Object
Class
Inheritance
Polymorphism
Encapsulation
Data Abstraction
OOps concept in Java

CS20004: Object Oriented Programming using Java

Lec- 26

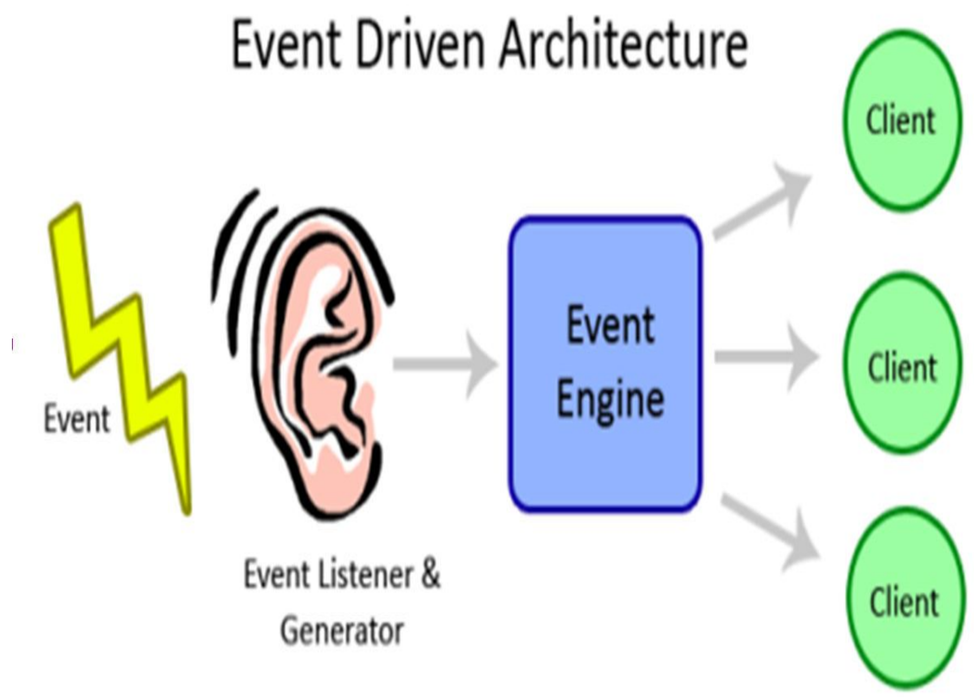
In this Discussion . . .

- User events and callbacks
  - Event-driven programming
  - Registering listeners to handle events
  - Events and Event Handler
  - User Interaction with a GUI through Events
  - EventLoops
  - Handling ActionEvents with ActionListeners

Event Driven Programming

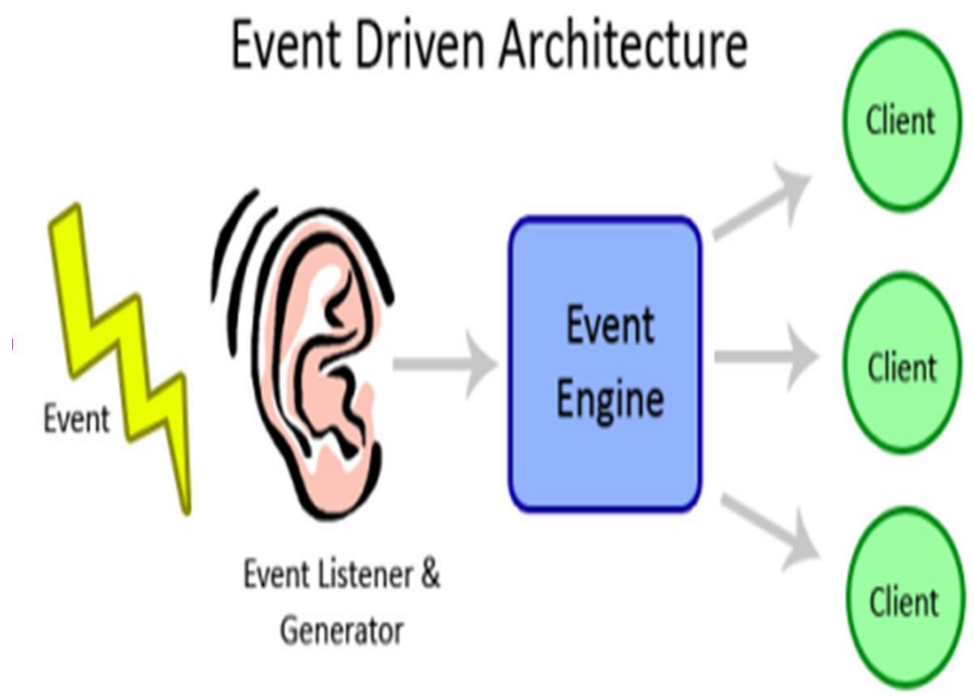
Event Driven Programming:

- A style of coding where a program's overall flow of execution is dictated by events



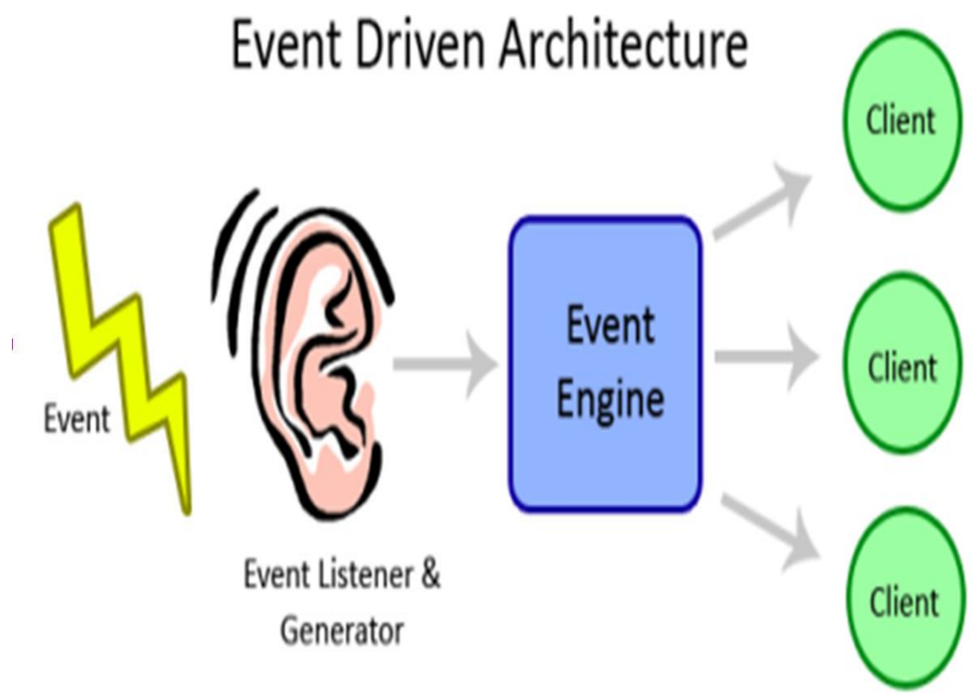
Event Driven Programming

- The program loads, then waits for user input events.
- As each event occurs, the program runs particular code to respond.
- The overall flow of what code is executed is determined by the series of events that occur.



Event Driven Programming

- Contrast with application- or algorithm-driven control where program expects input data in a predetermined order and timing
    - Typical of large non-GUI applications like web crawling, payroll, batch simulation



Graphical Events

- **Events:** An object that represents a user's interaction with a GUI component; can be "handled" to create interactive components
- **Listener:** An object that waits for events and responds to them.
  - To handle an event, attach a *listener* to a component.
  - The listener will be notified when the event occurs (For ex- button click).

What happens when a button is pressed

$$\begin{tikzcd}
\boxed{\text{JButton}} \arrow[r] & \boxed{\text{ActionEvent}} \arrow[r] & \boxed{\text{ActionListener}}
\end{tikzcd}$$

What happens when any event occurs

$$\begin{tikzcd}
\boxed{\text{event source}} \arrow[r] & \boxed{\text{event object}} \arrow[r] & \boxed{\text{event listener}}
\end{tikzcd}$$

Events and Event Handler

- In our previous class, we saw some examples as to how to create a GUI with various types of components.
- However, none of the components on the window seem to respond to the user interactions.
- In order to get the interface to "work" we must make it respond appropriately to all user input such as clicking buttons, typing in text fields, selecting items from list boxes etc… **To do this, we must investigate events.**

What is an event ?

