Tuning Java Swing applications for Mac OS X

Sven Van Caekenberghe
sven@beta9.be
Beta Nine B.V.B.A.

Version 1 - October 1, 2002

Abstract

Mac OS X runs any standard Java Swing application very well. The application will even inherit the Aqua look and feel automatically. In other words, it just works! But with a little bit more work, you can polish your application to conform to the full Mac OS X User Experience. In this article we show you how. The result will be a standard Java Swing application that is virtually undistinguishable from a native Mac OS X application.

Contents

1 Introduction **2**
- 1.1 Java on Mac OS X 2
- 1.2 The Mac OS X User Experience 2
- 1.3 Outline 3

2 Tools **5**
- 2.1 J2SE 5
- 2.2 MRJToolkit 5
- 2.3 MRJAppBuilder 5

3 Tuning your Java Swing Application **6**
- 3.1 Handling Basic Apple Events 6
- 3.2 Packaging your Application 9
- 3.3 Adding your own Icon 10
- 3.4 File Dialogs 12
- 3.5 About Boxes 13
- 3.6 Preferences 13
- 3.7 Customizing the Menu Bar 15
- 3.8 Supporting Finder Drag and Drop 16
- 3.9 Standard Help 19
- 3.10 Basic Multi-Document Support 20

4 Conclusion **21**

1 Introduction

1.1 Java on Mac OS X

Mac OS X combines the rock-solid reliability of UNIX with the ease of use of Macintosh. But best of all, it is the only high-volume desktop OS to ship with all of J2SE built in: a highly-optimized, tightly integrated implementation of Java 2 Standard Edition 1.3.1, including native preemptive multitasking threads, automatic multiprocessor support, hardware graphics acceleration, as well as full Java Plugin and Java Web Start support. Together with Apple's gorgeous hardware like the PowerBook G4 Titanium, Mac OS X is emerging as the best place to develop and deploy your Java applications.

1.2 The Mac OS X User Experience

Standard Java Swing applications will automatically inherit the famous Aqua look and feel. Double-click a JAR file, view a Web page with Java Plugin Applets or load a Java Web Start JNLP file and off you go. It just works!

However, any Mac OS X user will quickly see that your application is not a native Mac OS X application. Little, but very visible details are off:

- the application has no nice icon, nor is it packaged as other applications
- the menu bar is mostly missing, looks weird and contains strange (Java class) names
- some of the fundamental menu items that are present are non-functional
- some windows have their own menu bar
- familiar keyboard shortcuts don't work as expected
- preferences files are often saved smack in a user's clean home directory
- file dialogs are non-standard
- it's not possible to drop files on an application icon
- the standard access to help is unavailable

In this article we'll show you how easy it is to tune your application to fix all of these problems. Best of all, we can do all this and maintain full standard Java Swing compatibility. In other words, the application will continue to run on other platforms as before.

On a more fundamental level, many Java Swing applications developed on other platforms, miss the real Mac OS X User Experience touch. Eye candy is just one thing, easy of use and user-centered design are much more important:

- error messages as well a the possible user responses should be clear and understandable,

- Mac OS X toolbars are much simpler and less cluttered
- not only can there be only one menu bar, but its format and layout, as well as standard keyboard shortcuts, have to comply to guidelines
- Mac OS X doesn't use complex, single-window, multi-document interfaces
- dialogs and windows should have clear, consistent and simple designs
- setup actions and installers should be avoided

This list is far from complete, but you get the idea. An excellent introduction into these issues can be found in [3]. The authority on the Mac OS X User Experience is of course [4]. In our simple examples, we tried to incorporate most of these requirements. In larger projects, developing, deploying and testing should occur on all targeted platforms, by engineers comfortable and experienced with those platforms.

1.3 Outline

In this article we will present two examples that we will adapt to Mac OS X:

- **Java Properties** is a tool type application that shows information about the current, default Java VM
- **JPEG Viewer** is a multi-document application to view JPEG image files

The examples are, like all examples, a bit unnatural: they were chosen to illustrate the two main application categories and designed so that we could apply the important techniques in a simple way. Either example could be extended and improved in many ways, this is left as an exercise. See figure 1 for a screenshot of what our finished Java Properties application will look like, once tuned for Mac OS X.

These are the techniques for tuning your Java Swing application for Mac OS X that we will explain:

Handling Basic Apple Events Each Mac OS X application has to respond to a number of basic OS events called Apple Events. Doing so will make your application respond to the menu items that the OS will add automatically to your application.

Packaging your Application Mac OS X applications look a certain way in the finder because they are packaged. It is easy to do this for Java applications.

Adding your own Icon No application, least of all a Mac OS X application is complete without a proper icon. Making a custom icon is hard, adding one is easy.

File Dialogs Users expect the file dialogs they are used to, so use them.

About Boxes Some simple code that produces About Boxes that look like those from the Cocoa frameworks.

Preferences In Mac OS X you should save your preferences in a particular place.

Customizing the Menu Bar Forcing your Java application to show a Mac like menu bar is easy to start with. But going all the way can be harder than you think.

Supporting Finder Drag and Drop Users should be able to drag and drop files onto your application. Once you know how, this is easy too.

Standard Help Mac OS X help is normally shown using an application called Help Viewer. Let's make our Java applications do the same.

