

## Article

# New and Updated Desktop Features in Java SE 6, Part 2

By *Robert Eckstein*, February 2007

[Articles Index](#)

Part 1 of this article discussed several new or updated features available for the Java desktop developer in the final release of [Java Platform, Standard Edition 6](#) (Java SE 6), including splash screens, the system tray, gray rect fix, LCD text, single-threaded rendering, and native look and feel. This article continues the outline, with details about the following elements.

## Contents

- [Table Sorting and Filtering](#)
- [Image I/O Library Improvements](#)
- [The New Modality Model](#)
- [The Desktop API](#)

## Table Sorting and Filtering

*From the Core Java Technologies Tech Tip "Sorting and Filtering Tables" by John O'Conner*

Java SE 6 adds some features that make sorting and filtering the contents of a Swing `JTable` much easier. Most modern table-driven user interfaces allow users to sort columns by clicking on the table header. This could be done with the Swing `JTable` support in place prior to Java SE 6. However, the developer had to add the functionality manually in a custom way for each table that needed this feature. With Java SE 6, enabling this functionality requires minimal effort. Filtering is another option commonly available with user interfaces. Filtering allows users to display only the rows in a table that match user-supplied criteria. With Java SE 6, enabling filtering of `JTable` contents is also much easier.

### Sorting Table Rows

The basis for sorting and filtering rows in Java SE 6 is the abstract `RowSorter` class. `RowSorter` maintains two mappings, one of rows in a `JTable` to the elements of the underlying model and the other from the underlying model back to the rows in the `JTable`. This allows a programmer to do sorting and filtering. The class is generic enough to work with both `TableModel` and `ListModel`. However, only a `TableRowSorter` is provided with the Java SE 6 libraries to work with `JTable`.

In the simplest case, you pass the `TableModel` to the `TableRowSorter` constructor, and you then pass the created `RowSorter` into the `setRowSorter()` method of `JTable`. Here's an example program that demonstrates the approach:

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

public class SortTable {
    public static void main(String args[]) {
        Runnable runner = new Runnable() {
            public void run() {
                JFrame frame = new JFrame("Sorting JTable");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                Object rows[][] = {
                    {"AMZN", "Amazon", 41.28},
                    {"EBAY", "eBay", 41.57},
                    {"GOOG", "Google", 388.33},
                    {"MSFT", "Microsoft", 26.56},
                    {"NOK", "Nokia Corp", 17.13},
                    {"ORCL", "Oracle Corp.", 12.52},
                    {"SUNW", "Sun Microsystems", 3.86},
                    {"TWX", "Time Warner", 17.66},
                    {"VOD", "Vodafone Group", 26.02},
                    {"YHOO", "Yahoo!", 37.69}
                };
                String columns[] = {"Symbol", "Name", "Price"};
                TableModel model =
                    new DefaultTableModel(rows, columns) {
                        public Class getColumnClass(int column) {
```

```

        Class returnValue;
        if ((column >= 0) && (column < getColumnCount())) {
            returnValue = getValueAt(0, column).getClass();
        } else {
            returnValue = Object.class;
        }
        return returnValue;
    }
};

JTable table = new JTable(model);
RowSorter<TableModel> sorter =
    new TableRowSorter<TableModel>(model);
table.setRowSorter(sorter);
JScrollPane pane = new JScrollPane(table);
frame.add(pane, BorderLayout.CENTER);
frame.setSize(300, 150);
frame.setVisible(true);
}
};
EventQueue.invokeLater(runner);
}
}

```

Figure 1 shows the output.

Symbol	Name	Price
AMZN	Amazon	41.28
EBAY	eBay	41.57
GOOG	Google	388.33
MSFT	Microsoft	26.56
NOK	Nokia Corp	17.13
ORCL	Oracle Corp.	12.52

Figure 1. Initial JTable Without Sorting

If you run the example, then click on one of the columns of the displayed table, the contents of the column will reorder, as shown in Figure 2.

Symbol	Name	Price
SUNW	Sun Microsyst...	3.86
ORCL	Oracle Corp.	12.52
NOK	Nokia Corp	17.13
TWX	Time Warner	17.66
VOD	Vodafone Group	26.02
MSFT	Microsoft	26.56

Figure 2. JTable After Sorting by Price

Why not simply use the `DefaultTableModel`, as opposed to creating a custom subclass of it? The answer is that `TableRowSorter` has a set of rules to follow for sorting columns. By default, all columns of a table are thought to be of type `Object`. So sorting is done by calling `toString()`. By overriding the default `getColumnClass()` behavior of `DefaultTableModel`, `RowSorter` sorts according to the rules of that class, assuming it implements `Comparable`. You can also install a custom `Comparator` for a column:

```
setComparator(int column, Comparator comparator)
```

Here are the key lines in the `SortTable` program that are pertinent to sorting:

```

JTable table = new JTable(model);
RowSorter<TableModel> sorter =
    new TableRowSorter<TableModel>(model);
table.setRowSorter(sorter);

```

The first line associates the model with the table. The second line and its continuation create a `RowSorter` specific to the model. The final line associates the `RowSorter` with the `JTable`. This enables a user to click on the column header to sort that column. Clicking a second time on the same column reverses the sort order.

If you want to add your own action when the sort order changes, you can attach a `RowSorterListener` to the `RowSorter`. The interface has one method:

```
void sorterChanged(RowSorterEvent e)
```

The method allows you to update the text on the status bar or perform some additional task. The `RowSorterEvent` for the action allows you to discover how many rows were present before the sort, in the event that the `RowSorter` filters rows in or out of the view.

### Filtering Table Rows

You can associate a `RowFilter` with the `TableRowSorter` and use it to filter the contents of a table. For instance, you can use a `RowFilter` such that a table displays only rows in which, to use the example program shown earlier, the company name starts with the letter A or in which the stock price is greater than \$50. The abstract `RowFilter` class has one method that is used for filtering:

```
boolean include(RowFilter.Entry<? extends M,? extends I> entry)
```

For each entry in the model associated with the `RowSorter`, the method indicates whether the specified entry should be shown in the current view of the model. In many cases, you don't need to create your own `RowFilter` implementation. Instead, `RowFilter` offers six static methods for creating filters.

```
public static <M,I> RowFilter<M,I>
    andFilter(Iterable<? extends RowFilter<? super M,? super I>> filters)

public static <M,I> RowFilter<M,I>
    dateFilter(RowFilter.ComparisonType type, Date date, int... indices)

public static <M,I> RowFilter<M,I>
    notFilter(RowFilter<M,I> filter)

public static <M,I> RowFilter<M,I>
    numberFilter(RowFilter.ComparisonType type, Number number, int... indices)

public static <M,I> RowFilter<M,I>
    orFilter(Iterable<? extends RowFilter<? super M,? super I>> filters)

public static <M,I> RowFilter<M,I>
    regexFilter(String regex, int... indices)
```

For the `RowFilter` factory methods that have an argument of indices -- `dateFilter`, `numberFilter`, and `regexFilter` -- the example program checks only the set of columns corresponding to the specified indices. If no indices are specified, the program checks all the columns for a match.

The `dateFilter` allows you to check for a matching date. The `numberFilter` checks for a matching number. The `notFilter` is used for reversing another filter. That is, it includes entries that the supplied filter does not include. You can use it to do things such as finding entries in which a particular action was *not* done on, for example, December 25, 2006. The `andFilter` and `orFilter` are for logically combining other filters. The `regexFilter` uses a regular expression for filtering. Here's a program, `FilterTable`, that uses a `regexFilter` to filter table content:

```
import javax.swing.*;
import javax.swing.table.*;
import java.awt.*;
import java.awt.event.*;
import java.util.regex.*;

public class FilterTable {
    public static void main(String args[]) {
        Runnable runner = new Runnable() {
            public void run() {
                JFrame frame = new JFrame("Sorting JTable");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                Object rows[][] = {
                    {"AMZN", "Amazon", 41.28},
                    {"EBAY", "eBay", 41.57},
                    {"GOOG", "Google", 388.33},
                    {"MSFT", "Microsoft", 26.56},
                };
            }
        };
        SwingUtilities.invokeLater(runner);
    }
}
```