- An ***event*** is something that happens in the program based on some kind of triggering input
- typically caused (i.e., generated) by user interaction (e.g., mouse press, button press, selecting from a list etc...)
    - the component that caused the event is called the ***source***.
- can also be generated internally by the program

How are Events Used in JAVA ?

- Events are objects. So, each type of event is represented by a distinct class (similar to the way exceptions are distinct classes).
- low-level events represent window-system occurrences or low-level input such as mouse and key events and component, container, focus, and window events.
- some events may be ignored, some may be ***handled***. We will write ***event handlers*** which are known as ***listeners***.

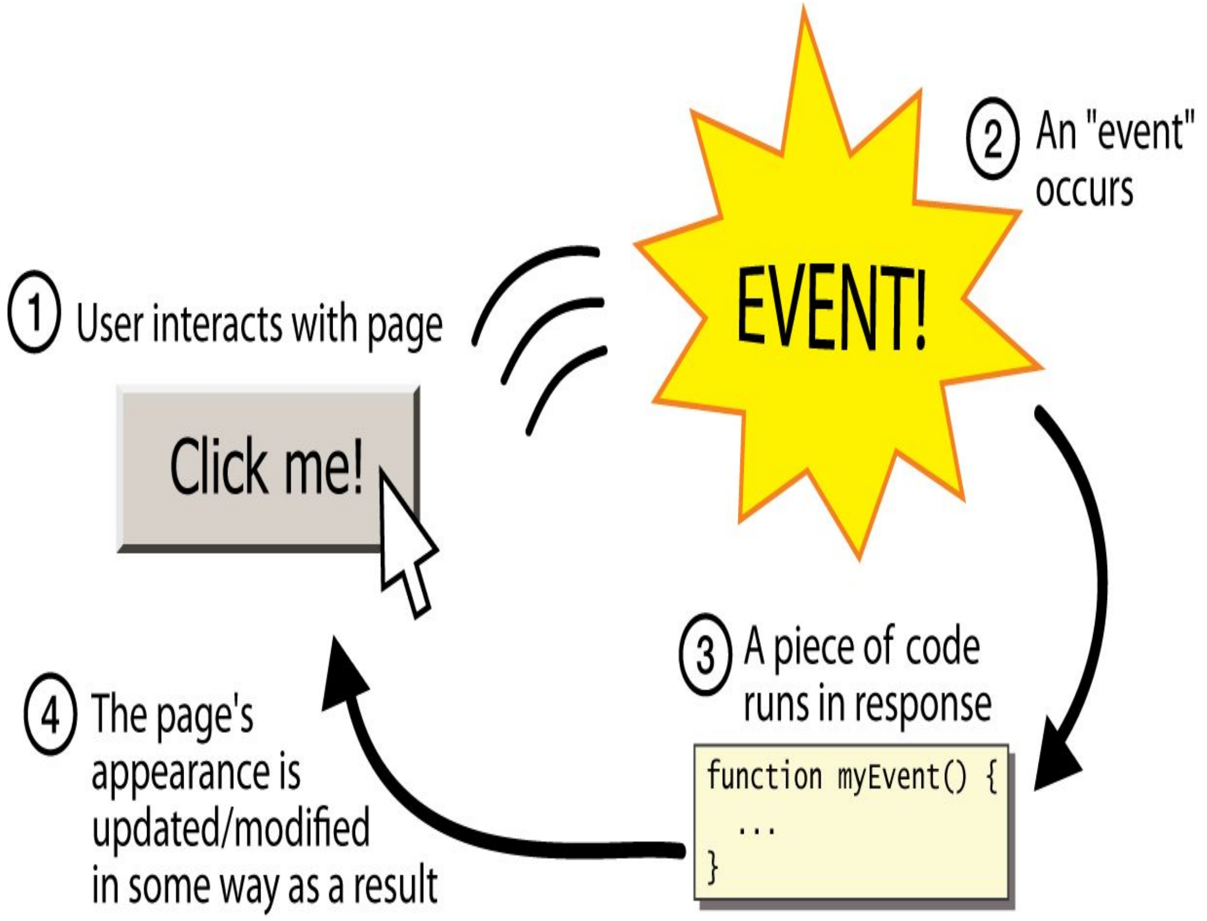
Nothing happens in your program UNLESS an event occurs.
JAVA applications are thus considered to be event-driven

Event Driven Programming

The main body of the program is an event loop.

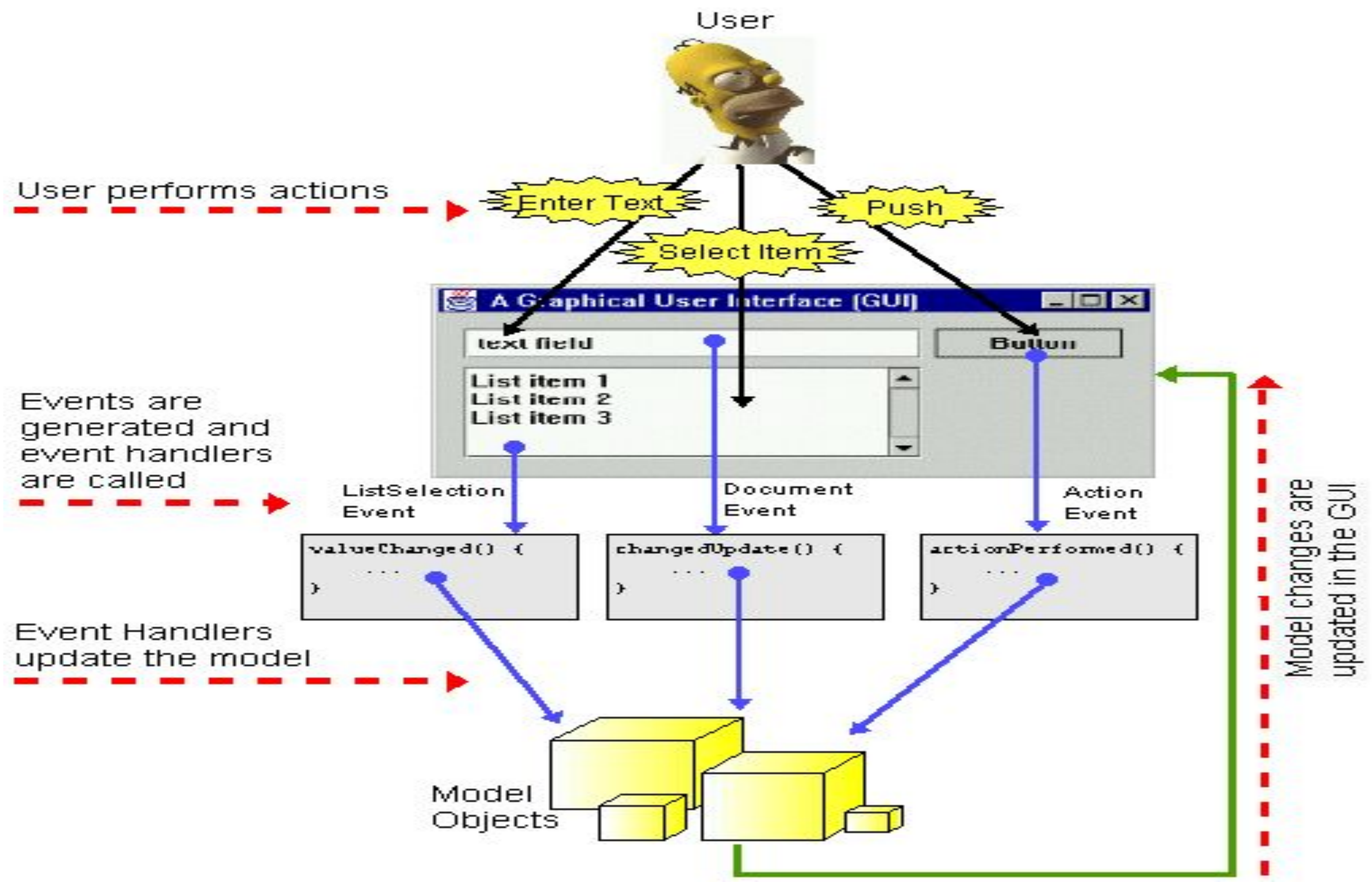
Abstractly:

```
do
{
    e = getNextEvent();
    process event e;
}while (e != quit);
```


1 User interacts with page
Click me!
EVENT!
2 An "event" occurs
3 A piece of code runs in response
function myEvent() {
...
}
4 The page's appearance is updated/modified in some way as a result