Basic Multi-Document Support Proper Mac OS X menu bar support was not easy, proper Mac OS X multi-document support is even harder. We show you one way to start.

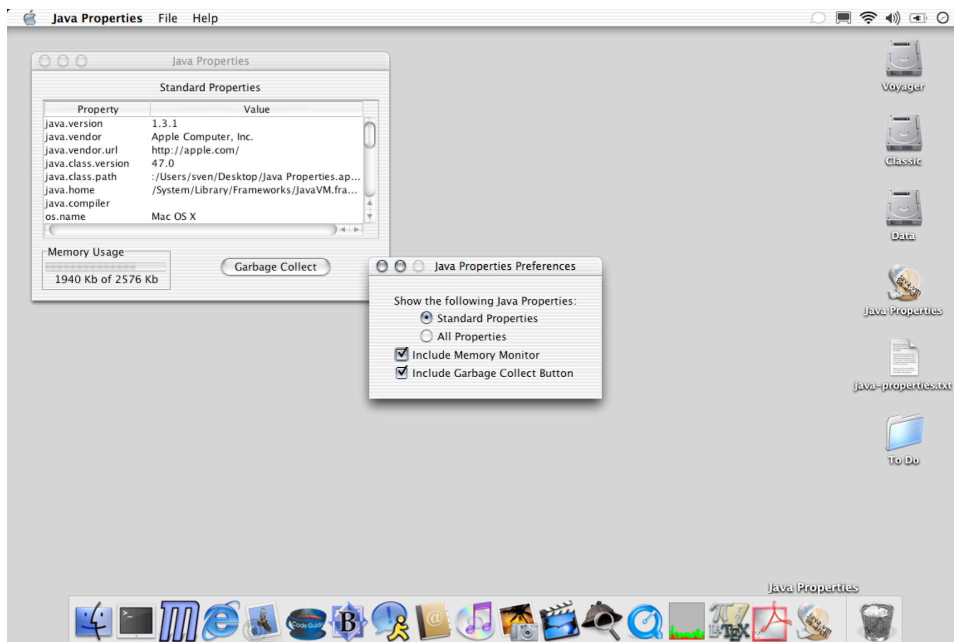

Figure 1: The Java Properties application, tuned for Mac OS X

Both the two finished example applications as well as the full source code can be downloaded from the following `.Mac` account:

`http://homepage.mac.com/svc`

2 Tools

2.1 J2SE

An implementation of Java 2 Standard Edition (J2SE) is an integral, standard, non-optional part of Mac OS X. This includes support for Java Web Start and the Java Plugin. For running Java applications this is all that is needed.

To develop Java applications, it is best to install the Mac OS X Development Tools. This is an extra, optional install that comes for free with most retail distributions of Mac OS X. If necessary, you can download the tools from the Apple Developer Connection, using a free Online Program account.

Most other well known Java tools and development environments work on Mac OS X. For these examples, we used:

- **Ant**, a Java-based build tool, similar to Make, but using XML
  See http://jakarta.apache.org/ant/
- **CodeGuide 5**, a Java IDE featuring on-the-fly error checking of the whole project, refactoring, instant compilation, ...
  See http://www.omnicore.com/htmls/codeguide.html

A detailed discussion of what runs well on Mac OS X is beyond the scope of this article.

2.2 MRJToolkit

MRJToolkit (MRJ stands for Macintosh Runtime for Java[1]) is a small library that offers access to some Mac specific functionality from Java. In the rest of this article we will be using the MRJToolkit API a lot.

The nice thing about MRJToolkit is that there is a dummy library available that essentially does nothing. By adding this dummy library to your classpath, code that uses this API will keep on working on other Java platforms. On Mac OS X, the system built-in MRJToolkit implementation will take precedence over the dummy and respond as expected.

2.3 MRJAppBuilder

Mac OS X applications are packaged in something called a bundle. In reality, a bundle is nothing more than a specific kind of directory structure that contains meta information about as well as the actual resources that make up an application. The Finder treats bundles in a special way, so that users see an application that they can double-click, and not a directory.

MRJAppBuilder[2] is a small utility application that helps you to construct an application bundle from a number of JAR files. In essence,

[1] Macintosh Runtime for Java was the name of Apple Java implementation in the days of Mac OS 9, now long gone

[2] MRJAppBuilder is located in the directory /Developer/Applications.

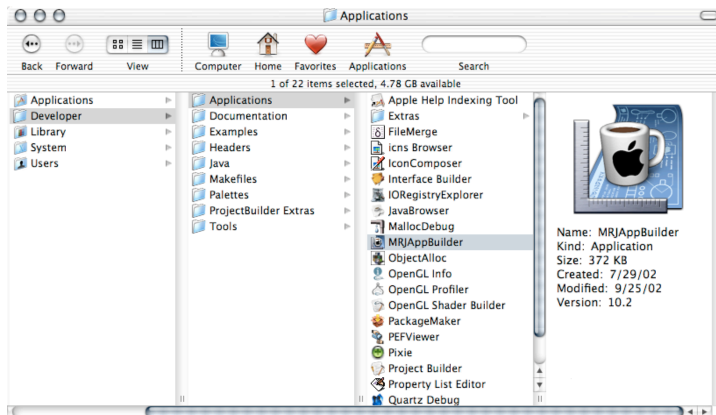

Figure 2: The MRJAppBuilder Tool

it will turn your JAR files into a nicely package Mac OS X application. There are other tools that can create application bundles, like Project Builder, the IDE that Apple includes with Mac OS X. It is even possible to create bundles manually. To maintain your application bundle during development, it is often easier to simply copy files into it, or to edit the meta files by hand. We'll show how to use MRJAppBuilder when we package our first example.

