Java Swing events

Events are an important part in any GUI program. All GUI applications are event-driven. An application reacts to different event types which are generated during its life. Events are generated mainly by the user of an application. But they can be generated by other means as well, e.g. Internet connection, window manager, timer. In the event model, there are three participants:

- event source
- event object
- event listener

The *Event source* is the object whose state changes. It generates Events. The *Event object* (Event) encapsulates the state changes in the event source. The *Event listener* is the object that wants to be notified. Event source object delegates the task of handling an event to the event listener.

Event handling in Java Swing toolkit is very powerful and flexible. Java uses Event Delegation Model. We specify the objects that are to be notified when a specific event occurs.

An event object

When something happens in the application, an event object is created. For example, when we click on the button or select an item from a list. There are several types of events, including `ActionEvent`, `TextEvent`, `FocusEvent`, and `ComponentEvent`. Each of them is created under specific conditions.

An event object holds information about an event that has occurred. In the next example, we will analyse an `ActionEvent` in more detail.

```
package com.zetcode;

import java.awt.Container;
import java.awt.Dimension;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;
import javax.swing.AbstractAction;
import javax.swing.BorderFactory;
import javax.swing.DefaultListModel;
import javax.swing.GroupLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import static javax.swing.JFrame.EXIT_ON_CLOSE;
import javax.swing.JList;
```

```
public class EventObjectEx extends JFrame {

    private JList list;
    private DefaultListModel model;


    public EventObjectEx() {

        initUI();
    }

    private void initUI() {

        Container pane = getContentPane();
        GroupLayout gl = new GroupLayout(pane);
        pane.setLayout(gl);

        model = new DefaultListModel();
        list = new JList(model);
        list.setMinimumSize(new Dimension(250, 150));
        list.setBorder(BorderFactory.createEtchedBorder());

        JButton okButton = new JButton("OK");
        okButton.addActionListener(new ClickAction());

        gl.setAutoCreateContainerGaps(true);

        gl.setHorizontalGroup(gl.createSequentialGroup()
                .addComponent(okButton)
                .addGap(20)
                .addComponent(list)
        );

        gl.setVerticalGroup(gl.createParallelGroup()
                .addComponent(okButton)
                .addComponent(list)
        );

        pack();

        setTitle("Event object");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
    }

    private class ClickAction extends AbstractAction {

        @Override
        public void actionPerformed(ActionEvent e) {

            Locale locale = Locale.getDefault();
            Date date = new Date(e.getWhen());
            String tm = DateFormat.getTimeInstance(DateFormat.SHORT,
                    locale).format(date);

            if (!model.isEmpty()) {
                model.clear();
            }

            if (e.getID() == ActionEvent.ACTION_PERFORMED) {
                model.addElement("Event Id: ACTION_PERFORMED");
            }

            model.addElement("Time: " + tm);

            String source = e.getSource().getClass().getName();
            model.addElement("Source: " + source);

            int mod = e.getModifiers();

            StringBuffer buffer = new StringBuffer("Modifiers: ");

            if ((mod & ActionEvent.ALT_MASK) > 0) {
                buffer.append("Alt ");
            }

            if ((mod & ActionEvent.SHIFT_MASK) > 0) {
                buffer.append("Shift ");
            }

            if ((mod & ActionEvent.META_MASK) > 0) {
                buffer.append("Meta ");
            }

            if ((mod & ActionEvent.CTRL_MASK) > 0) {
                buffer.append("Ctrl ");
            }
```

```
            model.addElement(buffer);
        }
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {
            @Override
            public void run() {
                EventObjectEx ex = new EventObjectEx();
                ex.setVisible(true);
            }
        });
    }
}
```

The code example shows a button and a list. If we click on the button, the information about the event is displayed in the list. In our case, we are talking about an `ActionEvent` class. The data will be the time when the event occurred, the id of the event, the event source, and the modifier keys.

```
okButton.addActionListener(new ClickAction());
```

The `ClickAction` listens to the events of the OK button.

```
private class ClickAction extends AbstractAction {

    @Override
    public void actionPerformed(ActionEvent e) {
        ...
    }
}
```

