Graphical Interfaces and Event-Driven Programming

Type	📒 Lecture
Date	@March 22, 2022
Lecture #	1
Lecture URL	https://youtu.be/A_BV43krS2I
Notion URL	https://21f1003586.notion.site/Graphical-Interfaces-and-Event-Driven-Programming-5b1c1b8d4bc949fbb624221b795744ab
Week #	12

GUIs and Events

- Multiple applications simultaneously displayed on screen
- Keystrokes, mouse clicks have to be sent to appropriate window
- In parallel to main activity, record and respond to these events

    - Web browser renders current page
    - Clicking on a link loads a different page

Keeping track of events

- Remember coordinates and extent (size) of each window
- Track coordinates of mouse
- OS reports mouse click at (x, y)
    - Check which windows are positioned at (x, y)
    - Check if one of them is “active”
    - Inform that window about mouse click
- Tedious and error-prone process
- Programming language support for higher level events
    - Run time support for language maps low level events to high level events
    - OS reports low level events: mouse clicked at (x, y), key `'a'` pressed
    - Program sees high level events: Button was clicked , box was ticked ...

Better Programming Language (PL) support for events

- Programmer directly defines components such as windows, buttons, ... that generate high level events
- Each event is associated with a listener that knows what to do
    - e.g. click `Close Window` exits the application
- Programming language has mechanisms for
    - Describing what types of events a component can generate
    - Setting up an association between components and listeners
- Different events invoke different functions
    - Window frame has `Maximize, Iconify, Close` buttons
- Language “sorts” out events and automatically calls the correct function in the listener

Example

- A `Button` with one event, press button
- Pressing the button invokes the function `buttonpush(...)` in a listener

```
interface ButtonListener {
  public abstract void buttonpush(...);
}

class MyClass implements ButtonListener {
  ...
  public void buttonpush(...) {
    // what to do when a button is pushed
    ...
  }
}
```

- We have set up an association between `Button b` and a listener `ButtonListener m`
- Nothing more needs to be done

```
Button b = new Button();
MyClass m = new MyClass();
b.add_listener(m); // Tell b to notify m when pushed
```

- Communicating each button push to the listener is done automatically by the runtime system
- Information about the button push even is passed as an object to the listener
- `buttonpush(...)` has arguments
  - Listeners can decipher source of event, for instance

Timer

- Recall `Timer` Example
- `Myclass m` creates a `Timer t` that runs in parallel
- `Timer t` notifies a `Timerowner` when it is done, via a function `timerdone()`
- Abstractly, timer duration elapsing is an event, and `Timerowner` is notified when the event occurs
  - In the timer, the notification is done explicitly, manually
  - In the button example, the notification is handled internally, automatically
- In our example, `Myclass m` was itself the `Timerowner` to be notified

- In principle, `Timer t` could be passed a reference to any object that implements `Timerowner` interface

Summary

- Event driven programming is a natural way of dealing with the graphical user interface interactions
- User interacts with object through mouse clicks, etc.
- These are automatically translated into events and passed to listeners
- Listeners implement methods that react appropriately to different types of events

Swing ToolKit

Type	📒 Lecture
Date	@March 22, 2022
Lecture #	2
Lecture URL	https://youtu.be/sV4ItidrL8s
Notion URL	https://21f1003586.notion.site/Swing-ToolKit-3dec12f4f8c140c9b3a9ffadd0ecd5e0
Week #	12

Event driven programming in Java

- `Swing` toolkit to define high-levle components
- Built on top of lower level event handling system called `AWT`
- Relationship between components generating events and listeners is flexible
  - One listener can listen to multiple objects
    - Three buttons on window frame all report to common listener
  - One component can inform multiple listeners

        - `Exit browser` report to all windows currently open
- Must explicitly set up association between component and listener
- Events are “lost” if nobody is listening

A button that paints its background red

- `JButton` is Swing class for buttons
- Corresponding listener class is `ActionListener`
- Only one type of event, button push
    - Invokes `actionPerformed(...)` in listener
- Button push is an `ActionEvent`

```
public class MyButtons {
  private JButton b;
  public MyButtons(ActionListener a) {
    // Set the label on the button
    b = new JButton("MyButton");
    // Associate a listener
    b.addActionListener(a);
  }
  public class MyListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      ...
      // When a button is pressed, the code here executes
    }
  }
  public class XYZ {
    // ActionListener l
    MyListener l = new MyListener();
    // Button m, reports to l
    MyButtons m = new MyButtons(l);
  }
}
```