```

        {"NOK", "Nokia Corp", 17.13},
        {"ORCL", "Oracle Corp.", 12.52},
        {"SUNW", "Sun Microsystems", 3.86},
        {"TWX", "Time Warner", 17.66},
        {"VOD", "Vodafone Group", 26.02},
        {"YHOO", "Yahoo!", 37.69}
    };
    Object columns[] = {"Symbol", "Name", "Price"};
    TableModel model =
        new DefaultTableModel(rows, columns) {
        public Class getColumnClass(int column) {
            Class returnValue;
            if ((column >= 0) && (column < getColumnCount())) {
                returnValue = getValueAt(0, column).getClass();
            } else {
                returnValue = Object.class;
            }
            return returnValue;
        }
    };
    JTable table = new JTable(model);
    final TableRowSorter<TableModel> sorter =
        new TableRowSorter<TableModel>(model);
    table.setRowSorter(sorter);
    JScrollPane pane = new JScrollPane(table);
    frame.add(pane, BorderLayout.CENTER);
    JPanel panel = new JPanel(new BorderLayout());
    JLabel label = new JLabel("Filter");
    panel.add(label, BorderLayout.WEST);
    final JTextField filterText =
        new JTextField("SUN");
    panel.add(filterText, BorderLayout.CENTER);
    frame.add(panel, BorderLayout.NORTH);
    JButton button = new JButton("Filter");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String text = filterText.getText();
            if (text.length() == 0) {
                sorter.setRowFilter(null);
            } else {
                try {
                    sorter.setRowFilter(
                        RowFilter.regexFilter(text));
                } catch (PatternSyntaxException pse) {
                    System.err.println("Bad regex pattern");
                }
            }
        }
    });
    frame.add(button, BorderLayout.SOUTH);
    frame.setSize(300, 250);
    frame.setVisible(true);
}
};
EventQueue.invokeLater(runner);
}
}

```

The display sets a filter for all strings with the characters `SUN` somewhere in them. This is specified by the string `SUN`. Use the characters `^` and `$` to test for exact matches at the beginning and end of the string, respectively. See Figure 3.



Figure 3. Entering a Table Filter

The filter internally uses `Matcher.find()` for inclusion testing when the user presses the Filter button at the bottom of the pane. See Figure 4.

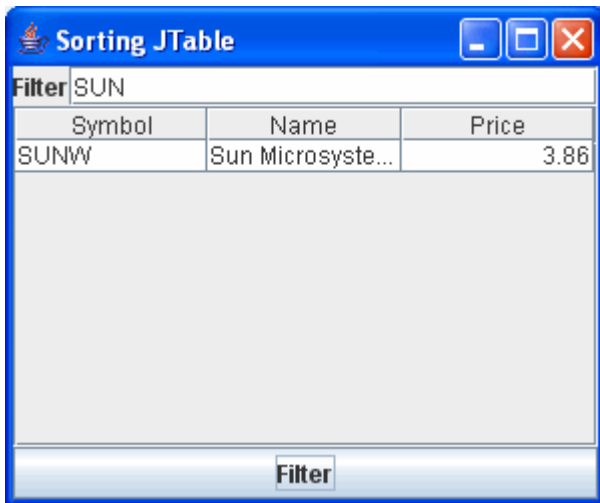


Figure 4. The Filtered Results

Change the filter text to change the set of rows shown in the table. If you want to see all the rows in the table, remove the filter text.

## Image I/O Library Improvements

From the blog entry *'400 Horsepower: Image I/O Improvements in JDK 6'* by Chris Campbell

In JDK 5.0, the desktop team did some performance work in the Image I/O APIs, specifically to avoid finalization in the `com.sun.imageio.plugins.jpeg.JPEGImageReader` class, which made for huge gains in scalability and performance of reading JPEG images. Next, the team spent a few solid weeks digging deep into the Image I/O API framework and the core plug-ins to find ways to improve performance. The team fixed the following Image I/O-related bugs:

- [6347575](#): `FileImageInputStream.readInt()` and similar methods are inefficient.
- [6348744](#): `PNGImageReader` should skip metadata if `ignoreMetadata=true`.
- [6354056](#): `JPEGImageReader` could be optimized.
- [6299405](#): `ImageInputStreamImpl` still uses a `finalize()`, which causes `java.lang.OutOfMemoryError`.
- [6354112](#): Increase compiler optimization level for `libjpeg` to improve runtime performance

Figure 5 shows the improvements.

# Image I/O Improvements in Java SE 6

(2x 2.8 GHz P4, 1 GB RAM, Windows XP, 32-bit Client VM)

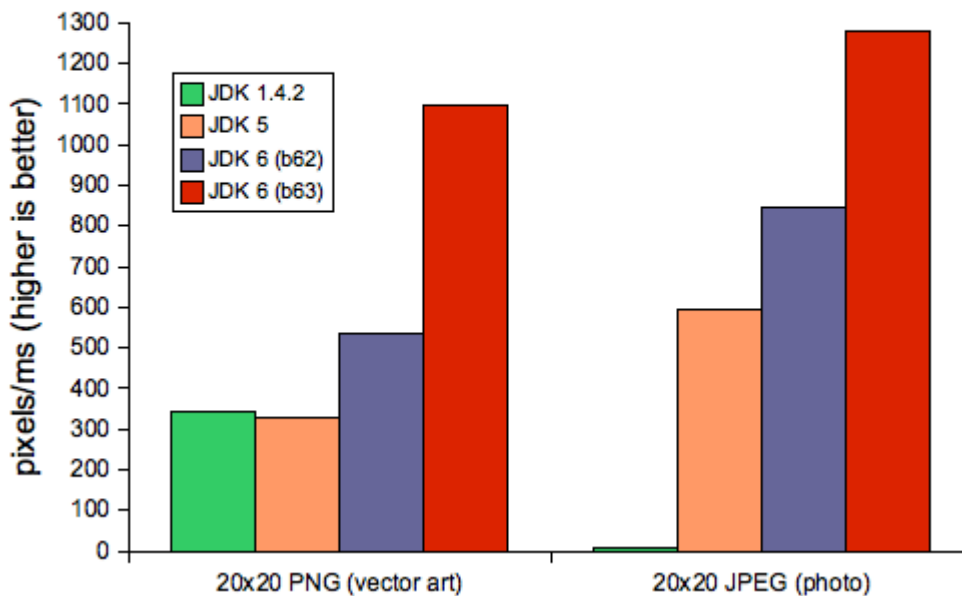


Figure 5. Image I/O Improvements in Java SE 6

Note that Figure 5's `JPEG` score of 20x20 for JDK 1.4.2 isn't a glitch -- it really was that bad. Also, note that this chart shows results only for small images, but similar improvements can be seen for larger images as well. For example, the desktop team measured a 53 percent improvement in reading 4000x4000 pixel `JPEG` images between JDK 5.0 and JDK 6.

## The New Modality Model

From the article "[The New Modality API in Java SE 6](#)" by Artem Ananiev and Dana Nourie