The `actionPerformed()` method is invoked when an action occurs. Its parameter is an `ActionEvent` object.

```
Locale locale = Locale.getDefault();
Date date = new Date(e.getWhen());
String s = DateFormat.getTimeInstance(DateFormat.SHORT,
        locale).format(date);
```

We get the time when the event occurred. The `getWhen()` method returns time value in milliseconds. So we must format it appropriately.

```
String source = e.getSource().getClass().getName();
```

```
model.addElement("Source: " + source);
```

Here we add the name of the source of the event to the list. In our case the source is a `JButton`.

```
int mod = event.getModifiers();
```

We get the modifier keys. It is a bitwise-or of the modifier constants.

```
if ((mod & ActionEvent.SHIFT_MASK) > 0)
    buffer.append("Shift ");
```

Here we determine whether we have pressed a Shift key.

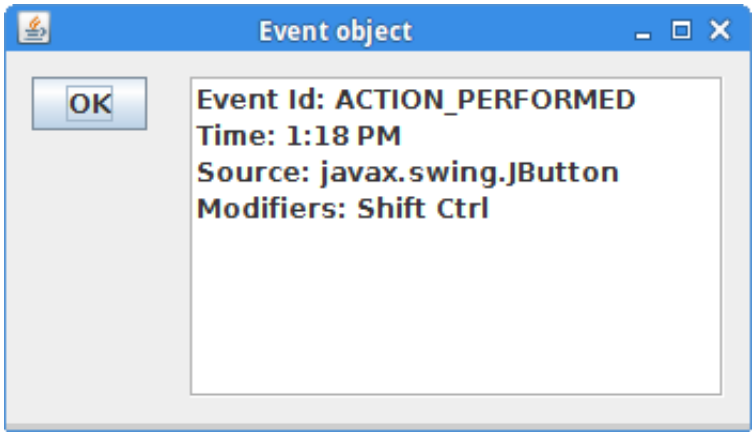

Figure: Event Object

Implementation

There are several ways, how we can implement event handling in Java Swing toolkit.

- Anonymous inner class
- Inner class
- Derived class

Anonymous inner class

We will illustrate these concepts on a simple event example.

```
package com.zetcode;

import java.awt.Container;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.GroupLayout;
import javax.swing.JButton;
```

```
import javax.swing.JFrame;


public class AnonymousInnerClassEx extends JFrame {


    public AnonymousInnerClassEx() {

        initUI();
    }

    private void initUI() {

        Container pane = getContentPane();
        GroupLayout gl = new GroupLayout(pane);
        pane.setLayout(gl);

        JButton closeButton = new JButton("Close");
        closeButton.setBounds(40, 50, 80, 25);

        closeButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent event) {
                System.exit(0);
            }
        });

        gl.setAutoCreateContainerGaps(true);

        gl.setHorizontalGroup(gl.createSequentialGroup()
                .addComponent(closeButton)
                .addGap(220)
        );

        gl.setVerticalGroup(gl.createSequentialGroup()
                .addComponent(closeButton)
                .addGap(180)
        );

        pack();

        setTitle("Anonymous inner class");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
```

```
    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {

            @Override
            public void run() {
                AnonymousInnerClassEx ex =
                        new AnonymousInnerClassEx();
                ex.setVisible(true);
            }
        });
    }
}
```

In this example, we have a button that closes the window upon clicking.

```
JButton closeButton = new JButton("Close");
```

The Close button is the *event source*. It will generate events.

```
closeButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent event) {
        System.exit(0);
    }
});
```

Here we *register* an action listener with the button. The events are sent to the *event target*. The event target in our case is `ActionListener` class; in this code, we use an *anonymous inner class*.