3 Tuning your Java Swing Application

3.1 Handling Basic Apple Events

Any application[3] receives an Application Menu in the Finder. This is the menu next to the system Apple Menu named after the application itself. The layout and functionality of the application menu is predefined. See figure 3 for an example.

Even if you double-click a JAR file, you get such a menu. However, without adding some code, most items in the application menu will do nothing.

The way to make things work, it to implement some basic Apple Event handlers, since that is the way the Finder implements the application menu. Apple Events can also be sent to your application using other means like Apple Script, but that is another story. Using MRJToolkit, you can easily implement handlers for the following events:

- the **About** event sent when the user selects the 'About XYZ...' application menu item

[3]Any application with a GUI that is, as a UNIX system Mac OS X has many more faceless applications operating in the background. A Java application can executed like this too.

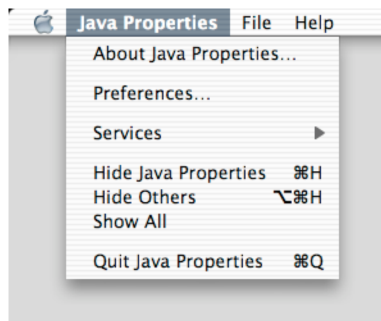

Figure 3: A Mac OS X Application Menu as it is supposed to look

- the **Quit** event sent when the user selects the 'Quit XYZ' application menu item
- the **Preferences** event sent when the users selects the 'Preferences...' application menu item
- the **Open Application** event sent when your application launches
- the **Open Document** event sent when your application should open a document (see the Finder Drag and Drop subsection)
- the **Print Document** event sent when your application should print a document

For our first example, the Java Properties application, we will only implement the About, Quit and Preferences handlers. In Java code, you do this by implementing an interface (and accompanying method) for each event and registering the event handler with the system. Step one is to declare that one of your classes (probably the main one) implements some handlers. Notice how we import the `com.apple.mrj` package.

```
import javax.swing.*;
import com.apple.mrj.*;

/** Example application to look at Java Properties */
public class JavaProperties
extends JFrame
       implements MRJAboutHandler,
                  MRJQuitHandler,
                  MRJPrefsHandler {

    //...

}
```

In most cases, you will already have methods that do the right thing in some standard Java way, so implementing the handlers is easy: just delegate.

```
public void handleQuit() {
    quit();
}

public void handlePrefs() {
    preferences();
}

public void handleAbout() {
    about();
}
```

The final step is to register the handlers. This is most easily done in your main class' constructor.

```
private static boolean macOS =
    System.getProperty("mrj.version") != null;

public JavaProperties() {
    super("Java Properties");
    //...
    MRJApplicationUtils.registerAboutHandler(this);
    MRJApplicationUtils.registerQuitHandler(this);
    if (macOS)
      MRJApplicationUtils.registerPrefsHandler(this);
    //...
}
```

Due to some quirk in the dummy MRJToolkit library, we cannot call the `registerPrefsHandler` on non Mac OS X platforms. Asking for the `mrj.version` system property is the official way to find out if we are on Mac OS X.

The simple additions we made in this section have virtually zero impact on your application's code base, yet improve Finder integration a lot. Even a double-clicked JAR file now has a functional application menu.

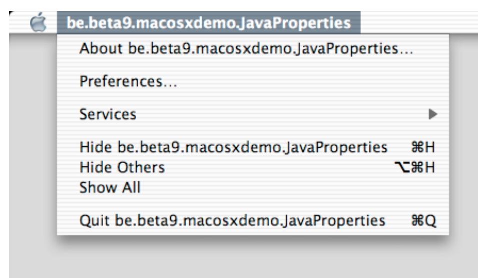

Figure 4: A Mac OS X Application Menu from a JAR file

3.2 Packaging your Application

Double-clicking a JAR file works just fine, thank you. But this is not the way a normal Mac OS X application is presented to the user. What we need to do is package the application's JAR files into a proper application bundle. Another anomaly with simple JAR file execution is that our application is named - in its application menu and in the dock - by its main class name, like `be.beta9.macosxdemo.JavaProperties`. You can see this in figure 4. This too can be fixed by packaging. Using MRJAppBuilder packaging is easy.

We start with a JAR file containing our application code (as built by invoking `ant jar` in the top directory of the source distribution): `JavaProperties.jar`. Next invoke MRJAppBuilder (see figure 2) and switch to the 'Merge Files' tab. Click the 'Add...' button and select the `JavaProperties.jar` file. The tab should now look like in figure 5.

