



JavaOS™ for Business™



**JavaOS™ for Business™ Version 2.0
Localization Guide**

June 1998

Copyright 1998 Sun Microsystems, Inc., 10201 N. DeAnza Blvd • Cupertino, California 94303 U.S.A.; IBM Corporation, Old Orchard Road, Armonk, New York 10504. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun Logo, Java, Hot Java Browser, JavaOS and JavaOS for Business are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries, and are used under license by IBM. The JavaOS For Business technology is the result of a collaboration of Sun and IBM. IBM and the IBM Logo are registered trademarks of IBM Corp. in the United States and other countries, and are used by Sun Microsystems under license.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

All other product names mentioned herein are the trademarks of their respective owners.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements. U.S. Government approval required when exporting the product.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Govt is subject to restrictions of FAR 52.227-14(g) (2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015 (b)(6/95) and DFAR 227.7202-3(a).

Copyright 1998 Sun Microsystems, Inc.; IBM Corporation. Tous droits réservés. Distribué par des licences qui en restreignent l'utilisation. Sun, Sun Microsystems, le logo Sun, Java, JavaOS et JavaOS for Business sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays et elles sont utilisées sous licence par IBM. La technologie JavaOS for Business est le résultat d'une collaboration entre Sun et IBM. IBM et le logo IBM sont des marques déposées d'IBM Corporation aux Etat-Unis et dans d'autres pays et elles sont utilisées sous licence par Sun Microsystems. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la dé compilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, Java, JavaOS et JavaOS for Business sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays et elles sont utilisées sous licence par IBM. La technologie JavaOS for Business est le résultat d'une collaboration entre Sun et IBM. IBM et le logo IBM sont des marques déposées d'IBM Corporation aux Etat-Unis et dans d'autres pays et elles sont utilisées sous licence par Sun Microsystems.

L'interface d'utilisation graphique OPEN LOOK et Sun(TM) a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences é crites de Sun. L'accord du gouvernement américain est requis avant l'exportation du produit.

THIS PUBLICATION IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Contents

Preface i

1. Localization Files 3

HTML Files 4

Keyboard Tables 4

Resource Bundles 5

Configuring JavaOS With JCT 7

2. Using the Localization Build System 9

Configuring the Localization Build System 9

Adding Files to the Localization Build System 10

Building Localization Class Files 11

Running JavaOS With Localized Files 11

3. Managing a Font Server 13

How JavaOS Locates Fonts 13

font.properties File 14

FONTS.LST File 14

Adding TrueDoc (PFR) Fonts 15

Adding TrueType Fonts 15

Configuring a Font Server 16

4. Managing an Input Method Server	17
5. JavaOS Control Applets	19
Edit->Preferences->Audio Video...	19
AudioControlApplet	19
ScreenBlankApplet	19
FrameBufferResizerApplet	20
Edit->Preferences->Select Printers...	20
PrinterChooserApplet	20
View->Monitor->Memory	21
HeapMapApplet	21
View->Monitor->Printer Queue	21
PrinterQueueApplet	21
Help->About JavaOS	22
SysInfoApplet	22
6. Adding New Keyboard Layouts	23
Overview	23
Required Keyboard Data	24
Adding a New Keyboard	25
7. Adding Compose Files	27
Overview	27
Architecture	27
Adding a New Compose File to a Locale	28
Setting Up and Testing the Program	31

Tables

TABLE 1-1	HTML Files for Localization	4
TABLE 1-2	Control Applets in <code>src/javaos/sun/sun/javaos/applets</code>	5
TABLE 1-3	Application Launcher in <code>src/javaos/sun/sun/javaos/application</code>	5
TABLE 1-4	Spooling Classes in <code>src/javaos/sun/sun/javaos</code>	6
TABLE 1-5	Login Dialog Classes in <code>src/share/sun/sun/javaos</code>	6
TABLE 1-6	Printing Subsystem Classes in <code>src/share/sun/sun/printing</code>	6
TABLE 1-7	Logon Framework Classes in <code>src/javaos/ibm/ibm/javaos/logonfw</code>	6
TABLE 1-8	Copyright Classes in <code>src/javaos/ibm/ibm/javaos</code>	6
TABLE 2-1	build Script Options	11
TABLE 3-1	JavaOS Font Formats	13

Preface

All JavaOS localization data is stored in class files separate from the main JavaOS binary image. Using a separate build system for localization files has two major benefits:

- It reduces the size of the JavaOS binary by storing the class files on a server and then loading them when necessary.
- It prevents localizers from needing to modify the JavaOS source code directly.