Inner class

Here we implement the example using an inner `ActionListener` class.

```
package com.zetcode;

import java.awt.Container;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.GroupLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
```

```
public class InnerClassExample extends JFrame {

    public InnerClassExample() {

        initUI();
    }

    private void initUI() {

        Container pane = getContentPane();
        GroupLayout gl = new GroupLayout(pane);
        pane.setLayout(gl);

        JButton closeButton = new JButton("Close");

        ButtonCloseListener listener = new ButtonCloseListener();
        closeButton.addActionListener(listener);

        gl.setAutoCreateContainerGaps(true);

        gl.setHorizontalGroup(gl.createSequentialGroup()
                .addComponent(closeButton)
                .addGap(220)
        );

        gl.setVerticalGroup(gl.createSequentialGroup()
                .addComponent(closeButton)
                .addGap(180)
        );

        pack();

        setTitle("Inner class example");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private class ButtonCloseListener implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    }
```

```
    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {

            @Override
            public void run() {
                InnerClassExample ex = new InnerClassExample();
                ex.setVisible(true);
            }
        });
    }
}
```

We have a Close button on the panel. Its listener is defined inside a named inner class.

```
ButtonCloseListener listener = new ButtonCloseListener();
closeButton.addActionListener(listener);
```

Here we have a non-anonymous inner class.

```
private class ButtonCloseListener implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}
```

The button listener is defined here.

A derived class implementing the listener

The following example will derive a class from a component and implement an action listener inside the class.

```
package com.zetcode;

import java.awt.Container;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.GroupLayout;
import javax.swing.JButton;
```

```
import javax.swing.JFrame;

public class DerivedClassExample extends JFrame {

    public DerivedClassExample() {

        initUI();
    }

    private void initUI() {

        Container pane = getContentPane();
        GroupLayout gl = new GroupLayout(pane);
        pane.setLayout(gl);

        MyButton closeButton = new MyButton("Close");

        gl.setAutoCreateContainerGaps(true);

        gl.setHorizontalGroup(gl.createSequentialGroup()
                .addComponent(closeButton)
                .addGap(220)
        );

        gl.setVerticalGroup(gl.createSequentialGroup()
                .addComponent(closeButton)
                .addGap(180)
        );

        pack();

        setTitle("Derived class");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private class MyButton extends JButton
        implements ActionListener {

        public MyButton(String text) {
            super.setText(text);
            addActionListener(this);
        }

        @Override
```

```
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {

            @Override
            public void run() {
                DerivedClassExample ex = new DerivedClassExample();
                ex.setVisible(true);
            }
        });
    }
}
```

In this example, we create a derived MyButton class, which implements the action listener.

```
MyButton closeButton = new MyButton("Close");
```

Here we create the custom MyButton class.

```
private class MyButton extends JButton
    implements ActionListener {

    public MyButton(String text) {
        super.setText(text);
        addActionListener(this);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}
```

The `MyButton` class is extended from the `JButton` class. It implements the `ActionListener` interface. This way, the event handling is managed within the `MyButton` class.

Multiple sources

A listener can be plugged into several sources. This will be explained in the next example.

```
package com.zetcode;

import java.awt.BorderLayout;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.BorderFactory;
import javax.swing.GroupLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import static javax.swing.JFrame.EXIT_ON_CLOSE;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class MultipleSources extends JFrame {

    private JLabel statusbar;

    public MultipleSources() {

        initUI();
    }

    private void initUI() {

        JPanel panel = new JPanel();
        GroupLayout gl = new GroupLayout(panel);
        panel.setLayout(gl);

        statusbar = new JLabel("ZetCode");

        statusbar.setBorder(BorderFactory.createEtchedBorder());

        ButtonListener butlist = new ButtonListener();

        JButton closeButton = new JButton("Close");
        closeButton.addActionListener(butlist);

        JButton openButton = new JButton("Open");
        openButton.addActionListener(butlist);

        JButton findButton = new JButton("Find");
        findButton.addActionListener(butlist);

        JButton saveButton = new JButton("Save");
```

```
        saveButton.addActionListener(butlist);

        gl.setAutoCreateContainerGaps(true);
        gl.setAutoCreateGaps(true);

        gl.setHorizontalGroup(gl.createParallelGroup()
                .addComponent(closeButton)
                .addComponent(openButton)
                .addComponent(findButton)
                .addComponent(saveButton)
                .addGap(250)
        );

        gl.setVerticalGroup(gl.createSequentialGroup()
                .addComponent(closeButton)
                .addComponent(openButton)
                .addComponent(findButton)
                .addComponent(saveButton)
                .addGap(20)
        );

        gl.linkSize(closeButton, openButton,
                findButton, saveButton);

        add(panel, BorderLayout.CENTER);
        add(statusbar, BorderLayout.SOUTH);

        pack();

        setTitle("Multiple Sources");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private class ButtonListener implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {

            JButton o = (JButton) e.getSource();
            String label = o.getText();
            statusbar.setText(" " + label + " button clicked");
        }
    }
```

```
    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {

            @Override
            public void run() {
                MultipleSources ms = new MultipleSources();
                ms.setVisible(true);
            }
        });
    }
}
```