User Interaction with a GUI through Events

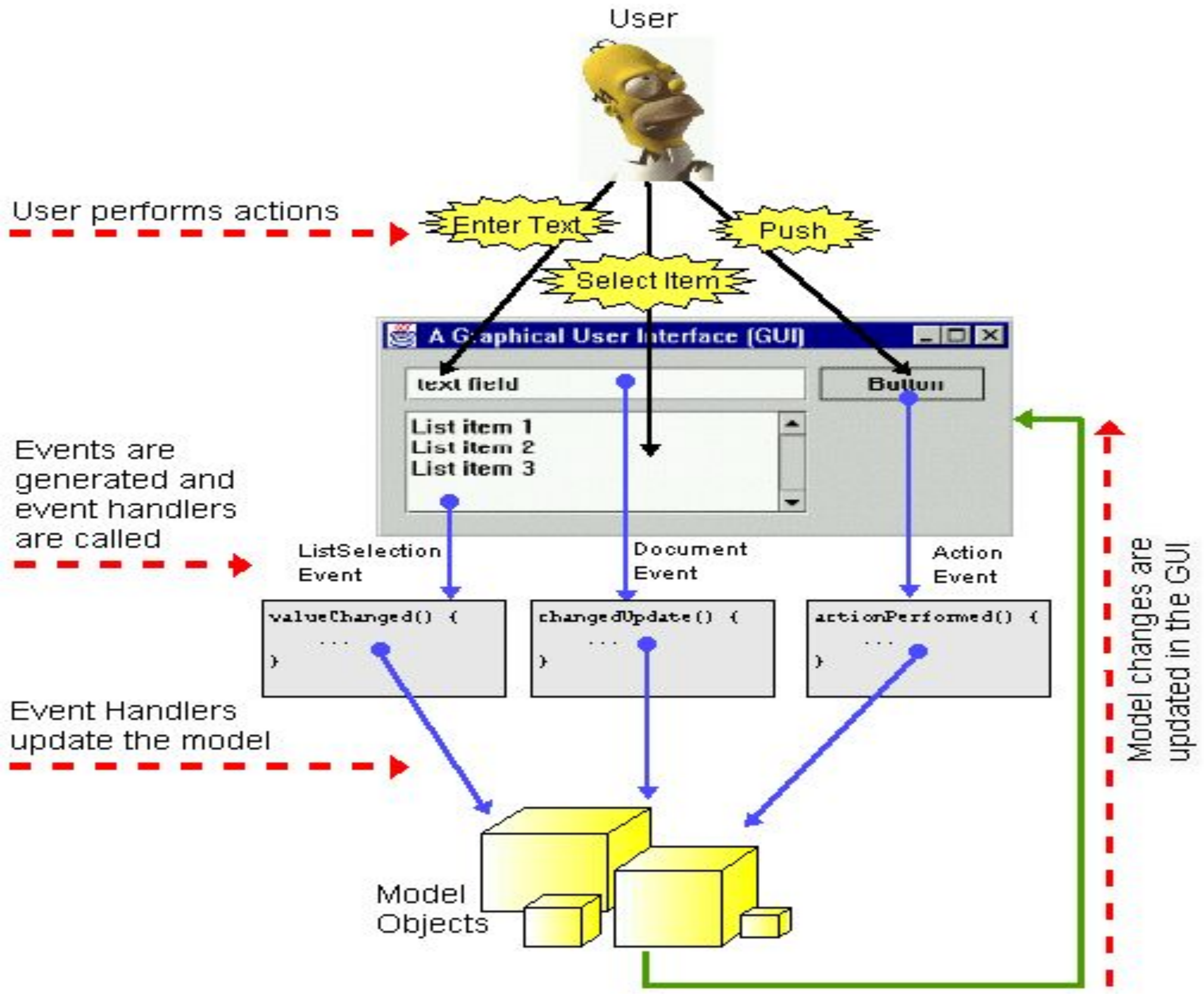
The picture on the right describes the process of user interaction with a GUI through events



User Interaction with a GUI through Events

Basically...here's how it works:

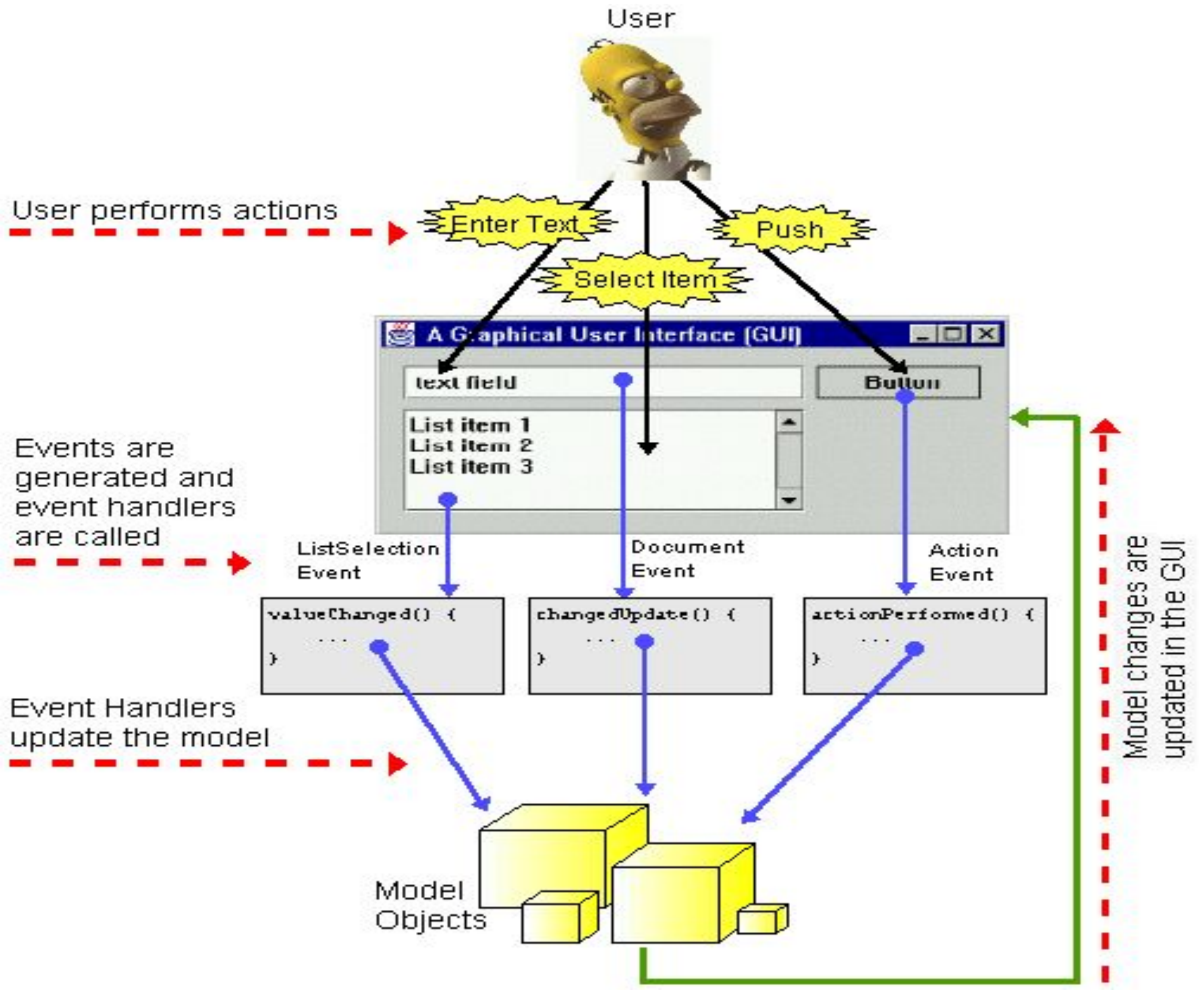
1. The user causes an event (e.g., click button, enter text, select list item etc...)
2. The JAVA virtual machine invokes (i.e., triggers) the appropriate event handler (if it has been implemented and registered).
   - This invocation really means that a **method is called** to handle the event.



User Interaction with a GUI through Events

Basically...here's how it works: (Contd.)

3. The code in the event handling method changes the model in some way.
4. Since the model has changed, the interface will probably also change and so components should be updated



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.

Java Event Model

- 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**.