Embedding the button inside a panel

- To embed the button in a panel — `JPanel`
    - First import required Java packages
    - The panel will also serve as the event listener

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

```
public class ButtonPanel extends JPanel implements ActionListener {
  ...
}
```

- Create the button, make the panel a listener and add the button to the panel

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

public class ButtonPanel extends JPanel implements ActionListener {
  private JButton redButton;
  public ButtonPanel() {
    redButton = new JButton("Red");
    redButton.addActionListener(this);
    add(redButton);
  }
  ...
}
```

- Listener sets the panel background to red when the button is clicked

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

public class ButtonPanel extends JPanel implements ActionListener {
  private JButton redButton;
  public ButtonPanel() {
    redButton = new JButton("Red");
    redButton.addActionListener(this);
    add(redButton);
  }
  public void actionPerformed(ActionEvent event) {
    Color color = Color.red;
    setBackground(color);
    repaint();
  }
}
```

Embedding the panel in a frame

- Embed the panel in a frame — `JFrame`

```
public class ButtonFrame extends JFrame implements WindowListener {
  public ButtonFrame() { ... }
  // Implement WindowListener
```

```
  ...
}
```

- Corresponding listener class is `WindowListener`
- `JFrame` generates seven different types of events
  - Each of the seven events automatically calls a different function in `WindowListener`

```
public class ButtonFrame extends JFrame implements WindowListener {
  public ButtonFrame() { ... }
  /*
  Seven methods required for implementing WindowListener
  // Six out of the seven are stubs
  */
  ...
}
```

- Need to implement `windowClosing` event to terminate the window
- Other six types of events can be ignored

```
public class ButtonFrame extends JFrame implements WindowListener {
  public ButtonFrame() { ... }
  // Six out of the seven methods required for
  // implementing WindowListener are stubs

  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
  public void windowActivated(WindowEvent e) {}
  public void windowClosed(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowOpened(WindowEvent e) {}
}
```

- `JFrame` is complex, many layers
- Items to be displayed have to be added to `ContentPane`

```
public class ButtonFrame extends JFrame implements WindowListener {
  private Container contentPane;
  public ButtonFrame() {
    setTitle("ButtonTest");
    setSize(300, 200);
```

```
    // ButtonFrame listens to itself
    addWindowListener(this);

    // ButtonPanel is added to the contentPane
    contentPane = this.getContentPane();
    contentPane.add(new ButtonPanel());
  }
}
```

`main` function

- Create a `JFrame` and make it visible

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

public class ButtonTest {
  public static void main(String[] args) {
    EventQueue.invokeLater(() -> {
      JFrame frame = new ButtonFrame();
      frame.setVisible(true);
    });
  }
}
```

- `EventQueue.invokeLater()` puts the Swing object in a separate event despatch thread
- Ensures that GUI processing does not interfere with other computation
- GUI does not get blocked, avoid subtle synchronization bugs

Summary

- The swing toolkit has different types of objects
- Each object generates its own type of event
- Create an appropriate event handler and link it with the object
- The unit that Swing displays is a frame
- Individual objects have to be embedded in panels which are then added to the frame

More Swing Examples

Type	📒 Lecture
Date	@March 22, 2022
Lecture #	3
Lecture URL	https://youtu.be/c6Z8BNSv9zY
Notion URL	https://21f1003586.notion.site/More-Swing-Examples-a061c1aa699748b1b9c3adbc2c8e6526
Week #	12

Connecting multiple events to a listener

- One listener can listen to multiple objects
- A panel with 3 buttons, to paint the panel red, yellow or blue

```
public class ButtonPanel extends JPanel implements ActionListener {
  // Panel has 3 buttons
  private JButton yellowButton, blueButton, redButton;
  public ButtonPanel() {
    yellowButton = new JButton("Yellow");
    blueButton = new JButton("Blue");
```

```
    redButton = new JButton("Red");
    ...
  }

  public void actionPerformed(ActionEvent event) {
    ...
  }
}
```

- Make the panel listen to all 3 buttons

```
public class ButtonPanel extends JPanel implements ActionListener {
  // Panel has 3 buttons
  private JButton yellowButton, blueButton, redButton;
  public ButtonPanel() {
    yellowButton = new JButton("Yellow");
    blueButton = new JButton("Blue");
    redButton = new JButton("Red");