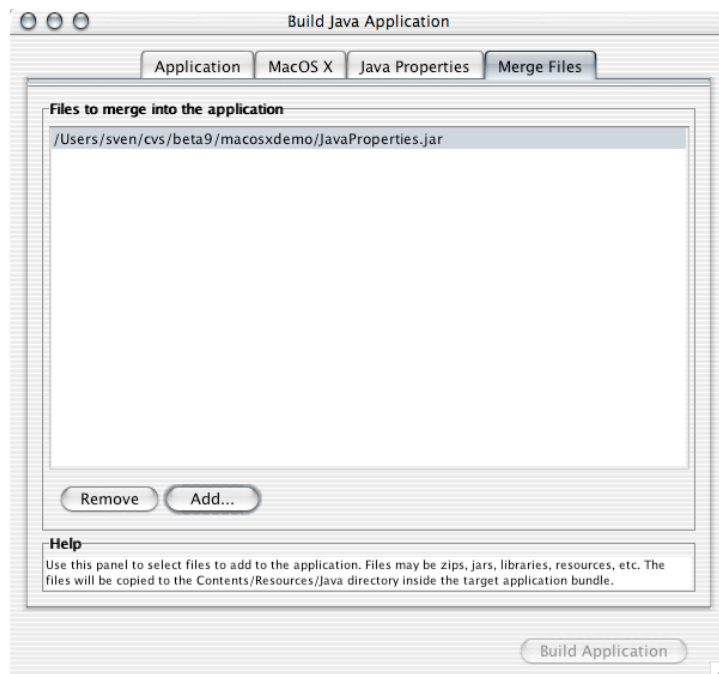

Figure 5: The 'Merge Files' tab of MRJAppBuilder

Then go to the 'Mac OS X' tab and enter the name of the application, 'Java Properties', as the value for the key `CFBundleName`. The tab should now look like in figure 6.

Now go to the 'Application' tab and enter the main class name, `be.beta9.macosxdemo.JavaProperties`. Select an output file by clicking the 'Set...' button, navigating to a location where you want to place the packaged application (your build directory) for example, and typing a name, 'Java Properties'. The tab should now look like in figure 7. Notice how the 'Build Application' button is now enabled.

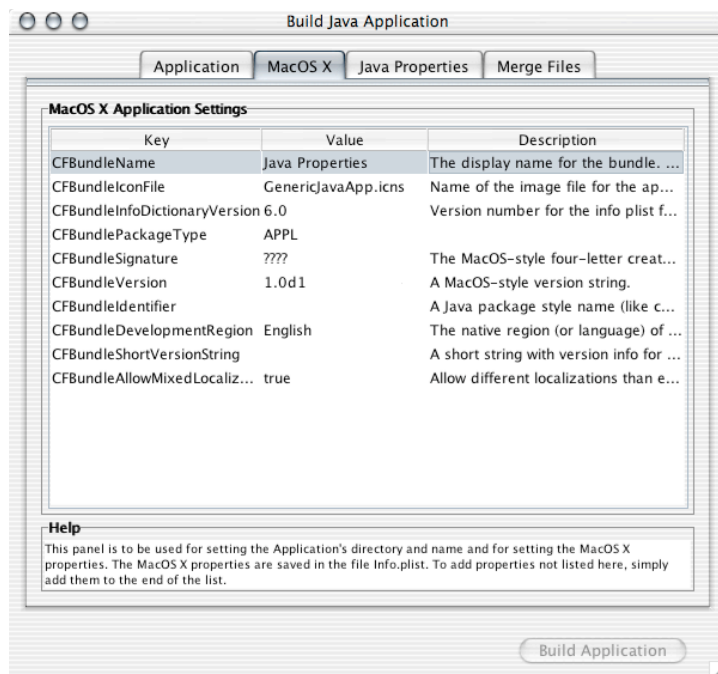

Figure 6: The 'Mac OS X' tab of MRJAppBuilder

We are now ready to build our packaged application bundle. Before clicking the 'Build Application' button, check the 'Java Properties' tab. It should look like figure 8. Note how the properties `main` and `classpath` are automatically set to the correct values. Now build the application. MRJAppBuilder will respond with the message 'Application Built'. Quit MRJAppBuilder and try double-clicking the new application.

3.3 Adding your own Icon

The application MRJAppBuilder constructed has a standard, generic icon by default. Although creating a good Mac OS X icon is hard (using 128 by 128 millions of colors pixels and transparency), adding one to our application bundle is easy. Just manually copy a `.icns` file into your application bundle. For example, assuming the icon that comes with the source distribution in the images directory, `java-properties.icns`, is in your `~/Pictures` directory and assuming you placed the previously built application on your desktop:

```
[sven@voyager:~]$ cd Desktop/
[sven@voyager:~/Desktop]$ cp ~/Pictures/java-properties.icns \
Java\ Properties/Contents/Resources/
```

Then edit the `Info.plist` meta file that's located in the Contents directory of your application bundle (thus in `Java\ Properties/Contents/`).