User Interaction with a GUI through Events

- Notice that JAVA itself waits for the user to initiate an action that will generate an event.
- This is similar to the situation of a cashier waiting for customers … the cashier does nothing unless an event occurs. Here are some events which may occur, along with how they may be handled:
    - a customer arrives - employee wakes up and looks sharp
    - a customer asks a question - employee gives an answer
    - a customer goes to the cash to buy - employee initiates sales procedure
    - time becomes 6:00pm - employee goes home

User Interaction with a GUI through Events

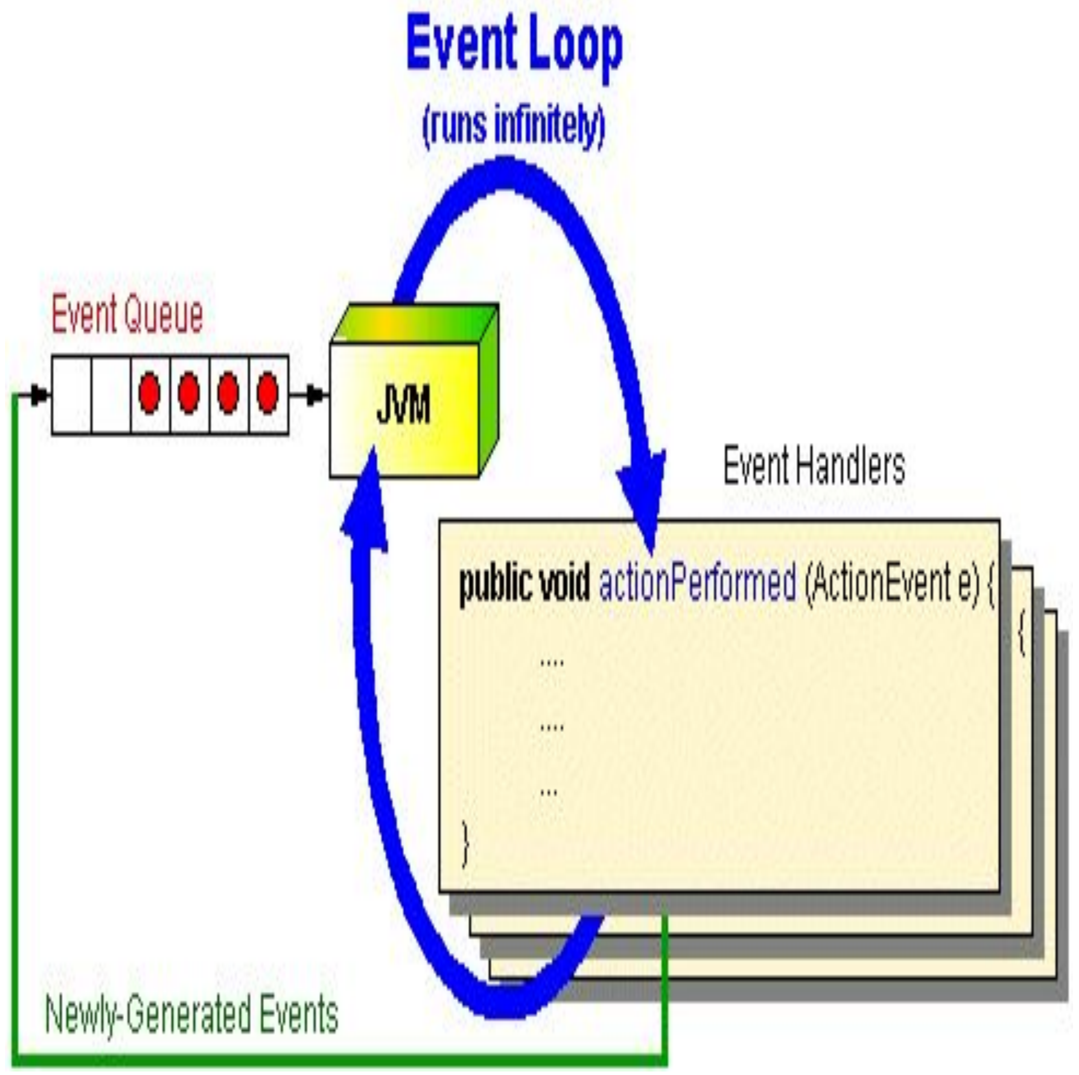
- JAVA acts like this employee who waits for a customer action.
  - JAVA does this by means of something called an ***EventLoop***.

An ***EventLoop*** is an endless loop that waits for events to occur:

- Events are queued (lined up on a first-come-first-served basis) in a buffer
- Events are handled one at a time by an event handler (i.e., code that evaluates when event occurs)
- Everything you want done in your application MUST go through this loop

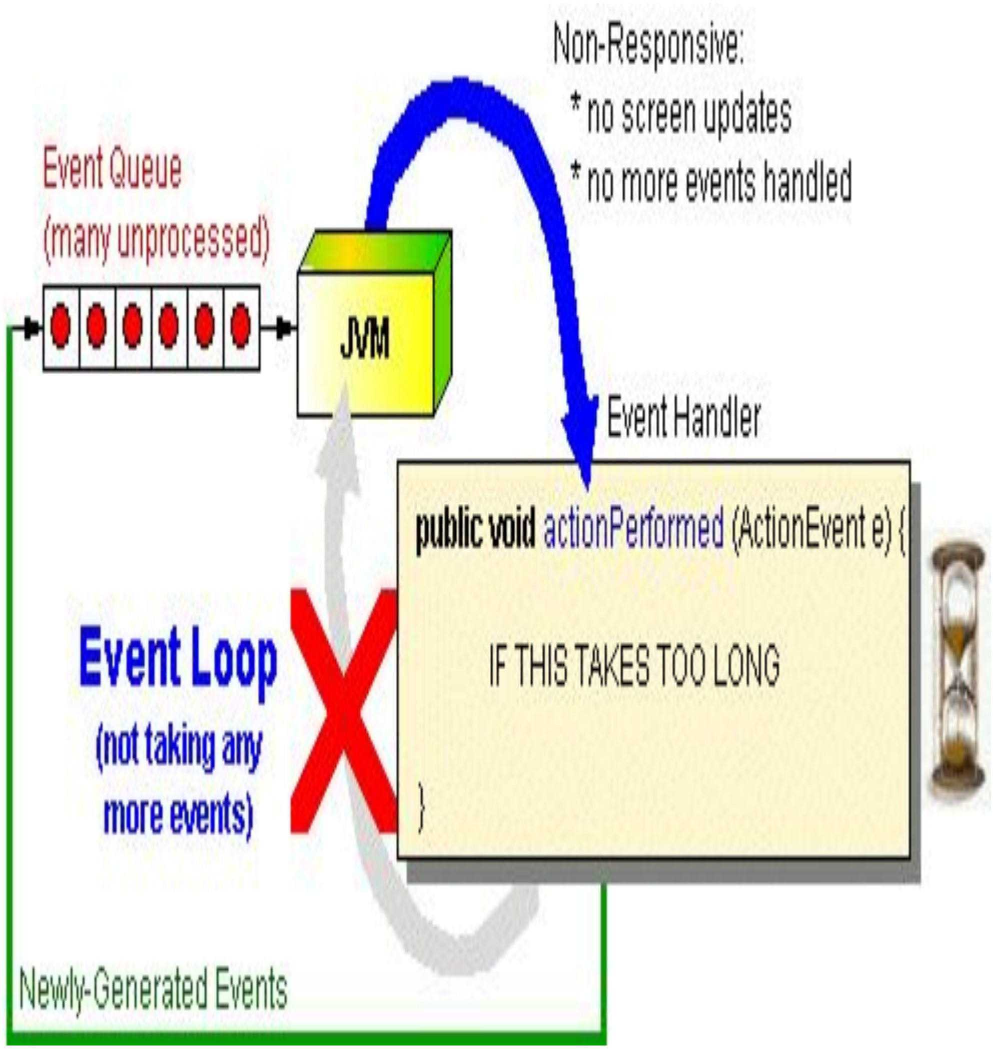
EventLoop

- Notice that incoming events (i.e., customers/clients) are stored in the event queue in the order that they arrive.
- As we will see later, events **MUST** be handled by your program.
- The JVM spends all of its time taking an event out of the queue, processing it and then going back to the queue for another.