We create four buttons and a statusbar. The statusbar will display a message upon clicking on the button.

```
JButton closeButton = new JButton("Close");
closeButton.addActionListener(butlist);

JButton openButton = new JButton("Open");
openButton.addActionListener(butlist);

...
```

Each button registers the same `ButtonListener` object.

```
private class ButtonListener implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {

        JButton o = (JButton) e.getSource();
        String label = o.getText();
        statusbar.setText(" " + label + " button clicked");
    }
}
```

We determine which button was pressed and create a message for the statusbar. The message is set with the `setText()` method.

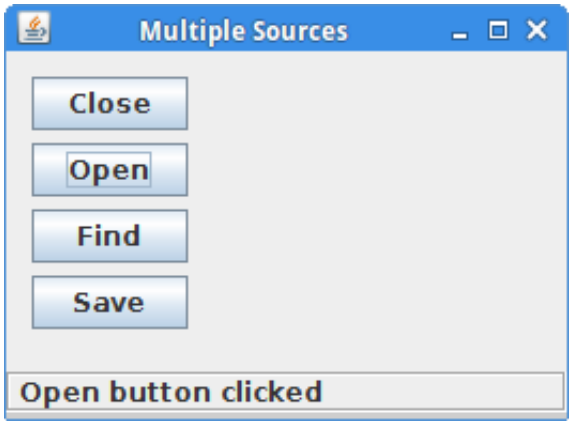

Figure: Multiple Sources

Multiple listeners

It is possible to register several listeners for one event.

```
package com.zetcode;

import java.awt.BorderLayout;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Calendar;
import javax.swing.BorderFactory;
import javax.swing.GroupLayout;
import static javax.swing.GroupLayout.Alignment.CENTER;
import static javax.swing.GroupLayout.DEFAULT_SIZE;
import static javax.swing.GroupLayout.PREFERRED_SIZE;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSpinner;
import javax.swing.SpinnerModel;
import javax.swing.SpinnerNumberModel;


public class MultipleListeners extends JFrame {

    private JLabel statusbar;
    private JSpinner spinner;
    private int count = 0;

    public MultipleListeners() {

        initUI();
```

```
    }

    private void initUI() {

        JPanel panel = new JPanel();
        GroupLayout gl = new GroupLayout(panel);
        panel.setLayout(gl);
        add(panel, BorderLayout.CENTER);

        statusbar = new JLabel("0");
        statusbar.setBorder(BorderFactory.createEtchedBorder());
        add(statusbar, BorderLayout.SOUTH);

        JButton addButton = new JButton("+");
        addButton.addActionListener(new ButtonListener1());
        addButton.addActionListener(new ButtonListener2());

        Calendar calendar = Calendar.getInstance();
        int currentYear = calendar.get(Calendar.YEAR);

        SpinnerModel yearModel = new SpinnerNumberModel(currentYear,
                currentYear - 100,
                currentYear + 100,
                1);

        spinner = new JSpinner(yearModel);
        spinner.setEditor(new JSpinner.NumberEditor(spinner, "#"));


        gl.setAutoCreateContainerGaps(true);

        gl.setHorizontalGroup(gl.createSequentialGroup()
                .addComponent(addButton)
                .addGap(20)
                .addComponent(spinner, DEFAULT_SIZE,
                        DEFAULT_SIZE, PREFERRED_SIZE)
        );

        gl.setVerticalGroup(gl.createSequentialGroup()
                .addGroup(gl.createParallelGroup(CENTER)
                        .addComponent(addButton)
                        .addComponent(spinner, DEFAULT_SIZE,
                                DEFAULT_SIZE, PREFERRED_SIZE))
        );
```

```
        pack();

        setTitle("Multiple Listeners");
        setSize(300, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private class ButtonListener1 implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {
            Integer val = (Integer) spinner.getValue();
            spinner.setValue(++val);
        }
    }

    private class ButtonListener2 implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {
            statusbar.setText(Integer.toString(++count));
        }
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {

            @Override
            public void run() {
                MultipleListeners ml = new MultipleListeners();
                ml.setVisible(true);
            }
        });
    }
}
```