A *dialog box* is a top-level pop-up window with a title and a border that is typically used to take some form of input from the user. With `JDK 5.0` and earlier, a dialog box must have either a frame or another dialog box defined as its owner when the dialog box is constructed, even if the box is invisible. When the user minimizes the owner window of a visible dialog box, this automatically hides the dialog box from the user. When the user subsequently restores the owner window, the dialog box becomes visible again.

A dialog box can be either modeless or modal. A *modal dialog box* is one that blocks input to some other top-level windows in the application, except for any windows created with the dialog box as their owner. The modal dialog box captures the window focus until it is closed, usually in response to a button press. A *modeless dialog box*, on the other hand, sits off on the side and allows you to change its state while other windows have focus. The latter is often used for a toolbar window, such as what you might find in an image-editing program.

This was the limit of the modality model in `JDK 5.0` and earlier versions. However, this modality model was not without problems. Perhaps the most famous issue involved `JavaHelp` tool windows. `JavaHelp`, an API to display help information from a Java technology-based application, uses a separate window to display all the necessary information. However, if an application shows any modal dialog box, such as a standard Save As dialog box, that dialog box blocks the user from interacting with the `JavaHelp` tool window.

Java SE 6 has resolved this issue and several others with a new Abstract Window Toolkit (AWT) modality model. This new model allows the developer to *scope*, or limit, a dialog box's modality blocking, based on the modality type that the developer chooses. By doing so, the modality type also allows windows and dialog boxes to be truly *parentless*, that is, to have a null parent, which helps to limit the scope of the windows' and dialog boxes' modality.

Java SE 6 technology supports four modality types:

- **Modeless.** A modeless dialog box does not block any other window while it is visible.
- **Document-modal.** A document-modal dialog box blocks all windows from the same document, except those from its child hierarchy. In this context, a *document* is a hierarchy of windows -- frames, dialog boxes, and so on -- that share a common ancestor, the *document root*, which is the closest ancestor window without an owner.
- **Application-modal.** An application-modal dialog box blocks all windows from the same application, except for those from its child hierarchy. If several applets are launched in a browser environment, the browser is allowed to treat them either as separate applications or as a single application. The behavior is implementation-dependent.

- **Toolkit-modal.** A toolkit-modal dialog box blocks all windows that run in the same toolkit, except those from its child hierarchy. If several applets are launched, all of them run with the same toolkit. Hence, a toolkit-modal dialog box shown from an applet may affect other applets and all windows of the browser instance that embeds the Java Runtime Environment (JRE) for this toolkit.

As with previous JDKs, a dialog box is modeless by default. But if you construct a modal dialog box in Java SE 6, it will now use the application-modal type by default. In addition, the behavior of both modal and modeless dialog boxes has changed so that they always appear on top of their parent window.

*Modality priority* is defined by the strength of blocking. The modality priority helps in situations in which two dialog boxes are visible and could block each other. Here is the priority in ascending order from weakest to strongest: modeless, document-modal, application-modal, and toolkit-modal. Such a priority naturally reflects the nesting of a dialog box's scope of blocking. A modeless dialog box has an empty scope of blocking. A document-modal dialog box's scope of blocking is complete in some applications. And all the applications are run in one toolkit. Figure 6 shows an example.

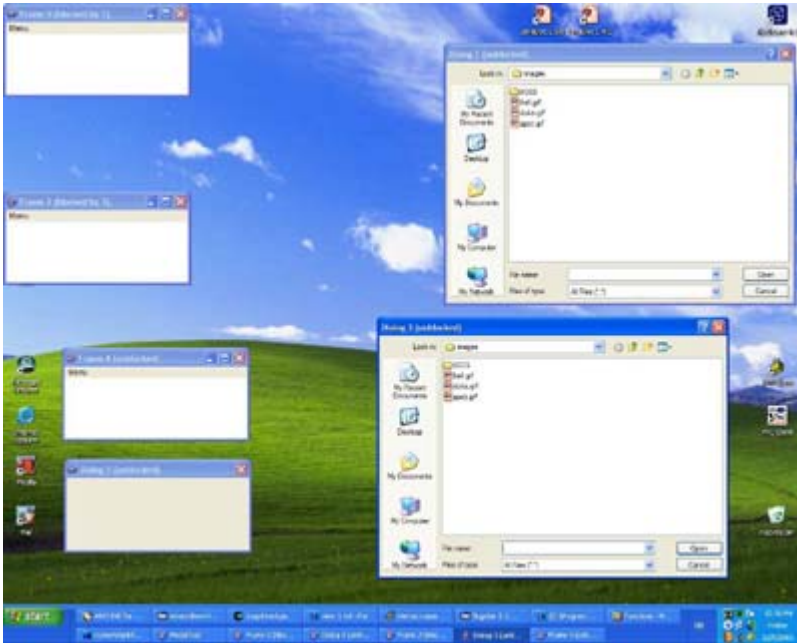