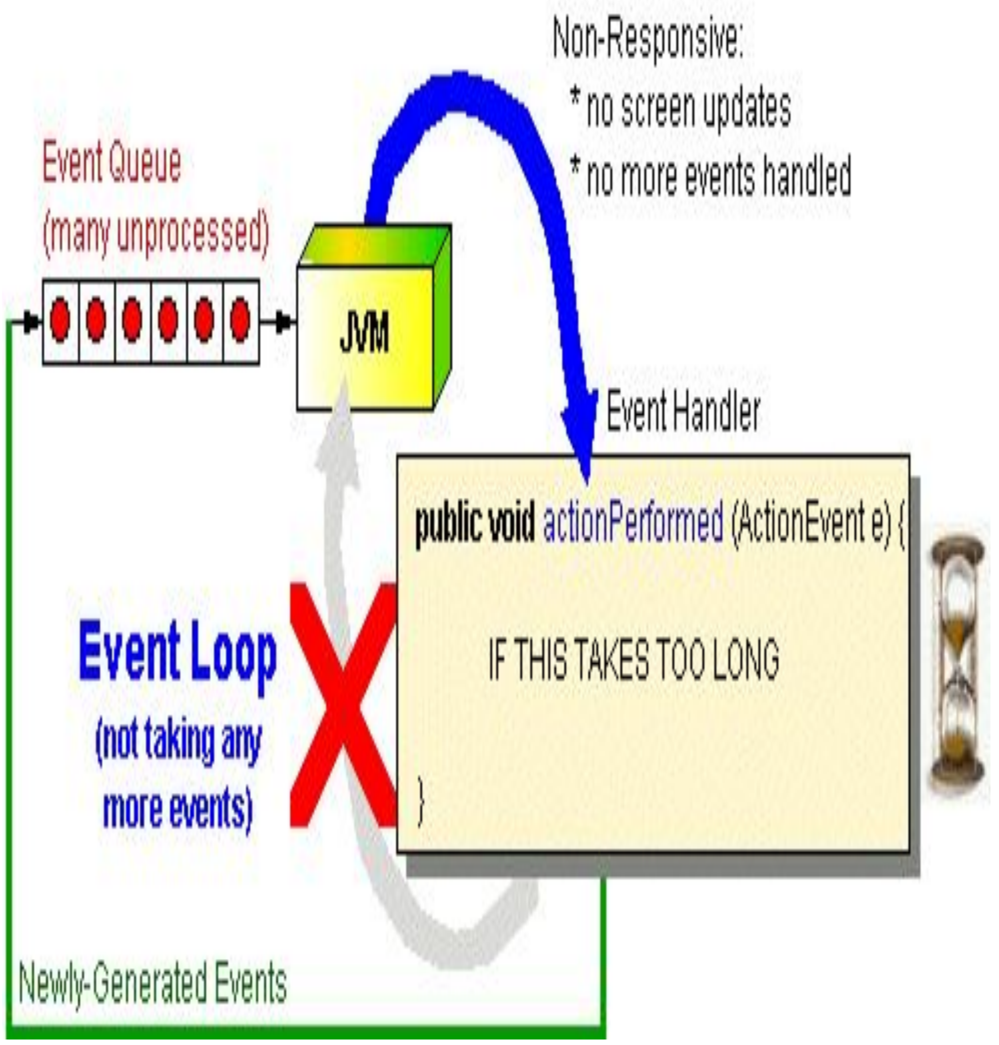
EventLoop (Contd.)

- While each event is being handled, JAVA is unable to process any other events.
- You MUST be VERY careful to make sure that your event handling code does not take too long.



EventLoop (Contd.)

- Otherwise the JVM will not take any more events from the queue.
- This makes your application seem to "hang" so that the screen no longer updates, and all buttons, window components seem to freeze up !!!



EventLoop

- In a way, the JVM event loop acts as a *server*.
- It serves (or handles) the incoming events one at a time on a first-come-first-served basis.
- So when an event is generated, JAVA needs to go to the appropriate method in your code to handle the event.

EventLoop

- In a way, the JVM event loop acts as a *server*.
- It serves (or handles) the incoming events one at a time on a first-come-first-served basis.
- So when an event is generated, JAVA needs to go to the appropriate method in your code to handle the event.

- **How does JAVA know which method to call ?**
- We will ***register*** each event-handler so that JAVA can call them when the events are generated.
  These event-handlers are called *listeners* (or *callbacks*)

Listener

A *listener:*

- acts on (i.e., handle) the event notification.
- must be registered so that it can be notified about events from a particular source.
- can be an instance of any class (as long as the class implements the appropriate listener interface)

Listener

A *listener:*

- acts on (i.e., handle) the event notification.
- must be registered so that it can be notified about events from a particular source.
- can be an instance of any class (as long as the class implements the appropriate listener interface)

So… when creating a GUI, we must:

- decide what types of events we want to handle
- inform JAVA which ones we want to handle by registering the event handlers (i.e., the listeners)
- write the event handling code for each event

Listeners

- You should understand now that when the user interacts with your user interface, some events will be generated automatically by JAVA
- There are many types of events that can occur, and we will choose to respond to some of them, while ignoring others.
- The JAVA VM is what actually generates the events, so we will have to "speak JAVA's language" in order to understand what the event means.
- In fact, to handle a particular event, we will have to write a particular method with a predefined name (chosen by JAVA).

Listeners

- Following represents the **list of commonly used types of events**:

Action Events	clicking buttons, selecting items from lists etc....
Component Events	changes in the component's size, position, or visibility
Focus Events	gain or lose the ability to receive keyboard input
Key Events	key presses; generated only by the component that has the current keyboard focus.
Mouse Events	mouse clicks and the user moving the cursor into or out of the component's drawing area.
Mouse Motion Events	changes in the cursor's position over the component
Container Events	component has been added to or removed from the container

Listeners

- Here are a couple of the "less used" types of events in JAVA

Ancestor Events	containment ancestors is added to or removed from a container, hidden, made visible, or moved.
Property Change Events	part of the component has changed (e.g., color, size,...).

- For each event type in JAVA, there are defined *interfaces* called **Listeners** which we must implement.

Each listener interface defines one or more methods that MUST be implemented in order for the event to be handled properly.

Listeners

- There are many types of events that are generated and commonly handled.
- In our next slide, we present a table of some of the common events.
- The table gives a short description of:
  - when the events may be generated,
  - gives the interface that must be implemented by you in order for you to handle the events and
  - finally lists the necessary methods that need to be implemented.
- **Note**, for a more complete description of these events, listeners and their methods, see the JAVA API specifications.

Listeners

Event Type or Event Object	Generated By	Listener Interface	Methods that "YOU" must Write
ActionEvent	a button was pressed, a menu item selected, pressing enter key in a text field or a timer event was generated	ActionListener	actionPerformed(ActionEvent e)
ChangeEvent	value of a component such as a JSlider has changed	ChangeListener	stateChanged(ChangeEvent e)
ItemEvent	caused via a selection or deselection of something from a list, a checkbox or a toggle button	ItemListener	itemStateChanged(ItemEvent e)

WindowEvent	open/close, activate/deactivate, iconify/deiconify a window	WindowListener	windowOpened(WindowEvent e)
			windowClosed(WindowEvent e)
			windowClosing(WindowEvent e)
			windowActivated(WindowEvent e)
			windowDeActivated(WindowEvent e)
			windowIconified(WindowEvent e)
			windowiconified(WindowEvent e)

MouseEvent	pressing/releasing/clicking a mouse button, moving a mouse onto or away from a component	MouseListener	mouseClicked(MouseEvent e)
			mouseEntered(MouseEvent e)
			mouseExited(MouseEvent e)
			mousePressed(MouseEvent e)

ContainerEvent	Adding or removing a component to a container such as a panel	ContainerListener	componentAdded(ContainerEvent e)
			componentRemoved(ContainerEvent e)

- So, if you want to handle a button press in your program, you need to write an **actionPerformed()** method:

```
public void actionPerformed(ActionEvent e)
{
    //Do what needs to be done when the button is clicked
}
```

- If you want to have something happen when the user presses a particular key on the keyboard, you need to write a **keyPressed()** method:

```
public void keyPressed(KeyEvent e)
{
    //Do what needs to be done when a key is pressed
}
```

Registering Listeners

- Once we decide which events we want to handle and then write our event handlers, we then need to register the event handler.
- This is like "plugging-in" the event handler to our window.
- In general, many applications can listen for events on the same component. So when the component event is generated, JAVA must inform everyone who is listening.

Registering Listeners

- Once we decide which events we want to handle and then write our event handlers, we then need to register the event handler.
- This is like "plugging-in" the event handler to our window.
- In general, many applications can listen for events on the same component. So when the component event is generated, JAVA must inform everyone who is listening.

- We must therefore tell the component that we are listening for (or waiting for) an event.
- If we do not tell the component, it will not notify us when the event occurs (i.e., it will not call our event handler).