This guide describes how to localize the JavaOS.

How This Book Is Organized

Chapter 1 describes translating Unicode-based localization files for specific locales.

Chapter 2 discusses how to use the build system to generate class files from the localization files.

Chapter 3 discusses managing a font server.

Chapter 4 discusses managing an input method server.

Chapter 5 discusses using JavaOS control applets for controlling system resources and preferences.

Chapter 6 discusses adding new keyboard layouts to JavaOS.

Chapter 7 describes adding compose files for European locales.

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name% You have mail.</code>
AaBbCc123	What you type, contrasted with on-screen computer output	<code>machine_name% su</code> <code>Password:</code>
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type <code>rm filename</code> .
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P-2 Shell Prompts

Shell	Prompt
C shell prompt	<code>machine_name%</code>
C shell superuser prompt	<code>machine_name#</code>
Bourne shell and Korn shell prompt	<code>\$</code>
Bourne shell and Korn shell superuser prompt	<code>#</code>

Localization Files

The JavaOS localization build system is separate from (but parallel to) the main JavaOS build system. It contains only those files that are part of the localization process: resource bundles, locales, formatters, keyboard tables, character converters, HTML files and properties files. The makefiles and build script have been adapted from the master JavaOS build system.

The localization data for locales, formatters and character converters is inherited from JDK 1.1.x. This leaves resource bundles, keyboard tables, HTML files and properties files to be translated by JavaOS localization centers.

Resource bundles contain string information used by applets in the JavaOS user interface. *Keyboard tables* describe keyboards that are used in specific locales. *HTML files* are wrappers for JavaOS control applets that are used with a Web browser such as HotJava. *Properties files* contain string information used by the JCT beans on the server side.

HTML Files

The HTML files are located in `src/javaos/lib/html` in the JavaOS localization build system and described in the table below.

TABLE 1-1 HTML Files for Localization

HTML File	Description
<code>aboutJavaos.html</code>	General information about JavaOS
<code>heapMap.html</code>	Memory usage
<code>preferences-printers.html</code>	Printer selection
<code>preferences-av.html</code>	Audio and video preference selection
<code>printerQueue.html</code>	Print job control
<code>sysinfo.html</code>	System information about JavaOS

Keyboard Tables

Keyboard tables describe keyboard layouts for specific locales. They are stored on a server and loaded at start-up. Keyboard table files are located in `src/javaos/sun/sun/javaos`.

Keyboard table files are built by using the `sun.javaos.KeyTable` subclass. The best approach for doing this is to start with `US103KeyTable.java` for an ISO Latin-1-based locale or `JP194KeyTable.java` for an Asian locale. The JCT bean `keytablecfg` allows the administrator to select the keyboard used by the client.

For information about adding keyboard tables to JavaOS, see Chapter 6.

Resource Bundles

The rest of the JavaOS localization files are resource bundle files located in `src/javaos/sun/sun/javaos` or `src/share/sun/sun/printing`. Each resource bundle file corresponds to one or more source files in the JavaOS source tree. The resource bundle files are stored in a separate localization source tree in a sub-directory named `resources` that corresponds to the JavaOS directory. Resource localization files contain the string `Resource` in their file name.

For example, the JavaOS source file for the screen blank applet is located in `src/javaos/sun/sun/javaos/applets` in the JavaOS source tree. The directory `src/javaos/sun/sun/javaos/applets/resources` in the localization source tree contains resource bundle files.

The following tables identify each resource bundle file and its corresponding JavaOS source file. The localization resource files are ASCII-based Java files with a straightforward format.