Figure 6. Modality Priority of Dialog Boxes  
[Click here for a larger image](#)

Note that the new modality model does not implement a *system modality*, which would block all applications -- Java technology-based or otherwise -- that are displayed on the desktop while a modal dialog box is active.

Adding the ability to offer truly parentless windows without breaking backward compatibility was a challenge for the AWT team. In version JDK 5.0 or earlier, you were allowed to pass `null` as a parent for `JDialog` or `JWindow`, but that automatically meant that an invisible, shared-owner frame became a parent of this dialog box or window. The shared-owner frame was invented to give the illusion of creating parentless dialog boxes. This was fine until Java SE 6, which introduced new document-modal dialog boxes that block all the windows from the same document.

Consequently, the toolkit saw such dialog boxes or windows as not having a null parent. In Java SE 6, you are still allowed to pass `null` as a parent to the older `JDialog` or `JWindow` constructors. And this accomplishes the same thing: The shared-owner frame again becomes the parent, in order to preserve backward compatibility. However, you can now pass `null` to the `Dialog` or `Window` constructors as well as to the new `JDialog` or `JWindow` constructors, which means that the dialog boxes really will be parentless.

Some of the available constructors are as follows:

- `JDialog(Dialog owner)`  
Creates a modeless dialog box without a title with the specified `Dialog` as its owner
- `JDialog(Dialog owner, boolean modal)`  
Creates a dialog box with the specified owner `Dialog` and modality
- `JDialog(Dialog owner, String title)`  
Creates a modeless dialog box with the specified title and with the specified owner dialog box
- `JDialog(Dialog owner, String title, boolean modal)`  
Creates a dialog box with the specified title, modality, and owner `Dialog`

- `JDialog(Dialog owner, String title, boolean modal, GraphicsConfiguration gc)`  
Creates a dialog box with the specified title, owner `Dialog`, modality, and `GraphicsConfiguration`
- `JDialog(Frame owner)`  
Creates a modeless dialog box without a title with the specified `Frame` as its owner
- `JDialog(Window owner, String title, Dialog.ModalityType modalityType)`  
Creates a dialog box with the specified title, owner `Window`, and modality

As mentioned earlier, when one window or a dialog box is the parent of another, that parent component is said to "own" its child. Here are a few things that you should be aware of when working with owners in the new modality model:

- **Creating a document-modal dialog box without an owner.** In this case, because `Dialog` is a subclass of `Window`, a `Dialog` instance automatically becomes the root of the document if it has no owner. Thus, if such a dialog box is document-modal, its scope of blocking is empty, and it behaves the same way as would a modeless dialog box.
- **Creating an application-modal or toolkit-modal dialog box with an owner.** The scope of blocking for an application-modal or toolkit-modal dialog box, as opposed to the scope of blocking for a document-modal dialog box, does not depend on its owner. In this case, the only thing that the owner affects is the *Z-order*, the relative order of top-level components. If you have two windows, one overlaps the other, with the first one above or on top of the second. The topmost window is usually an active window. A related notion is that of always on top, in which one window always appears above all other windows in the system. The dialog box always stays on top of its owner.
- **Changing the modality type at runtime.** Changing the modality type for a visible dialog box may have no effect until the dialog box is hidden and then shown again.

A dialog box that uses `DOCUMENT_MODAL` blocks input to all top-level windows from the same document, except those from its own child hierarchy. A document is a top-level window without an owner. It may contain child windows that are treated as a single document together with the top-level window. Because every top-level window must belong to some document, its root can be found at the top-nearest window without an owner.

