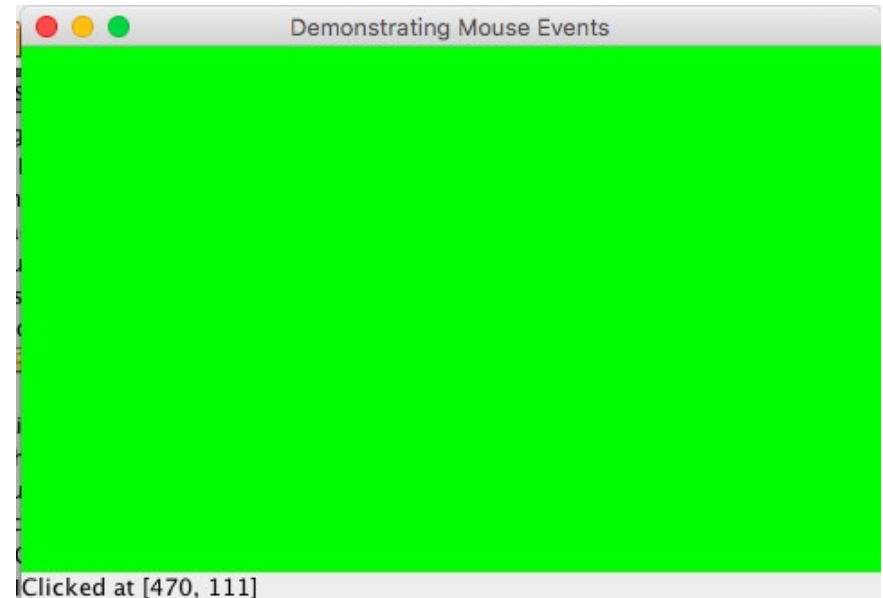
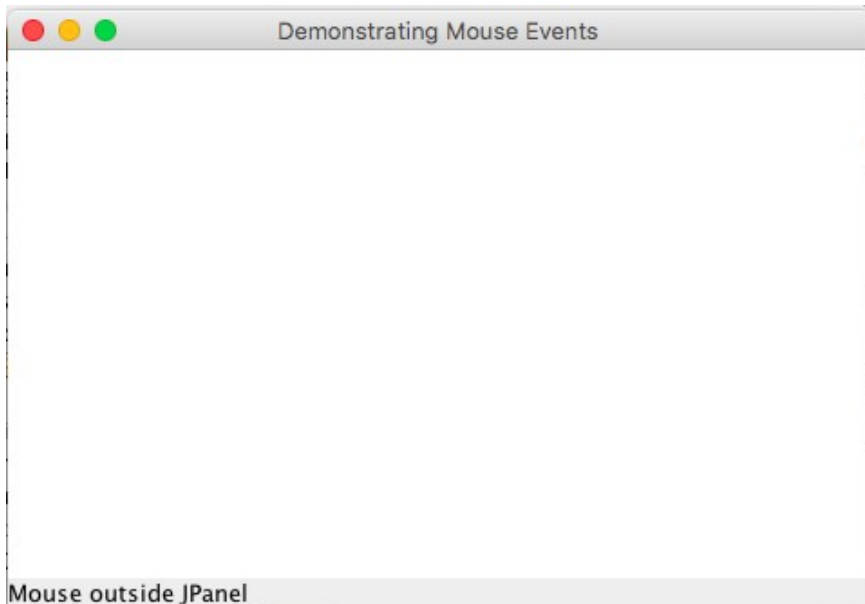
@Override
public void mouseDragged(MouseEvent e) {
    throw new UnsupportedOperationException("Not supported yet."); //To change body of generated
}

@Override
public void mouseMoved(MouseEvent e) {
    throw new UnsupportedOperationException("Not supported yet."); //To change body of generated
}
}
```

Part (4)



```
public static void main(String[] args) {  
  
    MouseTrackerFrame mouseTrackerFrame = new MouseTrackerFrame();  
    mouseTrackerFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    mouseTrackerFrame.setSize(300,100);  
    mouseTrackerFrame.setVisible(true);  
}  
  