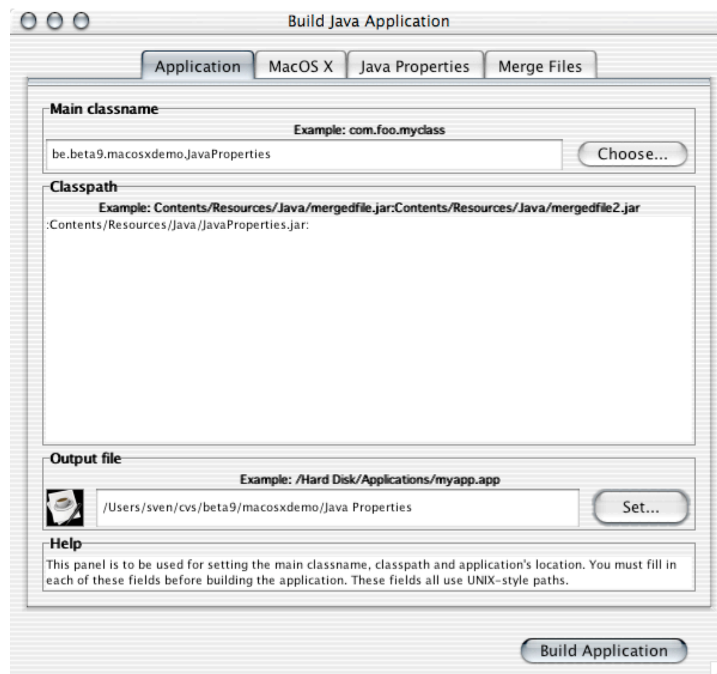

Figure 7: The ‘Application’ tab of MRJAppBuilder

This is an XML file, edit it so that its contents `<dict>` tag looks like:

```
<dict>
        <key>CFBundleName</key>
        <string>Java Properties</string>
        <key>CFBundleVersion</key>
        <string>1.0d1</string>
        <key>CFBundleAllowMixedLocalizations</key>
        <string>true</string>
        <key>CFBundleExecutable</key>
        <string>JavaApplicationStub</string>
        <key>CFBundleDevelopmentRegion</key>
        <string>English</string>
        <key>CFBundlePackageType</key>
        <string>APPL</string>
        <key>CFBundleSignature</key>
        <string>????</string>
        <key>CFBundleInfoDictionaryVersion</key>
        <string>6.0</string>
        <key>CFBundleIconFile</key>
        <string>java-properties.icns</string>
</dict>
```

We changed only the last key-string pair to name our `java-properties.icns` file instead of `GenericJavaApp.icns`. You may remove `GenericJavaApp.icns`

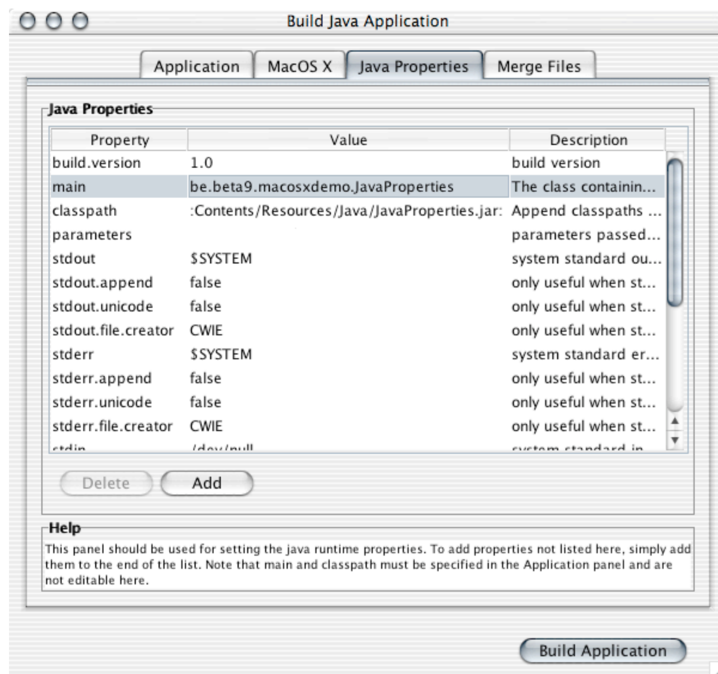

Figure 8: The 'Java Properties' tab of MRJAppBuilder

from the bundle as well. Depending on your interactions with the Finder, things might not get updated right away. The Finder is caching meta information about applications and can get confused when you are changes things under the hood during development. One trick that seems to work for me is to move the modified not-yet-updated application to another volume, renaming it, copying it back and renaming it again.

3.4 File Dialogs

In J2SE there are two different ways to bring up File Dialogs:

- `java.awt.FileDialog` that has been part of Java since 1.0
- `javax.swing.FileChooser` and friends as part of the Swing JFC

Of these two, we would strongly suggest that you use the `java.awt` version, as it will give a much more native look and feel (on any platform). Users will immediately recognize the dialog and feel comfortable using it. The only reason to use `javax.swing.FileChooser` and friends is when you have special needs that are not covered by the more limited `java.awt.FileDialog` class. Use an informative message in your File Dialogs, suggest a file name when saving and set a file filter when loading - make life easy for your users. This is the method implementing the open file functionality in the JPEG Viewer example application:

```
public void open() {
```

```
    FileDialog dialog = new FileDialog(this,
                                       "Open JPEG Document",
                                       FileDialog.LOAD);
    dialog.setFilenameFilter(new FilenameFilter() {
                public boolean accept(File dir, String name) {
                    return name.toLowerCase().endsWith("jpg");
                }
            });
    dialog.setVisible(true);
    if (dialog.getFile() != null) {
        File file = new File(dialog.getDirectory(),
                             dialog.getFile());
        viewJPEG(file);
    }
}
```