```
d22.setBounds(sw - 500 + 32, 232, 300, 200);
d22.addWindowListener(closeWindow);
d22.setLayout(new BorderLayout());
l = new Label("DOCUMENT_MODAL");
l.setBackground(Color.BLUE);
l.setAlignment(Label.CENTER);
l.setFont(labelFont);
d22.add(l, BorderLayout.CENTER);

// Third document

f3 = new Frame("Excluded Frame");

f3.setModalExclusionType(
    Dialog.ModalExclusionType.APPLICATION_EXCLUDE);
```

A dialog box set to `APPLICATION_MODAL` blocks all top-level windows from the same Java platform application, except those from its own child hierarchy. If several applets are launched in a browser, they can be treated either as separate applications or as a single one. This behavior is implementation-dependent.

Note that here, `f3` will not be blocked by `APPLICATION_MODAL` and `DOCUMENT_MODAL` dialog boxes:

```
f3.setBounds(32, sh - 200 + 32, 300, 200);
f3.addWindowListener(closeWindow);
f3.setLayout(new BorderLayout());
l = new Label("EXCLUDED FRAME");
l.setBackground(Color.GREEN);
l.setAlignment(Label.CENTER);
l.setFont(labelFont);
f3.add(l, BorderLayout.CENTER);
b = new Button("I'm alive!");
f3.add(b, BorderLayout.SOUTH);
f3.setVisible(true);

// Fourth document

f4 = new Frame("Parent Frame");
f4.setBounds(sw - 300 + 32, sh - 200 + 32, 300, 200);
f4.addWindowListener(closeWindow);
```

```

f4.setLayout(new BorderLayout());
l = new Label("FRAME");
l.setBackground(Color.GRAY);
l.setAlignment(Label.CENTER);
l.setFont(labelFont);
b = new Button("Show file dialog");
b.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        fd4.setVisible(true);
    }
});
f4.add(b, BorderLayout.SOUTH);
f4.setVisible(true);
fd4 = new FileDialog(f4, "File Dialog", FileDialog.LOAD);

```

File dialog boxes are `APPLICATION_MODAL` by default for backward compatibility.