In this example, we have a button, a spinner, and a statusbar. We use two button listeners for one event. One click of a button will add one year to the spinner component and update the statusbar. The statusbar will show how many times we have clicked on the button.

```
addButton.addActionListener(new ButtonListener1());
addButton.addActionListener(new ButtonListener2());
```

We register two button listeners.

```
SpinnerModel yearModel = new SpinnerNumberModel(currentYear,
        currentYear - 100,
        currentYear + 100,
        1);

spinner = new JSpinner(yearModel);
```

Here we create the spinner component. We use a year model for the spinner. The `SpinnerNumberModel` arguments are the initial value, min, and max values and the step.

```
spinner.setEditor(new JSpinner.NumberEditor(spinner, "#"));
```

We remove the thousands separator.

```
private class ButtonListener1 implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
        Integer val = (Integer) spinner.getValue();
        spinner.setValue(++val);
    }
}
```

The first button listener increases the value of the spinner component.

```
private class ButtonListener2 implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
        statusbar.setText(Integer.toString(++count));
    }
}
```

The second button listener increases the value of the status bar.

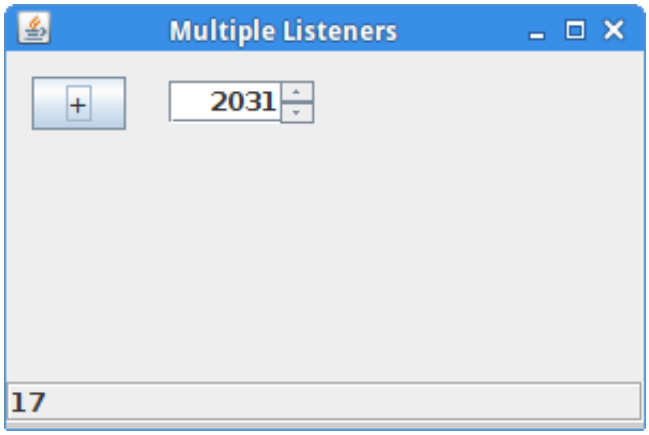

Figure: Multiple listeners

Removing listeners

It is possible to remove the registered listeners with the `removeActionListener()` method. The following example demonstrates this.

```
package com.zetcode;

import java.awt.Container;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.GroupLayout;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class RemoveListenerEx extends JFrame {

    private JLabel lbl;
    private JButton addButton;
    private JCheckBox activeBox;
    private ButtonListener buttonlistener;
    private int count = 0;

    public RemoveListenerEx() {

        initUI();
    }

    private void initUI() {
```

```
        Container pane = getContentPane();
        GroupLayout gl = new GroupLayout(pane);
        pane.setLayout(gl);

        addButton = new JButton("+");
        buttonlistener = new ButtonListener();

        activeBox = new JCheckBox("Active listener");
        activeBox.addItemListener(new ItemListener() {

            @Override
            public void itemStateChanged(ItemEvent event) {

                if (activeBox.isSelected()) {
                    addButton.addActionListener(buttonlistener);
                } else {
                    addButton.removeActionListener(buttonlistener);
                }
            }
        });

        lbl = new JLabel("0");

        gl.setAutoCreateContainerGaps(true);

        gl.setHorizontalGroup(gl.createSequentialGroup()
                .addGroup(gl.createParallelGroup()
                        .addComponent(addButton)
                        .addComponent(lbl))
                .addGap(30)
                .addComponent(activeBox)
        );

        gl.setVerticalGroup(gl.createSequentialGroup()
                .addGroup(gl.createParallelGroup()
                        .addComponent(addButton)
                        .addComponent(activeBox))
                .addGap(30)
                .addComponent(lbl)
        );

        pack();

        setTitle("Remove listener");
        setLocationRelativeTo(null);
```

```
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private class ButtonListener implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {

            lbl.setText(Integer.toString(++count));
        }
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {

            @Override
            public void run() {
                RemoveListenerEx ex = new RemoveListenerEx();
                ex.setVisible(true);
            }
        });
    }
}
```