Registering Listeners

- Once we decide which events we want to handle and then write our event handlers, we then need to register the event handler.
- This is like "plugging-in" the event handler to our window.
- In general, many applications can listen for events on the same component. So when the component event is generated, JAVA must inform everyone who is listening.

- We must therefore tell the component that we are listening for (or waiting for) an event.
- If we do not tell the component, it will not notify us when the event occurs (i.e., it will not call our event handler).

- So, when a component wants to signal/fire an event, it sends a specific message to all listener objects that have been registered (i.e., anybody who is "listening").
- For every event, therefore, that we want to handle, we must write the listener (i.e., event handler) and also register that listener.

Example Scenario : Olympic Games

- To help you understand why we need to do this, think of the Olympic games.

  - There are various events in the Olympics and we may want to participate (i.e., handle) a particular event.
  - ***Our training and preparation for the event*** is ***like writing the event handler code which defines what we do when the event happens.***

Example Scenario : Olympic Games

- To help you understand why we need to do this, think of the Olympic games.

- There are various events in the Olympics and we may want to participate (i.e., handle) a particular event.
- ***Our training and preparation for the event*** is ***like writing the event handler code which defines what we do when the event happens.***

- **But**, **we don't get to participate in the Olympic games unless we "sign-up" (or register) for the events** ... **right ?**
- ***So registering our event handlers*** is like ***joining JAVA's sign-up list so that JAVA informs us when the event happens and then allows our event handler to participate when the event occurs***.

Example Scenario : Olympic Games (Events Registration)

- To register for an event (i.e., enable it), we need to merely add the listener (i.e., your event handler) to the component by using an **addXXXListener()** method (where XXX depends on the type of event to be handled).

Here are some examples:

- Button.addActionListener(ActionListener anActionListener);
  aJPanel.addMouseListener(MouseListener aMouseListener);
  aJFrame.addWindowListener(WindowListener aWindowListener);

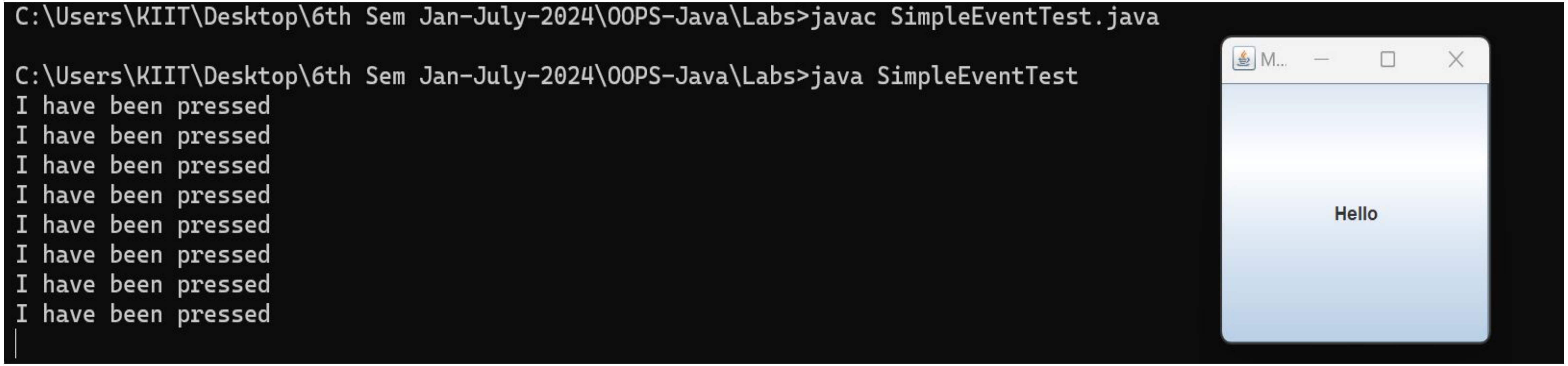
Here anActionListener, aMouseListener and aWindowListener can be instances of any class that implements the specific Listener interface

Handling a Button Press Event

- So, for example, if you wanted to have your application handle a button press, you can make your application itself be the ActionListener as follows:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class SimpleEventTest extends JFrame implements
ActionListener
{
        public SimpleEventTest(String name)
        {
                super(name);
                JButton aButton = new JButton("Hello");
                add(aButton);
                // Plug-in the button's event handler
                aButton.addActionListener(this);
setDefaultCloseOperation(EXIT_ON_CLOSE);
                setSize(200, 200);
        }
        // Must write this method now since SimpleEventTest
//implements the ActionListener interface
        public void actionPerformed(ActionEvent e)
        {
                System.out.println("I have been pressed");
        }
        public static void main(String[] args)
        {
JFrame frame = new SimpleEventTest("Making a Listener");
                frame.setVisible(true);
        }
}
```

- So, for example, if you wanted to have your application handle a button press, you can make your application itself be the ActionListener as follows:



```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac SimpleEventTest.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java SimpleEventTest
I have been pressed
I have been pressed
I have been pressed
I have been pressed
I have been pressed
I have been pressed
I have been pressed
I have been pressed
```

Example Scenario : Olympic Games (Events Unregistration)

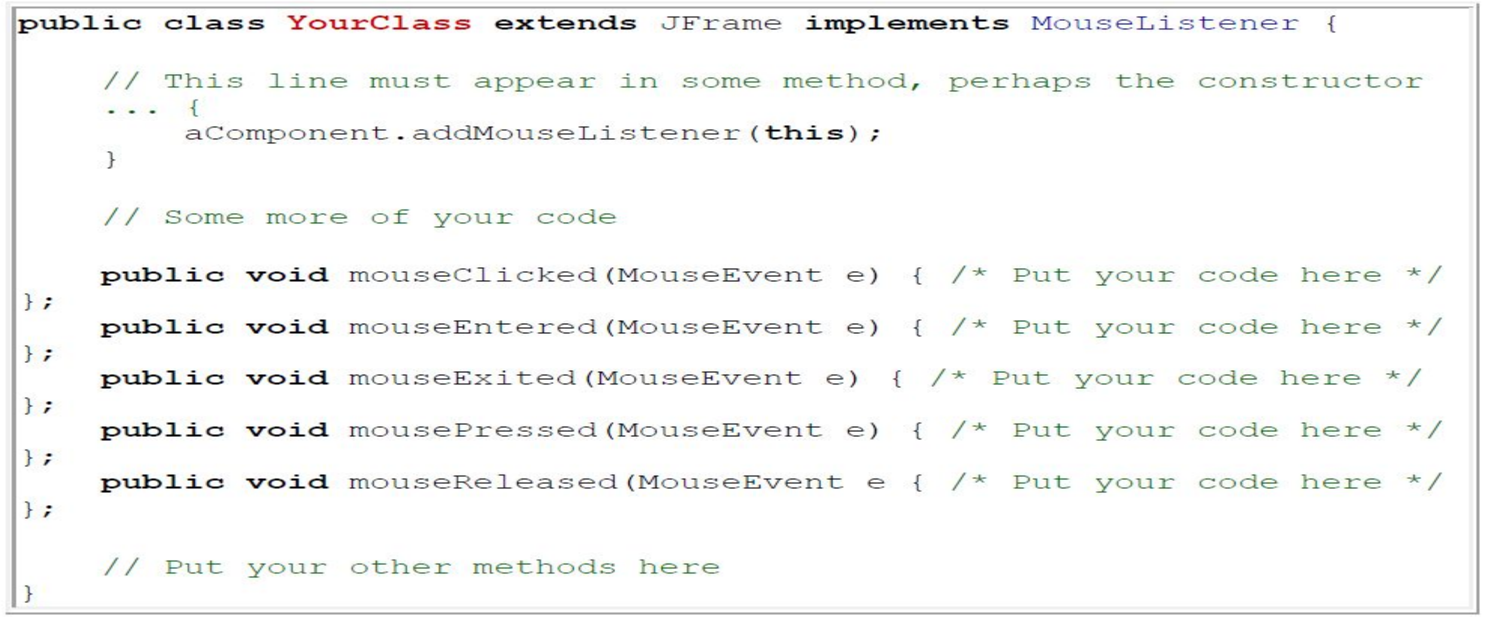
- You can also "unregister" from an event (i.e., disable the listener), by merely removing it using a remove XXXListener() method.

Here are some examples:

```
aButton.removeActionListener(ActionListener anActionListener);
aJPanel.removeMouseListener(MouseListener aMouseListener);
aJFrame.removeWindowListener(WindowListener aWindowListener);
```

Summary of Making your own Event Handlers

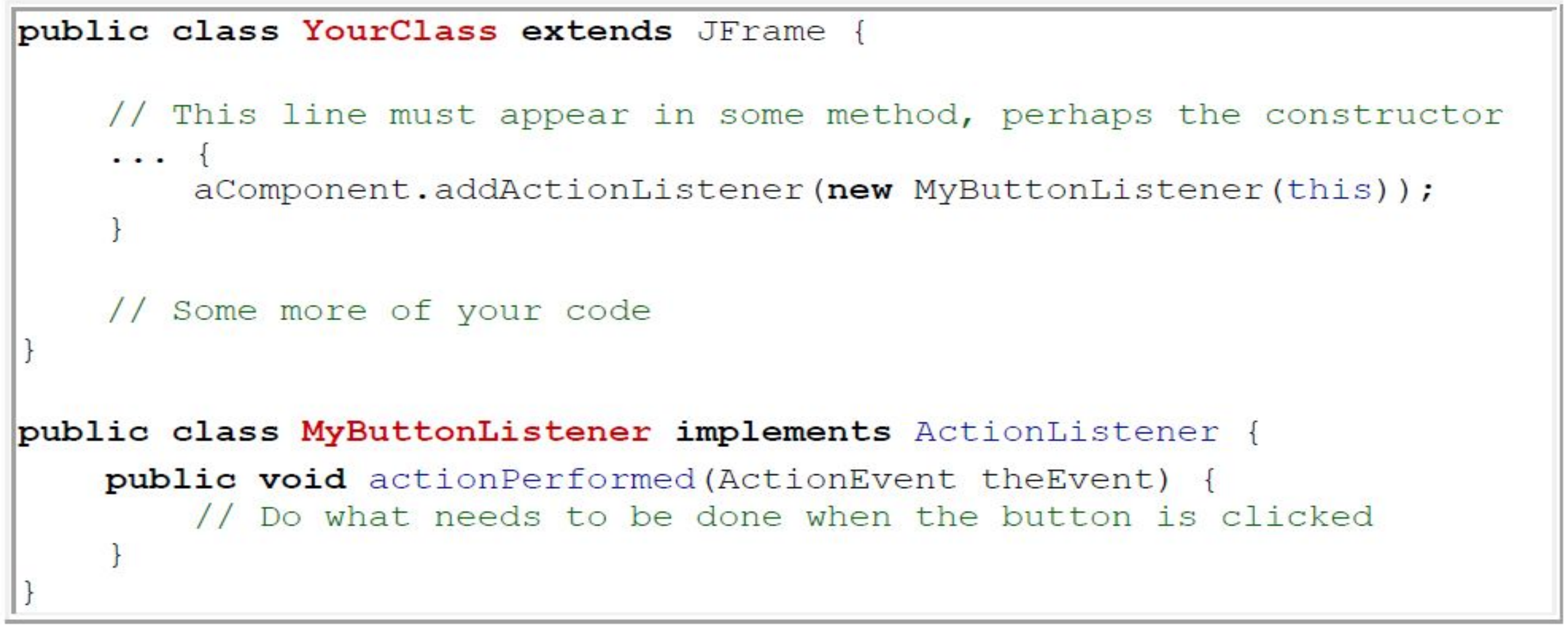
- **Case- I:** Make your class implement the specific interfaces needed:
    - Advantages:
        - Simple
    - Disadvantages:
        - must write methods for ALL events in the interface.
        - can get messy/confusing if your class has many components that trigger the same events or if your class handles many different types of events.



```
public class YourClass extends JFrame implements MouseListener {

    // This line must appear in some method, perhaps the constructor
    ... {
        aComponent.addMouseListener(this);
    }

    // Some more of your code

    public void mouseClicked(MouseEvent e) { /* Put your code here */
};
    public void mouseEntered(MouseEvent e) { /* Put your code here */
};
    public void mouseExited(MouseEvent e) { /* Put your code here */
};
    public void mousePressed(MouseEvent e) { /* Put your code here */
};
    public void mouseReleased(MouseEvent e { /* Put your code here */
};

    // Put your other methods here
}
```

Summary of Making your own Event Handlers

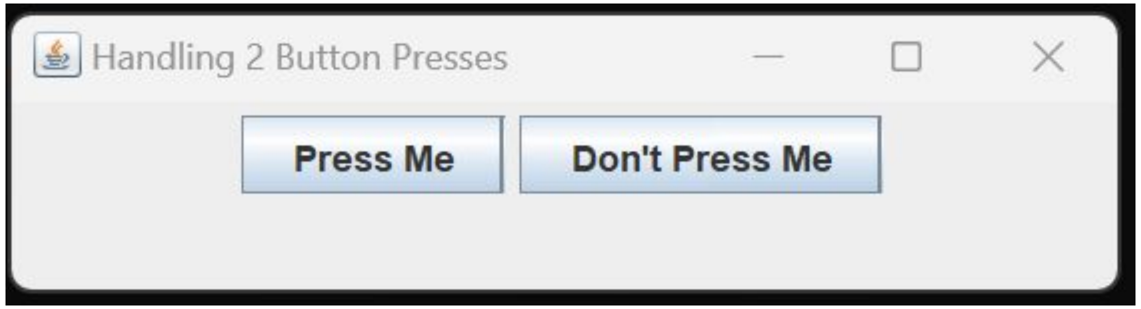
- **Case- II:** Create a separate class that implements the interface:
  - Advantages:
    - nice separation between your code and the event handlers.
    - class can be reused by other classes
  - Disadvantages:
    - can end up with a lot of classes and class files
    - can be confusing as to which classes are just event handler classes



```java
public class YourClass extends JFrame {

    // This line must appear in some method, perhaps the constructor
    ... {
        aComponent.addActionListener(new MyButtonListener(this));
    }

    // Some more of your code
}

public class MyButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent theEvent) {
        // Do what needs to be done when the button is clicked
    }
}
```

Handling ActionEvents with ActionListeners Example

- Here, we present some examples showing how to handle one or more ActionEvents from different kinds of objects.
- **Ex- 1: Handling two button clicks**



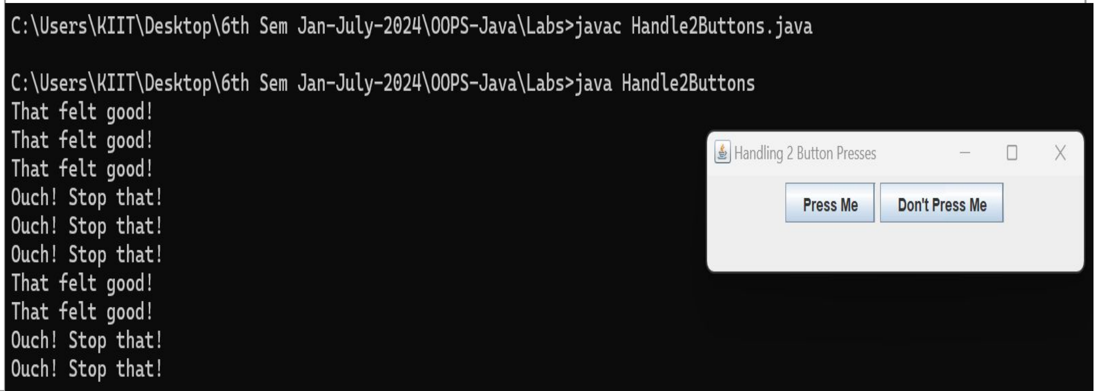
  - We have already seen how to handle a simple button press by writing an **ActionPerformed** method.
  - Here is an application that shows how to handle events for two different buttons.
  - We will make use of the **getActionCommand**() method for the **ActionEvent** class that allows us to determine the label on the button that generated the event.
  - Take notice of the packages that need to be imported.

Handling ActionEvents with ActionListeners Example

- **Ex- 1: Handling two button clicks**

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

public class Handle2Buttons extends JFrame implements
ActionListener
{
        public Handle2Buttons(String title)
        {
                super(title);
                JButton aButton1 = new JButton("Press Me");
                JButton aButton2 = new JButton("Don't Press Me");
                setLayout(new FlowLayout());
                add(aButton1);
                add(aButton2);
                // Indicate that this class will handle 2 button clicks
      // and that both buttons will go to the SAME event handler
                aButton1.addActionListener(this);
                aButton2.addActionListener(this);
                setDefaultCloseOperation(EXIT_ON_CLOSE);
                setSize(250,100);
        }
```

```java
        // This is the event handler for the buttons
        public void actionPerformed(ActionEvent e)
        {
                // Ask the event which button was the source that
generated it
                if (e.getActionCommand().equals("Press Me"))
                        System.out.println("That felt good!");
                else
                        System.out.println("Ouch! Stop that!");
        }
        public static void main(String args[])
        {
                Handle2Buttons frame = new
Handle2Buttons("Handling 2 Button Presses");
                frame.setVisible(true);
        }
}
```



```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac Handle2Buttons.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java Handle2Buttons
That felt good!
That felt good!
That felt good!
Ouch! Stop that!
Ouch! Stop that!
Ouch! Stop that!
That felt good!
That felt good!
Ouch! Stop that!
Ouch! Stop that!
```