}
```





Listener Interface

- Suppose your class directly implements **MouseListener**,
 - Then you must implement all five **MouseListener** methods.
 - Even if you care only about mouse clicks
- Methods for those events you don't care about can have empty bodies.
 - Resulting collection of empty method bodies can make code harder to read and maintain



Adapter Class

- Solution is to use adapter classes
- For example, the **MouseAdapter** class implements the **MouseListener** interface.
- An adapter class implements empty versions of all its interface's methods.
- To use an adapter
 - Create a subclass of it, instead of directly implementing a listener interface.
 - By extending **MouseAdapter**, your class inherits empty definitions of all five of the methods that **MouseListener** contains.

Adapter Classes

- Characteristics of an adapter class
 - Implements interface
 - Provides default implementation of each interface method
 - Used when all methods in interface is not needed

Event-adapter class in <code>java.awt.event</code>	Implements interface
<code>ComponentAdapter</code>	<code>ComponentListener</code>
<code>ContainerAdapter</code>	<code>ContainerListener</code>
<code>FocusAdapter</code>	<code>FocusListener</code>
<code>KeyAdapter</code>	<code>KeyListener</code>
<code>MouseAdapter</code>	<code>MouseListener</code>
<code>MouseMotionAdapter</code>	<code>MouseMotionListener</code>
<code>WindowAdapter</code>	<code>WindowListener</code>



Example - Part (1)

```
import java.awt.BorderLayout;
import java.awt.Graphics;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MouseDetailsFrame extends JFrame{

    private String details; // string representing
    private JLabel statusBar; // JLabel that appears at the bottom of the window

    // constructor sets title bar String and register mouse listener
    public MouseDetailsFrame()
    {
        super("Mouse clicks and buttons");

        statusBar = new JLabel ("Click the mouse");
        add(statusBar, BorderLayout.SOUTH);
        addMouseListener(new MouseClickHandler()); // add handler
    }
}
```

Part (2)



```
// inner class to handle mouse event
private class MouseClickHandler extends MouseAdapter
{
    public void mouseClicked(MouseEvent event)
    {
        int xPos = event.getX(); // get pos x of the mouse
        int yPos = event.getY(); // get pos y of the mouse

        details = String.format("Clicked %d time(s)",
            event.getClickCount());

        if(event.isMetaDown()) // right mouse button
            details += " with right mouse button";
        else if (event.isAltDown()) // middle mouse button
            details += " with center mouse button";
        else
            details += " with left mouse button";

        statusBar.setText(details);
    }
}
```

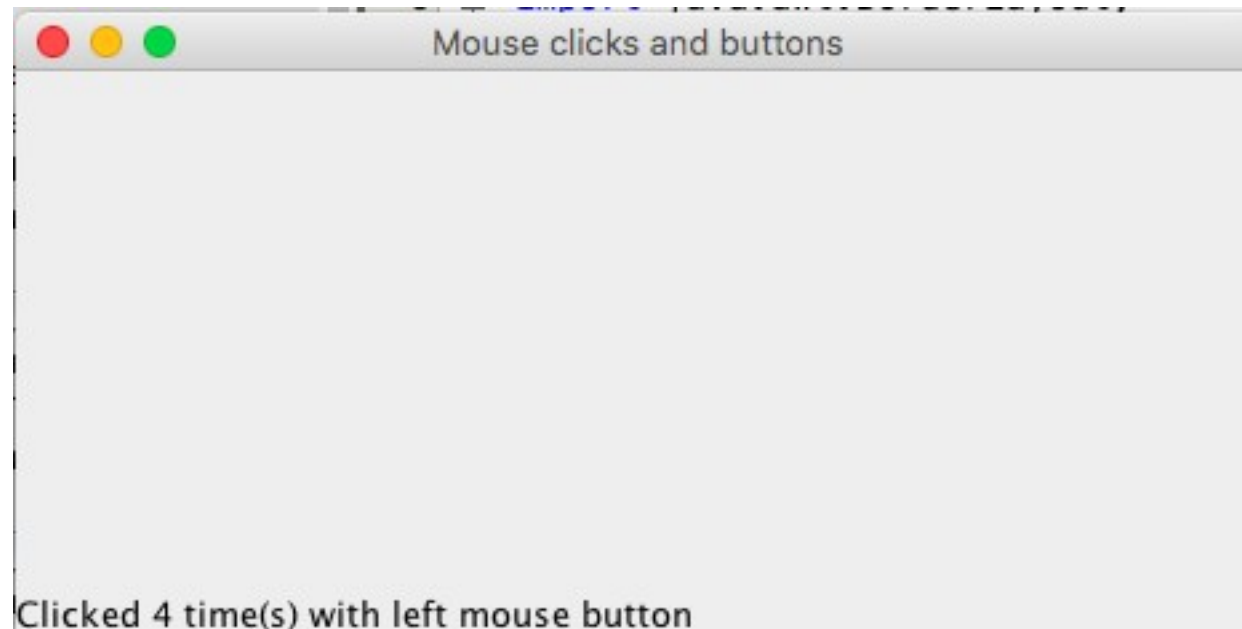
You don't need to implement all the others methods!

Check if the event is right, left or center clicked



Part (3)