Next is part of the method implementing the export functionality in the Java Properties example application:

```
FileDialog dialog = new FileDialog(this,
                                   "Save Java Properties As",
                                   FileDialog.SAVE);
dialog.setVisible(true);
dialog.setFile("java-properties.txt");
if (dialog.getFile() != null) {
    File file = new File(dialog.getDirectory(),
                         dialog.getFile());
    //...
}
```

3.5 About Boxes

For many applications, the splash screen and about box are part of their image. If you feel like it, please use your own designs. However, in the Cocoa frameworks of Mac OS X, about boxes come for free, in a mostly standard layout. In our examples, we tried to emulate that look in the class `be.beta9.framework.swing.AboutBox`. In figure 9 you can see what the result is. The application icon is reused, and the text is composed of the title, a version description and a copyright. Now you got one thing less to worry about ;-)

3.6 Preferences

There is nothing special about preferences in Mac OS X, just the requirement to save them in a particular place. The standard location to save preferences is in a directory called **Preferences** in the users' **Library** folder: thus in `~/Library/Preferences/`.

In the examples, we are using a simple Preferences object (implementated in `be.beta9.macosxdemo.Preferences`) that uses the Java Properties file format. The Mac OS X specific part of this class is in the constructor:

Figure 9: A Cocoa-like About Box

```
public class Preferences {

    private Properties properties;
    private File file;

    public Preferences(String name) {
        this.properties = new Properties();
        if (System.getProperty("mrj.version") != null) {
            try {
                File dir = MRJFileUtils.findFolder(
                             MRJFileUtils.kPreferencesFolderType);
                file = new File(dir, name + ".properties");
            } catch (FileNotFoundException ignore) {
            }
        }
        if (file == null) {
            File dir = new File(System.getProperty("user.home"));
            file = new File(dir, "." + name + ".properties");
        }
        load();
    }

    //...

}
```

Again, we are using the MRJToolkit to find the preferences folder, which will yield `~/Library/Preferences/` on Mac OS X.

Another typical Mac OS X thing is that in many cases, preferences take immediate effect and are saved automatically. This makes your preferences windows simpler and easier to use, while helping your users understand the effects of changes. Check out the example implementation in the source code.

3.7 Customizing the Menu Bar

Even after all the steps in the previous subsections, we still have an application that looks slightly off (see figure 10). Mac OS X users will immediately notice the missing menu bar on the top of the screen and will be puzzled by a menu bar inside the main window.

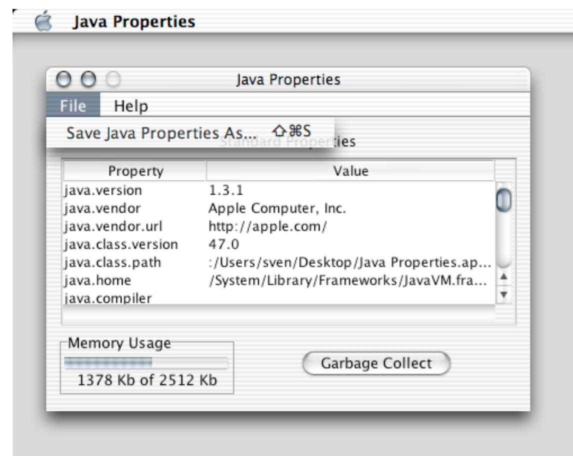

Figure 10: The Java Properties application, with odd menu bar

Moving the menu bar to the proper location is easy: open the meta file `MRJApp.properties` inside your application's `Contents/Resources` directory and add the following line:

```
com.apple.macos.useScreenMenuBar=true
```

This will place the JMenuBar of a JFrame on top of the screen. However this fixes only the placement problem. For example, in a typical Windows menu bar, the first menu, File, will contain a Quit item. Since a Mac OS X application has a Quit item in its Application menu, we have a problem. The solution is to build the menu bar with variations for specific platforms. A straightforward approach is to do it like this:

```
public JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();
    int mask = Toolkit.getDefaultToolkit().getMenuShortcutKeyMask();
    JMenu fileMenu = new JMenu("File");
    JMenuItem saveMenuItem = new JMenuItem("Save Java Properties As...");
    saveMenuItem.setAccelerator(
       KeyStroke.getKeyStroke(
          KeyEvent.VK_S, mask + KeyEvent.SHIFT_MASK));
    saveMenuItem.addActionListener(
       new GenericActionListener(this, "export"));
    fileMenu.add(saveMenuItem);
    if (!macOS) {
        fileMenu.addSeparator();
```

```
        JMenuItem prefsMenuItem = new JMenuItem("Preferences...");
        prefsMenuItem.addActionListener(
           new GenericActionListener(this, "preferences"));
        fileMenu.add(prefsMenuItem);
        fileMenu.addSeparator();
        JMenuItem quitMenuItem = new JMenuItem("Quit");
        quitMenuItem.addActionListener(
           new GenericActionListener(this, "quit"));
        fileMenu.add(quitMenuItem);
    }
    menuBar.add(fileMenu);
    //...
    return menuBar;
}
```

If we are not on Mac OS X, we add a Preferences and Quit item to the File menu. Notice how we are using the method getMenuShortcutKeyMask to find the platform specific mask for menu bar keyboard shortcuts.