We have three components on the panel: a button, a check box, and a label. By toggling the check box, we add or remove the listener for a button.

```
buttonlistener = new ButtonListener();
```

We have to create a non-anonymous listener if we want to later remove it.

```
if (activeBox.isSelected()) {
    addButton.addActionListener(buttonlistener);
} else {
    addButton.removeActionListener(buttonlistener);
}
```

We determine whether the check box is selected. Then we add or remove the listener.

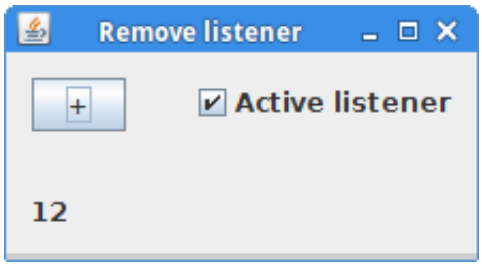

Figure: Remove listener

Moving a window

The following example will look for a position of a window on the screen.

```
package com.zetcode;

import java.awt.Container;
import java.awt.EventQueue;
import java.awt.Font;
import java.awt.event.ComponentEvent;
import java.awt.event.ComponentListener;
import javax.swing.GroupLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;


public class MovingWindowEx extends JFrame
        implements ComponentListener {

    private JLabel labelx;
    private JLabel labely;

    public MovingWindowEx() {

        initUI();
    }

    private void initUI() {

        Container pane = getContentPane();
        GroupLayout gl = new GroupLayout(pane);
        pane.setLayout(gl);

        addComponentListener(this);

        labelx = new JLabel("x: ");
        labelx.setFont(new Font("Serif", Font.BOLD, 14));
        labelx.setBounds(20, 20, 60, 25);

        labely = new JLabel("y: ");
        labely.setFont(new Font("Serif", Font.BOLD, 14));
        labely.setBounds(20, 45, 60, 25);
```

```
        gl.setAutoCreateContainerGaps(true);
        gl.setAutoCreateGaps(true);

        gl.setHorizontalGroup(gl.createParallelGroup()
                .addComponent(labelx)
                .addComponent(labely)
                .addGap(250)
        );

        gl.setVerticalGroup(gl.createSequentialGroup()
                .addComponent(labelx)
                .addComponent(labely)
                .addGap(130)
        );

        pack();

        setTitle("Moving window");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    @Override
    public void componentResized(ComponentEvent e) {
    }

    @Override
    public void componentMoved(ComponentEvent e) {

        int x = e.getComponent().getX();
        int y = e.getComponent().getY();

        labelx.setText("x: " + x);
        labely.setText("y: " + y);
    }

    @Override
    public void componentShown(ComponentEvent e) {
    }

    @Override
    public void componentHidden(ComponentEvent e) {
    }
```

```
    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {

            @Override
            public void run() {
                MovingWindowEx ex = new MovingWindowEx();
                ex.setVisible(true);
            }
        });
    }
}
```

The example shows the current window coordinates on the panel. To get the window position, we use the `ComponentListener`.

```
public class MovingWindowExample extends JFrame
        implements ComponentListener {
```

The main class implements the `ComponentListener` interface. It has to provide implementation of all its methods.

```
@Override
public void componentResized(ComponentEvent e) {
}

@Override
public void componentMoved(ComponentEvent e) {

    int x = e.getComponent().getX();
    int y = e.getComponent().getY();

    labelx.setText("x: " + x);
    labely.setText("y: " + y);
}

@Override
public void componentShown(ComponentEvent e) {
}

@Override
public void componentHidden(ComponentEvent e) {
}
```

We have to create all four methods even it we are interested in one of them—`componentMoved()`.