```
public static void main(String[] args) {  
  
    MouseDetailsFrame mouseDetailsFrame = new MouseDetailsFrame();  
    mouseDetailsFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    mouseDetailsFrame.setSize(400, 150);  
    mouseDetailsFrame.setVisible(true);  
}
```



FILE HANDLING



Introduction

- Storage of data in variables and arrays is temporary
 - when the program is done running or when computer is turned off The data is gone!
- Files used for long-term retention of large amounts of data, even after the programs that created the data terminate
- Persistent data – exists beyond the duration of program execution
- Files stored on secondary storage devices (hard disks, CD-ROMs, etc.)
- Requirements information stored in secondary memory
 - can be retrieved in the future
 - kept separate from other documents, programs, etc.



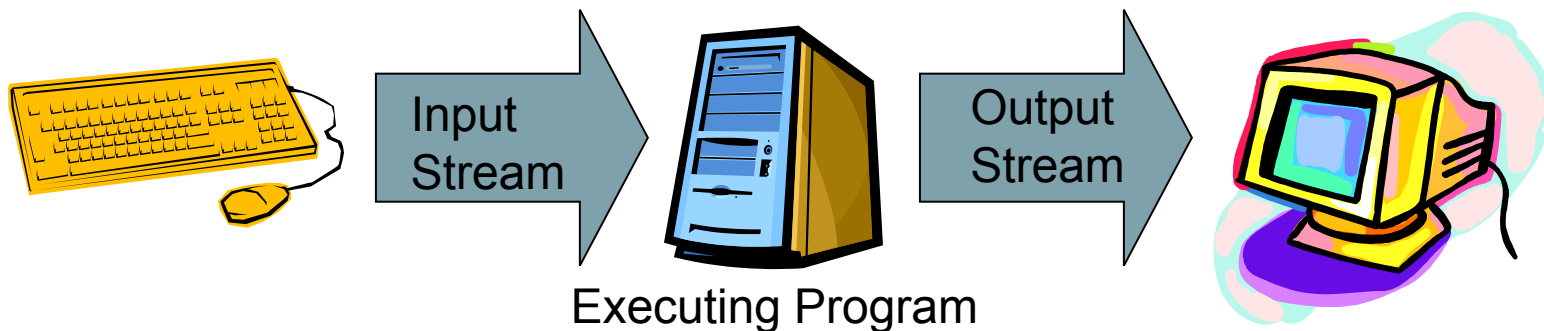
Intro Example: Stock Market Analysis

- Problem: large amounts of stock prices/indices are recorded and processed
 - taken every day
 - stored in a text file **dataStockMarket.dat**
- Analyzing this data means find minimum, maximum, and average of the data
- Data must be read from file, statistics calculated, results written to a text output file



Java's I/O System

- All input and output in Java is accomplished by classes called **streams**
- The purpose of the stream abstraction is to keep program code independent from physical devices.





Type of Stream

- There are 2 kinds of streams
 - byte streams: stores data in binary format
 - character streams: stores data as a sequence of characters
- Java opens file by creating an object and associating a stream with it
- Java libraries define a number of stream classes (`java.io` classes):
 - **Reader** and **Writer** for dealing with character formatted data (e.g., unicode characters).
 - **InputStream** and **OutputStream** for dealing with unformatted data (bytes)

Note that the classes **InputStream**, **OutputStream**, **Reader**, and **Writer** are *abstract* classes.



Predefined Stream

- Three stream objects are automatically created for every application: `System.in`, `System.out`, and `System.err`.
- `System.in`
 - **`InputStream`** object, usually for the keyboard
- `System.out`
 - a buffered **`PrintStream`** object, usually the screen or an active window
- `System.err`
 - an unbuffered **`PrintStream`** object usually associated with the screen or console window



File Class

- Useful for retrieving information about files and directories from disk
- Object of class File doesn't open the actual files/directories.
- It provides methods to operate on the files/directories named.

Create a File object

Creating a File object specifies name/path only:

```
File myDirectory = new File("/Desktop/directory");
```

```
File myFile = new File("/Desktop/directorydataset.txt");
```

```
File myFile = new File("/Desktop/directory", "dataset.txt");
```

```
File myFile = new File(myDirectory, "dataset.txt");
```



Methods to manipulate Files and Directory

- `exists()`
- `isDirectory()`
- `isFile()`
- `canRead()`
- `canWrite()`
- `isHidden()`
- `getName()`
- `getPath()`
- `getAbsolutePath()`
- `getParent()`
- `list()`
- `length()`
- `renameTo(newPath)`
- `delete()`
- `mkdir()`
- `createNewFile()`



Listing files

```
import java.io.*;
import java.util.*;

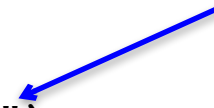
public class ListFile {

    public static void main(String[] args) {
        File currentDirectory = new File(".");

        String[] contents = currentDirectory.list();

        for (int i = 0; i < contents.length; i++)
        {
            System.out.println(contents[i]); }
    }
}
```

It is considering the current directory, but you could pass any path



FileReader

- FileReader opens file for reading and throws exception if open fails.

Constructors:

```
FileReader reader = new FileReader("filename");  
FileReader reader = new FileReader(fileObject);
```

- Provides basic set of methods for reading:
 - read character (mapped to character set).
 - read array of characters.
- Also provides method to close stream.



FileWriter

- FileWriter opens file for writing and throws exception if open fails
- Constructors:

```
FileWriter writer = new FileWriter("filename");  
FileWriter writer = new FileWriter(fileObject);
```
- Provides basic set of write methods:
 - write character.
 - write array of characters.
 - write String.
- Also provides method to close stream.
- If you don't close the file some data may not be written to file.



Remember

- All FileReader/Writer methods throw exceptions. You should use try/catch blocks or write methods with a throws declaration.
- If you open an existing file for writing, you *delete* existing contents, unless **append mode** is selected:

```
FileWriter writer = new FileWriter(fileObject, true) ;
```

boolean append_to_file





Example – Copying a text file

```
import java.io.*;

public class CopyTextFile {

    public static void main(String[] args) throws IOException {

        File inputFile = new File("text.txt");
        File outputFile = new File("text_copy.txt");

        FileReader in = new FileReader(inputFile);
        FileWriter out = new FileWriter(outputFile);

        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
        in.close();
        out.close();
    }
}
```

Copy one character at a time. read() returns -1 when no more data.

Reading

Writing

Reading/writing files using Byte Streams

- Java methods to read and write bytes from and to a file:
 - `FileInputStream`
 - `FileOutputStream`
- Create an object of one of these classes to open a file
- Specify the name of the file as argument to the constructor.
- Once the file is open you can read/write it



Example – copy a file as bytes

```
import java.io.*;

public class CopyBytesFile {
    public static void main(String[] args) throws IOException {

        File inputFile = new File("Text.txt");
        File outputFile = new File("Text_copy.txt");

        FileInputStream in = new FileInputStream(inputFile);
        FileOutputStream out = new
        FileOutputStream(outputFile);

        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
        in.close();
        out.close();
    }
}
```

Type int used to store byte.

Copy one byte at a time.

BufferedReader and BufferedWriter Class

- An object of these classes reads/writes an entire line of text into a buffer
- The constructor take as parameter an object of type Reader
- It provides a **readLine** method to read complete line into a String

```
BufferedReader in = new BufferedReader(new FileReader("data.txt"));  
String s = in.readLine();
```

- String can then be converted to other types (int, double, etc.)



Example – Copying a file

```
import java.io.*;

public class ReadString{
    public static void main(String[] args) throws IOException {

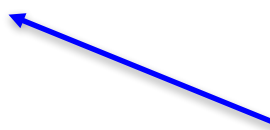
        FileReader in = new FileReader("File1.txt");
        BufferedReader br = new BufferedReader(in);

        FileWriter out = new FileWriter("File2.txt", true);
        BufferedWriter bw = new BufferedWriter(out);

        String line = br.readLine();
        while(line != null){
            bw.write(line);
            bw.newLine();
            line = br.readLine();
        }
        in.close();
        br.close();
        bw.close();
        out.close();

    }
}
```

**Use readLine()
from BufferedReader
to read a line of
text**





Other Reader/Writers

- **StringReader, StringWriter**: read/write to/from strings rather than files or writers.
- **InputStreamReader**: Convert from stream to reader.
- **OutputStreamWriter**: Convert from writer to stream.

Plus a family of `InputStream` and `OutputStream` classes.



Summary

- We saw file handling in Java
- Remember that it is important to save data in files and permanent storage.
- Reading and writing data from and to a file is a must for a good developer!