Java Swing first programs

In this chapter, we will program our first Swing programs. We will cover some basic functionality.

Our first example

The first example shows a basic window on the screen.

```
package com.zetcode;

import java.awt.EventQueue;
import javax.swing.JFrame;

public class SimpleEx extends JFrame {

    public SimpleEx() {

        initUI();
    }

    private void initUI() {

        setTitle("Simple example");
        setSize(300, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {

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

While this code is very short, the application window can do quite a lot. It can be resized, maximised, or minimised. All the complexity that comes with it has been hidden from the application programmer.

```
import java.awt.EventQueue;
import javax.swing.JFrame;
```

Here we import Swing classes that will be used in the code example.

```
public class SimpleEx extends JFrame {
```

The `SimpleEx` class inherits from the `JFrame` component. `JFrame` is a top-level container. The basic purpose of containers is to hold components of the application.

```
public SimpleEx() {

    initUI();
}
```

It is a good programming practice not to put the application code into constructors, but delegate the task to a specific method.

```
setTitle("Simple example");
```

Here we set the title of the window using the `setTitle()` method.

```
setSize(300, 200);
```

This code will resize the window to be 300 px wide and 200 px tall.

```
setLocationRelativeTo(null);
```

This line will center the window on the screen.

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

This method will close the window if we click on the close button of the titlebar. By default nothing happens.

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

We create an instance of our code example and make it visible on the screen. The `invokeLater()` method places the application on the Swing Event Queue. It is used to ensure that all UI updates are concurrency-safe. In other words, it is to prevent GUI from hanging in certain situations.

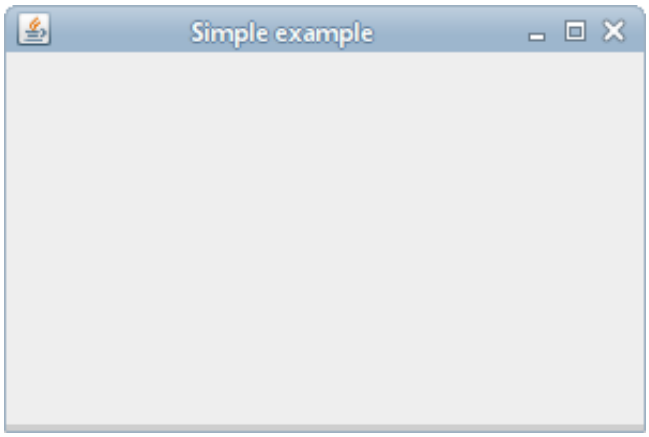

Figure: Simple example

Quit button

In our next example, we will have a button. When we click on the button, the application terminates.

```
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.JComponent;
import javax.swing.JFrame;

public class QuitButtonEx extends JFrame {

    public QuitButtonEx() {

        initUI();
    }

    private void initUI() {

        JButton quitButton = new JButton("Quit");

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

```
            }
        });

        createLayout(quitButton);

        setTitle("Quit button");
        setSize(300, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private void createLayout(JComponent... arg) {

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

        gl.setAutoCreateContainerGaps(true);

        gl.setHorizontalGroup(gl.createSequentialGroup()
                .addComponent(arg[0])
        );

        gl.setVerticalGroup(gl.createSequentialGroup()
                .addComponent(arg[0])
        );
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {

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

We position a `JButton` on the window and add an action listener to this button.

```
JButton quitButton = new JButton("Quit");
```

Here we create a button component. This constructor takes a string label as a parameter.

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

We plug an action listener to the button. The listener's `actionPerformed()` method will be called when we click on the button. The action terminates the application by calling the `System.exit()` method.

```
createLayout(quitButton);
```

The child components need to be placed into containers. We delegate the task to the `createLayout()` method.

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

The content pane of a `JFrame` is an area where child components are placed. The children are organised by specialised non-visible components called layout managers. The default layout manager of a content pane is the `BorderLayout` manager. This manager is very simple and is useful only in certain cases. In this tutorial, we use the `GroupLayout` manager which is more powerful and flexible.

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

The `setAutoCreateContainerGaps()` method creates gaps between components and the edges of the container. Space or gaps are important part of the design of each application.

```
gl.setHorizontalGroup(gl.createSequentialGroup()
        .addComponent(quitButton)
);

gl.setVerticalGroup(gl.createSequentialGroup()
        .addComponent(quitButton)
);
```

`GroupLayout` manager defines the layout for each dimension independently. In one step, we lay out components alongside the horizontal axis; in the other step, we lay out components along the vertical axis. In both kinds of layouts we can arrange components sequentially or in parallel. In a horizontal layout, a row of components is called a sequential group and a column of components is called a parallel group. In a vertical layout, a column of components is called a sequential group and a row of components a parallel

group.

In our example we have only one button, so the layout is very simple. For each dimension, we call the `addComponent()` method with the button component as a parameter. (Each child component must be added for both dimensions.)

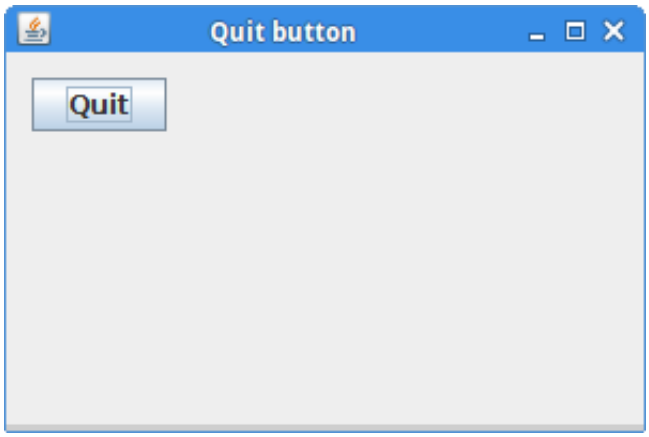

Figure: Quit button

A tooltip

Tooltips are part of the internal application's help system. Swing shows a small window if we hover a mouse pointer over an object that has a tooltip set.