```

    fd4.setBounds(sw - 400 + 32, sh - 300 + 32, 300, 200);
}
}

```

## The Desktop API

From the article *"Using the Desktop API in Java SE 6"* by John O'Conner

Java SE 6 includes the [Desktop API](#). This API allows Java technology-based applications to interact with the default applications associated with specific file types on the host platform. Specifically, the new Desktop API allows your Java applications to do the following:

- Launch the host system's default browser with a specific Uniform Resource Identifier (URI)
- Launch the host system's default email client
- Launch applications to open, edit, or print files associated with those applications

The Desktop API uses your host operating system's file associations to launch applications associated with specific file types. For example, if OpenDocument text file extensions, `.odt`, are associated with the [OpenOffice](#) Writer application, your Java application could launch OpenOffice Writer to open, edit, or even print files with that association. Depending on your host system, different applications may be associated with each action.

Use the `Desktop.isDesktopSupported()` method to determine whether the Desktop API is available. On the Solaris Operating System and the Linux platform, this API is dependent on Gnome libraries. If those libraries are unavailable, this method will return `false`. After determining that the API is supported, that is, the `isDesktopSupported()` returns `true`, the application can retrieve a `Desktop` instance using the static `getDesktop()` method:

```

Desktop desktop = null;
// Before more Desktop API is used, first check
// whether the API is supported by this particular
// virtual machine (VM) on this particular host.
if (Desktop.isDesktopSupported()) {
    desktop = Desktop.getDesktop();
    ...
}

```

If your application doesn't check for API support using `isDesktopSupported()` before calling `getDesktop()`, it must be prepared to catch an `UnsupportedOperationException`, which is thrown when your application requests a `Desktop` instance on a platform that does not support these features. Additionally, if your application runs in an environment without a keyboard, mouse, or monitor, that is, a "headless" environment, the `getDesktop()` method will throw a `java.awt.HeadlessException`.

Once retrieved, the `Desktop` instance allows your application to browse, mail, open, edit, or even print a file or URI, but only if the retrieved `Desktop` instance supports these activities. Each of these activities is called an *action*, and each is represented as a `Desktop.Action` enumeration instance:

- **BROWSE**. Represents a browse action performed by the host's default browser
- **MAIL**. Represents a mail action performed by the host's default email client
- **OPEN**. Represents an open action performed by an application associated with opening a specific file type
- **EDIT**. Represents an edit action performed by an application associated with editing a specific file type
- **PRINT**. Represents a print action performed by an application associated with printing a specific file type

Before invoking any of these actions, an application should determine whether the `Desktop` instance supports them. This is different from determining whether a `Desktop` instance is available. The `Desktop.isDesktopSupported()` method tells you whether an instance can

be created. Once a `Desktop` object is acquired, you can query the object to find out which specific actions are supported.

For example, calling the following instance method will open your host's default browser:

```
public void browse(URI uri) throws IOException
```

Applications can launch the host's default email client, if that action is supported, by calling this `Desktop` instance method:

```
public void mail(URI uri) throws IOException
```

In the same manner, Java applications can open, edit, and print files from their associated application using a `Desktop` object's `open()`, `edit()`, and `print()` methods, respectively.

Interestingly, different applications may be registered for these different actions even on the same file type. For example, the Mozilla Firefox browser may be launched for the `OPEN` action, Emacs for the `EDIT` action, and yet a different application for the `PRINT` action. Your host desktop's associations are used to determine what application to invoke. JDK 6 does not allow the developer to manipulate desktop file associations. Those associations can be created or changed only with platform-dependent tools at this time.

## Conclusion

The final release of Java SE 6 has added or updated several features to help the Java desktop developer create desktop applications more efficiently. Download [Java SE 6](#) to take advantage of these improvements.

---

\* As used on this web site, the terms "Java Virtual Machine" or "JVM" mean a virtual machine for the Java platform.

## For More Information

Java Platform, Standard Edition 6 (Java SE 6)

- [New and Updated Desktop Features in Java SE 6, Part 1](#)
- [Update: Desktop Java Features in Java SE 6 \(January 2006\)](#)

JTable Sorting

- [Documentation for RowSorter, TableRowSorter, and RowFilter classes](#)

Desktop API

- [Desktop API](#)
- Download the [DesktopDemo](#) application

## Rate and Review

Tell us what you think of the content of this page.

Excellent  Good  Fair  Poor

Comments:



Your email address (no reply is possible without an address):

[Sun Privacy Policy](#)

Note: We are not able to respond to all submitted comments.

Submit »