The next problem is that in Swing there is no concept of a main menu bar. Instead, each JFrame can have a JMenubar associated with it. If you bring up multiple frames with different (or absent) menu bars, then the menu bar displayed on the top of the screen will change depending on which frame is in focus. This will seriously confuse your Mac OS X users. Fixing this elegantly is hard, a workaround is to duplicate[4] your main menu bar and add it to each frame[5]. Bringing up our preferences panel looks like this:

```
public void preferences() {
    JFrame frame = new JavaPropertiesPreferences(this);
    frame.pack();
    SwingTools.centerComponent(frame, null);
    if (macOS) frame.setJMenuBar(createMenuBar());
    frame.setVisible(true);
}
```

In the subsection 'Basic Multi-Document Support' we will engage in some further menu bar hacking to bridge the conceptual gaps between Swing and Mac OS X.

3.8 Supporting Finder Drag and Drop

Our second example has some characteristics of a multi-document application. JPEG Viewer does what its name says: view JPEG documents. See figure 11 for a screen shot of JPEG Viewer in action.

[4]Each menu bar has to be an independent copy, sharing the same menu bar doesn't seem to work.

[5]Since in Swing only frames and not dialogs can have a menu bar, the problem of the disappearing menu bar persists and remains unfixable for dialogs.

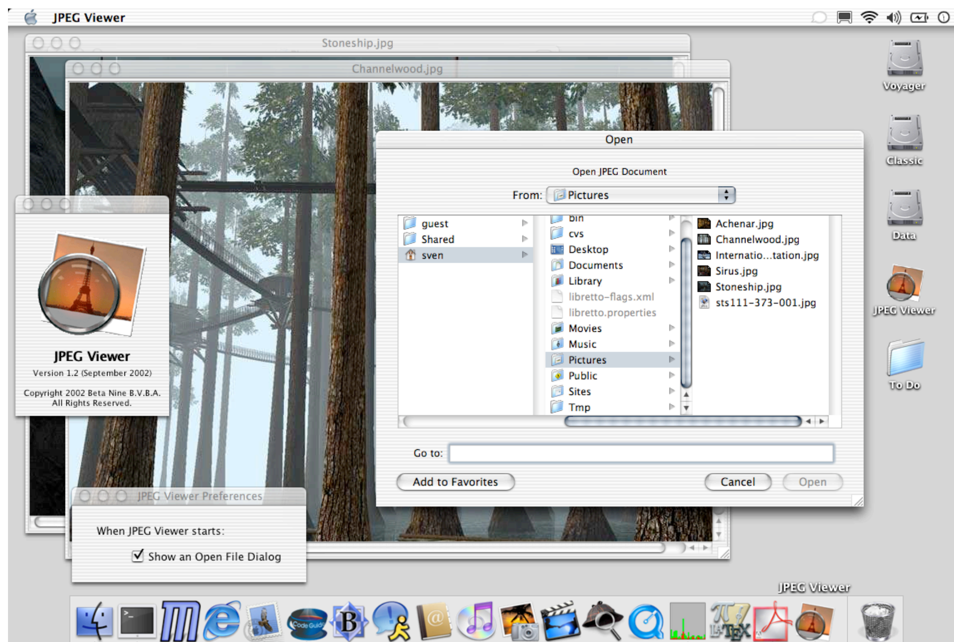

Figure 11: The JPEG Viewer application in action

One of the things that we want from an application handling a specific document type is good integration with the Finder. We want to be able to:

- drag and drop a JPEG document onto our application icon in the Finder and have it highlight when it accepts the drop and open the document
- drag and drop a JPEG document onto our application icon in the Dock and have it highlight when it accepts the drop and open the document
- ask the Finder to open JPEG document by using the Context Menu or even select our JPEG Viewer as default JPEG handler

Again the basic interface between the Finder and our application is through Apple Events, in this case the 'Open Document' event. The steps to make this work are familiar: implement the interface, register the handler and implement the handler.

```
public class JPEGViewer extends JFrame
    implements
    MRJAboutHandler, MRJQuitHandler, MRJPrefsHandler,
    MRJOpenDocumentHandler {

    //...

    public JPEGViewer() {
```

```
        super("JPEG Viewer");
        MRJApplicationUtils.registerOpenDocumentHandler(this);
        //...
    }

    public void handleOpenFile(final File file) {
        if (!file.exists()) return;
        if (!file.getName().toLowerCase().endsWith("jpg")) return;
        allowOpenDialog = false;
        SwingUtilities.invokeLater(new Runnable() {
                    public void run() {
                        viewJPEG(file);
                    }
                });
    }

    //...
}
```

Again we assume the application already knew how to load and view a JPEG file, so we just have to delegate. It is always wise to place Swing GUI interaction on the main Swing event thread, since we cannot be sure on which thread the Apple Event is delivered. The application itself is now ready to handle the 'Open Document' event, we will however not receive it from the Finder since the Finder doesn't know that we can handle it. In other words we need to adjust our meta information again. Edit the `Info.plist` file of the JPEG Viewer application bundle and add a `CFBundleDocumentTypes` key with an `array` value as an element to the `<dict>` tag:

```
<plist version="0.9">
<dict>
    <key>CFBundleName</key>
    <string>JPEG Viewer</string>
    <!-- ... -->
    <key>CFBundleIconFile</key>
    <string>jpegviewer.icns</string>
    <key>CFBundleDocumentTypes</key>
    <array>
        <dict>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>jpg</string>
            </array>
            <key>CFBundleTypeMIMETypes</key>
            <array>
                <string>image/jpeg</string>
            </array>
            <key>CFBundleTypeName</key>
            <string>JPEG Image</string>
            <key>CFBundleTypeOSTypes</key>
```

```
            <array>
                <string>JPEG</string>
            </array>
            <key>CFBundleTypeRole</key>
            <string>Viewer</string>
        </dict>
    </array>
</dict>
</plist>
```