    // ButtonPanel listens to all 3 buttons
    yellowButton.addActionListener(this);
    blueButton.addActionListener(this);
    redButton.addActionListener(this);

    add(yellowButton);
    add(blueButton);
    add(redButton);
  }
  ...
}
```

- Determine what colour to use by identifying source of the event
  - Keep the existing colour if the source is not one of these three buttons

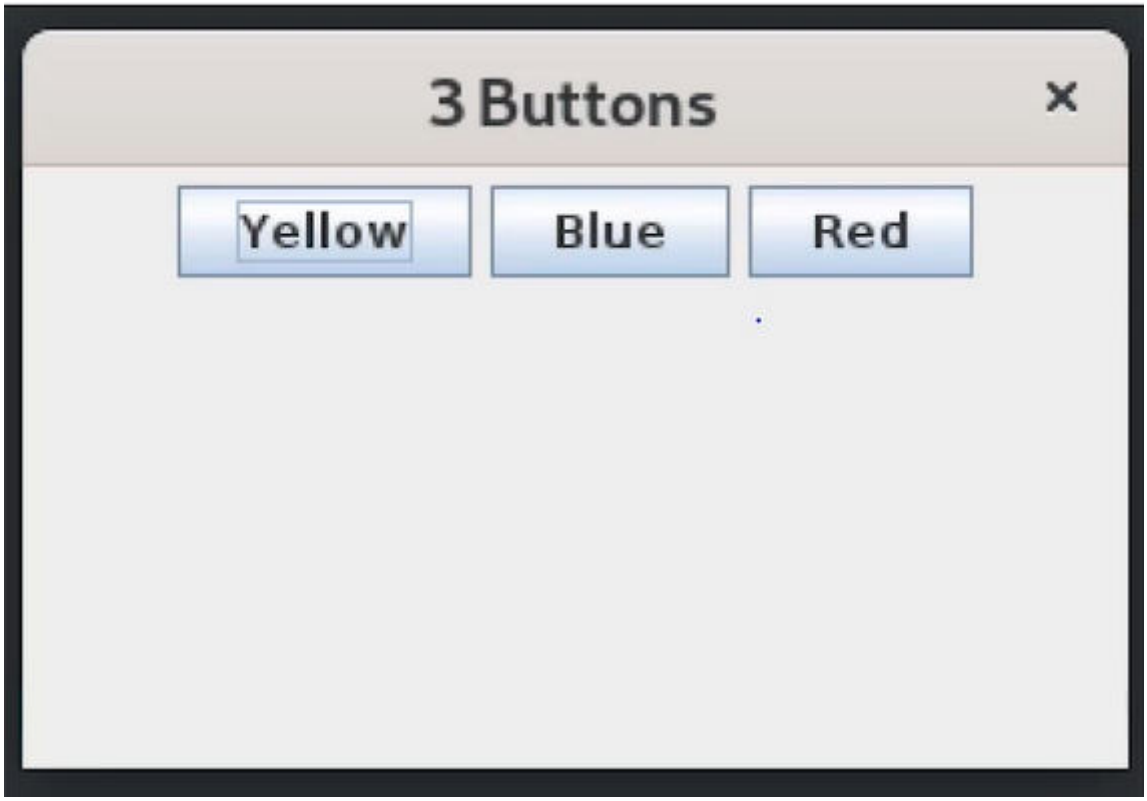
```
public class ButtonPanel extends JPanel implements ActionListener {
  ...
  public void actionPeformed(ActionEvent event) {
    // Find the source of the event
    Object source = event.getSource();
    // Get current background colour
    Color color = getBackground();

    if(source == yellowButton) {
      color = Color.yellow;
    } else if(source == blueButton) {
      color = Color.blue;
    } else if(source == redButton) {
      color = Color.red;
    }

    setBackground(color);
```

```
    repaint();
  }
}
```

- Output



Multicasting: multiple listeners for an event

- Two panels, each with 3 buttons, `Red, Blue, Yellow`

```
import ...;
public class ButtonPanel extends JPanel implements ActionListener {
  // Panel has 3 buttons
  private JButton yellowButton, blueButton, redButton;
  public ButtonPanel() {
    yellowButton = new JButton("Yellow");
    blueButton = new JButton("Blue");
    redButton = new JButton("Red");
    ...
    add(yellowButton);
    add(blueButton);
    add(redButton);
  }
  ...
}
```

- Clicking a button in either panel changes the background colour in both panels
- Both panels must listen to all six buttons
  - However, each panel has references only for its local buttons
- Associate an `ActionCommand` with a button
  - Assign the same action command to both `Red` buttons ...