Handling ActionEvents with ActionListeners Example

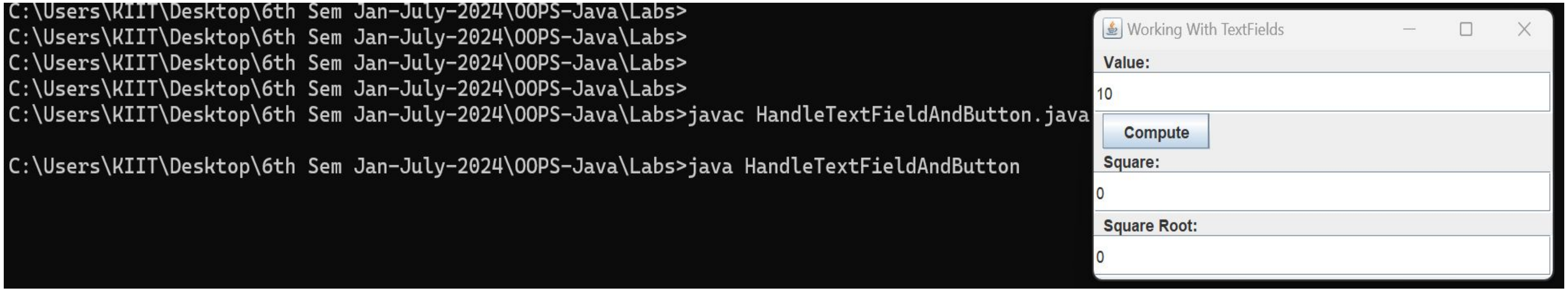
- **Ex- 1: Handling two button clicks**

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

public class Handle2Buttons extends JFrame implements
ActionListener
{
        public Handle2Buttons(String title)
        {
                super(title);
                JButton aButton1 = new JButton("Press Me");
                JButton aButton2 = new JButton("Don't Press Me");
                setLayout(new FlowLayout());
                add(aButton1);
                add(aButton2);
                // Indicate that this class will handle 2 button clicks
      // and that both buttons will go to the SAME event handler
                aButton1.addActionListener(this);
                aButton2.addActionListener(this);
                setDefaultCloseOperation(EXIT_ON_CLOSE);
                setSize(250,100);
        }
        // This is the event handler for the buttons
        public void actionPerformed(ActionEvent e)
        {
                // Ask the event which button was the source that
generated it
                if (e.getActionCommand().equals("Press Me"))
                        System.out.println("That felt good!");
                else
                        System.out.println("Ouch! Stop that!");
        }
        public static void main(String args[])
        {
                Handle2Buttons frame = new
Handle2Buttons("Handling 2 Button Presses");
                frame.setVisible(true);
        }
}
```

- **Notice that the getActionCommand() method is sent to the ActionEvent.**
- **It returns a String containing the text that is on the button that generated the event.**
- **We then compare this string with the labels that we put on the buttons to determine which button was pressed**

Handling ActionEvents with ActionListeners (Example-II)

- **Ex- 2: Working with TextFields**
  - Here is a new application that has a button and some text fields.
  - One text field will hold an integer.
    - When the button is pressed, it will compute and display (in two other text fields) the "square" as well as the "square root" of the value within the first text field.



```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac HandleTextFieldAndButton.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java HandleTextFieldAndButton
```

Working With TextFields

Value: 10

Compute

Square: 0

Square Root: 0

Handling ActionEvents with ActionListeners (Example-II)

- **Ex- 2: Working with TextFields**

    - Note a few things about the code:
        - When creating JTextFields, we can specify the initial content to be displayed (a string) as well as the maximum number of characters allowed to be entered in them (8, 16 and 20 in this example).
        - We need to convert to and from Strings when accessing/modifying text field data
        - We access/modify a text field's contents using getText() and setText()
        - The code below will generate exceptions if a valid integer is not entered within the value text field.

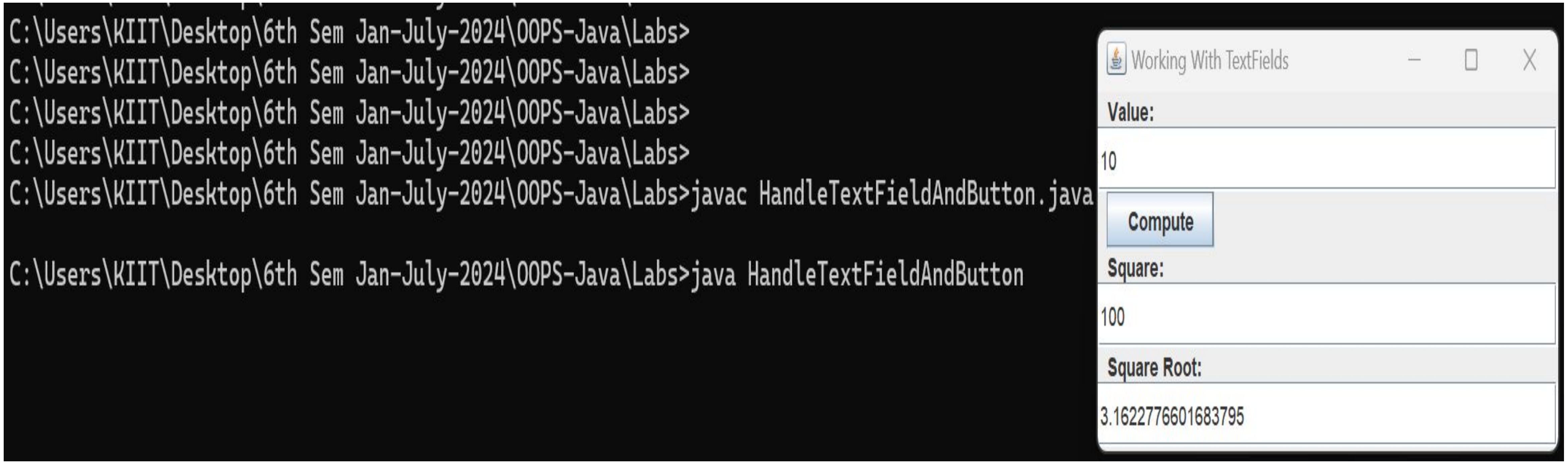
Handling ActionEvents with ActionListeners (Example-II)

- **Ex- 2: Working with TextFields**

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

public class HandleTextFieldAndButton extends JFrame
{
        JTextField valueField, squareField, rootField;
        public HandleTextFieldAndButton(String title)
        {
                super(title);
                setLayout(new
BoxLayout(this.getContentPane(),BoxLayout.Y_AXIS));
                // Add the value text field, along with a label
                add(new JLabel("Value:"));
                valueField = new JTextField("10", 8);
                add(valueField);
                // Add the compute button
                JButton aButton = new JButton("Compute");
                add(aButton);
                // Add the square text field, along with a label
                add(new JLabel("Square:"));
                squareField = new JTextField("0", 16);
                add(squareField);
               // Add the square root text field, along with a label
                add(new JLabel("Square Root:"));
                rootField = new JTextField("0", 20);
                add(rootField);

                // Handle the button click
                aButton.addActionListener(new ActionListener()
                {
                        public void actionPerformed(ActionEvent e)
                        {
                int value = Integer.parseInt(valueField.getText());
                                squareField.setText("" + value * value);
                                rootField.setText("" + Math.sqrt(value));
                        }
                        }
                );
setDefaultCloseOperation(EXIT_ON_CLOSE); setSize(250,180);
        }

                public static void main(String args[])
                {
                        HandleTextFieldAndButton frame = new
HandleTextFieldAndButton("Working With TextFields");
                        frame.setVisible(true);
                }
}
```

Handling ActionEvents with ActionListeners (Example-II)

- **Ex- 2: Working with TextFields**



```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac HandleTextFieldAndButton.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java HandleTextFieldAndButton
```

Working With TextFields

Value:

10

Compute

Square:

100

Square Root:

3.1622776601683795