Other three methods are empty.

```
int x = e.getComponent().getX();
int y = e.getComponent().getY();
```

Here we get the x and the y positions of the component.

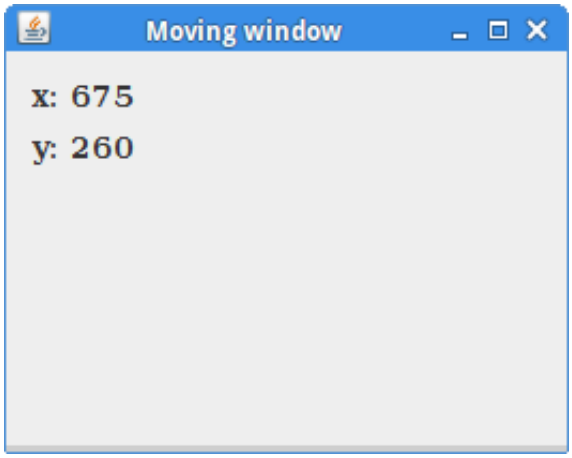

Figure: Moving a window

Adapters

An adapter is a convenient class that provides empty implementations all required methods. In the previous code example, we had to implement all four methods of a `ComponentListener` class—even if we did not use them. To avoid unnecessary coding, we can use adapters. We then use implement those methods that we actually need. There is no adapter for a button click event because there we have only one method to implement—the `actionPerformed()`. We can use adapters in situations where we have more than one method to implement.

The following example is a rewrite of the previous one, using a `ComponentAdapter`.

```
package com.zetcode;

import java.awt.Container;
import java.awt.EventQueue;
import java.awt.Font;
import java.awt.event.ComponentAdapter;
import java.awt.event.ComponentEvent;
import javax.swing.GroupLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class AdapterExample extends JFrame {

    private JLabel labelx;
    private JLabel labely;
```

```
    public AdapterExample() {

        initUI();
    }

    private void initUI() {

        Container pane = getContentPane();
        GroupLayout gl = new GroupLayout(pane);
        pane.setLayout(gl);

        addComponentListener(new MoveAdapter());

        labelx = new JLabel("x: ");
        labelx.setFont(new Font("Serif", Font.BOLD, 14));

        labely = new JLabel("y: ");
        labely.setFont(new Font("Serif", Font.BOLD, 14));

        gl.setAutoCreateContainerGaps(true);
        gl.setAutoCreateGaps(true);

        gl.setHorizontalGroup(gl.createParallelGroup()
                .addComponent(labelx)
                .addComponent(labely)
                .addGap(250)
        );

        gl.setVerticalGroup(gl.createSequentialGroup()
                .addComponent(labelx)
                .addComponent(labely)
                .addGap(130)
        );

        pack();

        setTitle("Adapter example");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private class MoveAdapter extends ComponentAdapter {

        @Override
```

```
        public void componentMoved(ComponentEvent e) {

            int x = e.getComponent().getX();
            int y = e.getComponent().getY();

            labelx.setText("x: " + x);
            labely.setText("y: " + y);
        }
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {

            @Override
            public void run() {
                AdapterExample ex = new AdapterExample();
                ex.setVisible(true);
            }
        });
    }
}
```

This example is a rewrite of the previous one. Here we use the `ComponentAdapter`.

```
addComponentListener(new MoveAdapter());
```

Here we register the component listener.

```
private class MoveAdapter extends ComponentAdapter {

    @Override
    public void componentMoved(ComponentEvent e) {

        int x = e.getComponent().getX();
        int y = e.getComponent().getY();

        labelx.setText("x: " + x);
        labely.setText("y: " + y);
    }
}
```

Inside the `MoveAdapter` inner class, we define the `componentMoved()` method. All the other methods are left empty.

This part of the Java Swing tutorial was dedicated to Swing events.