```
import ...;
public class ButtonPanel extends JPanel implements ActionListener {
  // Panel has 3 buttons
  private JButton yellowButton, blueButton, redButton;
  public ButtonPanel() {
    yellowButton = new JButton("Yellow");
    blueButton = new JButton("Blue");
    redButton = new JButton("Red");

    yellowButton.setActionCommand("YELLOW");
    blueButton.setActionCommand("BLUE");
    redButton.setActionCommand("RED");

    add(yellowButton);
    add(blueButton);
    add(redButton);
  }
  ...
}
```

- Choose colour according to `ActionCommand`

```
public class ButtonPanel extends JPanel implements ActionListener {
  ...
  public void actionPeformed(ActionEvent event) {
    Color color = getBackground();
    String cmd = event.getActionCommand();

    if(cmd.equals("YELLOW")) {
      color = Color.yellow;
    } else if(cmd.equals("BLUE")) {
      color = Color.blue;
    } else if(cmd.equals("RED")) {
      color = Color.red;
    }

    setBackground(color);
    repaint();
  }
  ...
}
```

- Need to add both panels as listeners for each button
  - Add a public function to add a new listener to all buttons in a panel

```
public class ButtonPanel extends JPanel implements ActionListener {
  ...
  public void addListener(ActionListener o) {
    // Add a common listener for all
    // buttons in the panel
    yellowButton.addActionListener(o);
    blueButton.addActionListener(o);
    redButton.addActionListener(o);
  }
}
```

- Add both panels to the same frame

```
public class ButtonFrame extends JFrame implements WindowListener {
  private Container contentPane;
  private ButtonPanel b1, b2;

  public ButtonFrame() {
    ...
    b1 = new ButtonPanel();
    b2 = new ButtonPanel();

    // Each panel listens to both sets of buttons
    b1.addListener(b1);
    b1.addListener(b2);
    b2.addListener(b1);
    b2.addListener(b2);

    contentPane = this.getContentPane();
    // Set layout to separate out panels in frame
    contentPane.setLayout(new BorderLayout());
    contentPane.add(b1, "North");
    contentPane.add(b2, "South");
  }
}
```

Other elements - checkboxes

- `JCheckbox`: a box that can be ticked
- A panel with two checkboxes, `Red` and `Blue`
  - Only `Red` ticked, background red
  - Only `Blue` ticked, background blue
  - Both ticked, background green

- Only one action — click the box
  - Listener is again `ActionListener`
- Checkbox state: selected or not

```
import ...
public class CheckBoxPanel extends JPanel implements ActionListener {
  private JCheckBox redBox;
  private JCheckBox blueBox;

  public CheckBoxPanel() {
    redBox = new JCheckBox("Red");
    redBox = new JCheckBox("Blue");

    redBox.addActionListener(this);
    blueBox.addActionListener(this);

    redBox.setSelected(false);
    blueBox.setSelected(false);

    add(redBox);
    add(blueBox);
    ...
  }
}
```

- `isSelected()` returns the current state

```
public class CheckBoxPanel extends JPanel implements ActionListener {
  ...
  public void actionPerformed(ActionEvent event) {
    Color color = getBackground();
    if(blueBox.isSelected()) {
      color = Color.blue;
    } else if(redBox.isSelected()) {
      color = Color.red;
    } else if(blueBox.isSelected() && redBox.isSelected()) {
      color = Color.green;
    }

    setBackground(color);
    repaint();
  }
}
```

Summary

- Swing components such as buttons, checkboxes generate high level events
- Each event is automatically sent to a listener

    - Listener capability is described using an interface
    - Event is sent as an object — listener can query the event to obtain details such as event source, action label, ... and react accordingly
- Association between event generators and listeners is flexible
    - One listener can listen to multiple objects
    - One component can inform multiple listeners
- Must explicitly set up association between component and listener
    - Events are “lost” if nobody is listening
- Swing objects are the most aesthetically pleasing, but useful to understand how GUI programming works across other languages