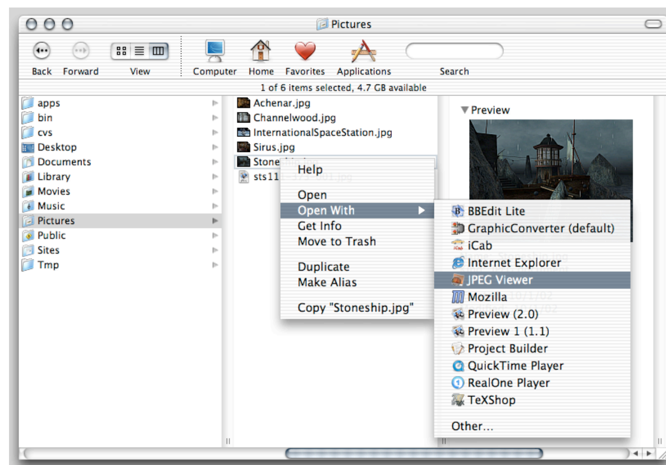

Figure 12: The Finder's Context Menu for JPEG files extended!

We actually copied the needed XML from the `Info.plist` of Mac OS X's built-in Preview application and then edited it a bit. Once installed like this, the Finder will be happy to drag and drop JPEG files onto our JPEG Viewer. Even multiple JPEG files are handled well, just try it. And the Finder's Context Menu for JPEG documents now includes our application as can be seen in figure 12.

3.9 Standard Help

Mac OS X's help system is HTML based and presented through an application called Help Viewer. Most Java Swing applications will already use HTML for help, so integration with Mac OS X's help system isn't too hard. A first step in that direction is this code excerpt from the `help` method:

```
if (macOS) {
   try {
       File helpFile = new File("/tmp/JPEGViewer.help");
       InputStream in = getClass().getResourceAsStream(
                            "/JPEGViewer.help");
       Reader reader = new InputStreamReader(in);
```

```
        FileWriter writer = new FileWriter(helpFile);
        int c;
        while ((c = reader.read()) != -1) writer.write(c);
        reader.close();
        writer.close();
        Runtime.getRuntime().exec(new String[] {
                    "open", helpFile.getAbsolutePath() });
    } catch (IOException ignore) {
    }
}
```

What we do here is retrieve an HTML file as a Java resource, copy it to some temporary location and ask it to be open using the Mac OS X command line utility `open`. Because of the `.help` extension, Help Viewer will be started automatically. On other platforms, we use a simple HTMLViewer helper class that makes use of the fact that Swing's JEditorPane can be configured to display HTML. Granted this is only a first step, extensions beyond this are left as an exercise and will require a good reading of Apple's documentation.

3.10 Basic Multi-Document Support

Complete Mac OS X style multi-document support is simply hard[6]. As noted before, Swing's concept of JMenuBars fixed to individual JFrames makes things even harder to implement. A first problem is that it must be possible to have a multi-document application open and active, with a functional menu bar, even when no windows are open. Our workaround for this problem was to create a dummy frame with a menu bar and move it off the screen (using negative coordinates in `setLocation`).

Another problem is that all frames are given the same menu bar, even the about box and preferences panel, but regardless of which frame has the actual focus, we want certain actions to be performed on the current document. In our example, the `close` method needs a reference to the current document, even it is invoked from the menu bar installed in the dummy frame or in the about box or preferences panel. Our solution to this problem was to track the current or last JPEG frame that was active using a stack-like datastructure. We update this stack by listening to some window events in the window listener of our JPEG frame and in the close method itself. Please study the source code of JPEG Viewer to better understand our implementation.

The stack could also be used to produce a menu listing all open documents, ordered according to their on screen layering. This is left as an exercise for the reader, but be aware that there currently is an issue with updating menus in a menu bar. It might be necessary to add and remove menus or submenus to get them updated.

[6]In the Cocoa Frameworks, this comes essentially for free. This is a huge time saver, but it remains a complex subject area once you go beyond the basics.

4 Conclusion

We showed how you can fine-tune your standard Java Swing application for the Mac OS X User Experience, with little effort and without compromising portability.

References

[1] Apple Computer. *Technical Note TN2042: Tailoring Java Applications for Mac OS X* May 21, 2002. Available online: http://developer.apple.com/technotes/tn/tn2042.html

[2] Apple,Computer. *Technical Note TN2031: Java Runtime Properties for Mac OS X* May 32, 2001. Available online: http://developer.apple.com/ja/technotes/tn2031.html

[3] Apple,Computer. *Switch to Mac OS X: A guide to key user experience differences between Microsoft Windows and Mac OS X* Available online: http://developer.apple.com/ue/switch/windows.html

[4] Apple,Computer. *Mac OS X User Experience* Website: http://developer.apple.com/ue/index.html