```
package com.zetcode;

import java.awt.EventQueue;
import javax.swing.GroupLayout;
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JPanel;


public class TooltipEx extends JFrame {

    public TooltipEx() {

        initUI();
    }

    private void initUI() {

        JButton btn = new JButton("Button");
        btn.setToolTipText("A button component");

        createLayout(btn);
```

```
        setTitle("Tooltip");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private void createLayout(JComponent... arg) {

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

        pane.setToolTipText("Content pane");

        gl.setAutoCreateContainerGaps(true);

        gl.setHorizontalGroup(gl.createSequentialGroup()
                .addComponent(arg[0])
                .addGap(200)
        );

        gl.setVerticalGroup(gl.createSequentialGroup()
                .addComponent(arg[0])
                .addGap(120)
        );

        pack();
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {

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

In the example, we set a tooltip for the frame and the button.

```
btn.setToolTipText("A button component");
```

To enable a tooltip, we call the `setTooltipText()` method.

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

A content pane is an instance of a `JPanel` component. The `getContentPane()` method returns a `Container` type. Since setting a tooltip requires a `JComponent` instance, we cast the object to a `JPanel`.

```
pane.setToolTipText("Content pane");
```

A tooltip is set for the content pane.

```
gl.setHorizontalGroup(gl.createSequentialGroup()
        .addComponent(arg[0])
        .addGap(200)
);

gl.setVerticalGroup(gl.createSequentialGroup()
        .addComponent(arg[0])
        .addGap(120)
);
```

We call the `addGap()` method for horizontal and vertical dimensions. It creates some space to the right and to the bottom of the button.

```
pack();
```

The `pack()` method automatically sizes the `JFrame` based on the size of its components. It takes the defined space into account too. Our window will display the button and the spaces that we have set with the `addGap()` method.

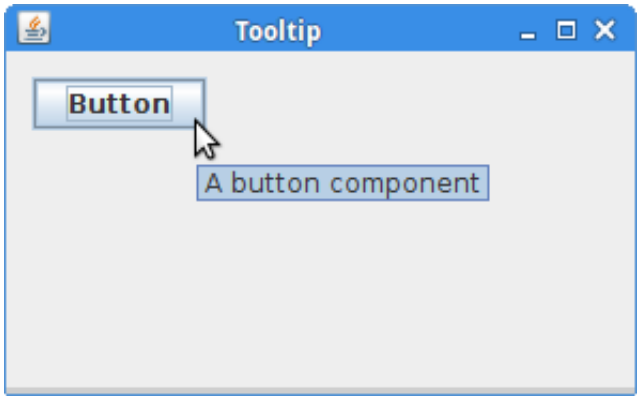

Figure: Tooltip

Mnemonics

Mnemonics are shortcut keys that activate a component that supports mnemonics. For instance, they can be used with labels, buttons, or menu items.

```
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.KeyEvent;
import javax.swing.GroupLayout;
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JFrame;


public class MnemonicEx extends JFrame {

    public MnemonicEx() {

        initUI();
    }

    private void initUI() {

        JButton btn = new JButton("Button");
        btn.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button pressed");
            }

        });

        btn.setMnemonic(KeyEvent.VK_B);

        createLayout(btn);

        setTitle("Mnemonics");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private void createLayout(JComponent... arg) {
```

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

        gl.setAutoCreateContainerGaps(true);

        gl.setHorizontalGroup(gl.createSequentialGroup()
                .addComponent(arg[0])
                .addGap(200)
        );

        gl.setVerticalGroup(gl.createParallelGroup()
                .addComponent(arg[0])
                .addGap(200)
        );

        pack();
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {

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

We have a button with an action listener. We set a mnemonic for this button. It can be activated with the Alt+B keyboard shortcut.

```
btn.setMnemonic(KeyEvent.VK_B);
```

The `setMnemonic()` method sets a keyboard mnemonic for the button. The mnemonic key is specified with a virtual keycode from the `KeyEvent` class. The mnemonic is combined with the look and feel's mouseless modifier (usually Alt).

At this moment, there are three ways to activate the button: a left mouse button click, the Alt+B shortcut, and the Space key (provided the button has the focus). The Space key binding was automatically created by Swing. (Under Metal look and feel, the focus is visually represented by a small rectangle around the

button's label.)

In this chapter, we have created some simple Java Swing programs.