To center the window on the screen, we must know the resolution of the monitor. For this, we use the *Toolkit* class.

```
Toolkit toolkit = getToolkit();

Dimension size = toolkit.getScreenSize();
```

In this example, we will see two new topics. Layout management and event handling. They will be touched only briefly. Both of the topics will have their own chapter.

```
JPanel panel = new JPanel();

getContentPane().add(panel);
```

We create a *JPanel* component. It is a generic lightweight container. We add the JPanel to the JFrame.

Creating a menubar

A menubar is one of the most visible parts of the GUI application. It is a group of commands located in various menus. While in console applications you had to remember all those arcane commands, here we have most of the commands grouped into logical parts. There are accepted standards that further reduce the amount of time spending to learn a new application.

In Java Swing, to implement a menubar, we use three objects. A **JMenuBar**, a **JMenu** and a **JMenuItem**.

We create a menu object. The menus can be accessed via the keybord as well. To bind a menu to a particular key, we use the **setMnemonic** method. In our case, the menu can be opened with the **ALT + F** shortcut.

```
fileClose.setToolTipText("Exit application");
```

A submenu is just like any other normal menu. It is created the same way. We simply add a menu to existing menu.

```
fileClose.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_W,

    ActionEvent.CTRL_MASK));
```

An accelerator is a key shortcut that launches a menu item. In our case, by pressing **Ctrl + W** we close the application.

```
file.addSeparator();
```

We put a vertical toolbar to the left side of the window. This is typical for graphics applications like **Xara Extreme** or **Inkscape**. In our example, we use icons from Xara Extreme application.

```
JToolBar toolbar = new JToolBar(JToolBar.VERTICAL);
```

We create a vertical toolbar.

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 it's life. Events are generated mainly by the user of an application. But they can be generated by other means as well. e.g. inter net 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 list. There are several types of events. An **ActionEvent**, **TextEvent**, **FocusEvent**, **ComponentEvent** etc. Each of them is created under specific conditions.

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

Inside the action listener, we have an event parameter. It is the instance of the event, that has occured. In our case it is an **ActionEvent**.

```
cal.setTimeInMillis(event.getWhen());
```

Here we get the time, when the event occured. The method returns time value in milliseconds. So we must format it appropriately.

```
String source = event.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**.

Implementation

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

- Anonymous inner class
- Inner class
- Derived class