TABLE 1-2 Control Applets in `src/javaos/sun/sun/javaos/applets`

Resource Bundle	JavaOS Source File
<code>AudioControlResource.java</code>	<code>AudioControlApplet.java</code>
<code>FrameBufferResizerResource.java</code>	<code>FrameBufferResizerApplet.java</code>
<code>HeapMapResource.java</code>	<code>HeapMapApplet.java</code>
<code>PrinterChooserResource.java</code>	<code>PrinterChooserApplet.java</code>
<code>PrinterQueueResource.java</code>	<code>PrinterQueueApplet.java</code>
<code>ScreenBlankResource.java</code>	<code>ScreenBlankApplet.java</code>
<code>SysInfoResource.java</code>	<code>SysInfoApplet.java</code>

TABLE 1-3 Application Launcher in `src/javaos/sun/sun/javaos/application`

Resource Bundle	JavaOS Source File
<code>LauncherResource.java</code>	<code>Launcher.java</code>
<code>ProgressDialogResource.java</code>	<code>ProgressDialog.java</code>

TABLE 1-4 Spooling Classes in `src/javaos/sun/sun/javaos`

Resource Bundle	JavaOS Source File
<code>ContentTransportResource.java</code>	<code>LocalContentTransport.java</code> <code>lpdContentTransport.java</code>
<code>JavaOSUserConsoleResource.java</code>	<code>JavaOSUserConsole.java</code>

TABLE 1-5 Login Dialog Classes in `src/share/sun/sun/javaos`

Resource Bundle	JavaOS Source File
<code>LoginDialogResource.java</code>	<code>LoginDialog.java</code>

TABLE 1-6 Printing Subsystem Classes in `src/share/sun/sun/printing`

Resource Bundle	JavaOS Source File
<code>MessageResource.java</code>	<code>DefaultAttributeGUI.java</code> <code>LocalSpooler.java</code> <code>PCL5ContentProducer.java</code> <code>PlatformPrintControl.java</code> <code>PlatformPrintDialog.java</code> <code>PostScriptContentProducer.java</code> <code>PrintControl.java</code> <code>PullAdapter.java</code>

TABLE 1-7 Logon Framework Classes in `src/javaos/ibm/ibm/javaos/logonfw`

Resource Bundle	JavaOS Source File
<code>LogonFWResources.java</code>	<code>LogonManager.java</code>
<code>StdRemoteAuthenticatorResources.java</code>	<code>NISPanel.java</code>

TABLE 1-8 Copyright Classes in `src/javaos/ibm/ibm/javaos`

Resource Bundle	JavaOS Source File
<code>CopyrightResources.java</code>	<code>JavaOSLogoPanel.java</code>

Configuring JavaOS With JCT

There are several JavaBeans that have been written to configure JavaOS using JCT. Each JavaBean can have translations completed for them. The beans are located in `src/server/com/ibm/joscfg/xxxxx/mri/*.properties` where `xxxxx` is the name of the configuration bean.

For example, the properties file for the AppLoad bean is located in `src/server/com/ibm/joscfg/appload/mri/AppLoad.properties`.

All translated properties files are located in the `mri` subdirectories on the server source tree.

All translated HTML files are located in the `html/xx` subdirectory on the server source tree, where `xx` is the locale for the translated versions of the HTML files.

Using the Localization Build System

This chapter describes the two main tasks that a developer needs to complete on the JavaOS localization build system: adding localized files and building the localization class files.

Configuring the Localization Build System

Before modifying or using the JavaOS localization system, the `build` script (located in `build/javaos`) must be modified to take into account the local environment.

- 1. At the top of the `build` script, change the `jdk` variable to indicate the location of local copy of JDK 1.1.x.**
- 2. Make sure `gnumake(1)` is in the shell path.**
- 3. Set the permissions of the `build` script to make it executable.**

Adding Files to the Localization Build System

The following steps describe how to add a localized file to the localization build system:

- 1. Add the localized file to the appropriate source directory in the localization hierarchy.**

- a. Place Java files in the directory that contains the default files.**

An underscore locale abbreviation is appended to the file name before the `.java` suffix. For example, the following files would be placed in the directory `src/javaos/sun/sun/javaos/applets/resources`:

`AudioControlResource.java` (default file)

`AudioControlResource_ja.java` (Japanese locale)

`AudioControlResource_fr_CH.java` (Swiss-French locale)

- b. Place HTML files in a directory that is parallel to the default files.**

A language abbreviation is prepended as the first element of their path. For example, the following HTML files would be placed in the directory `src/javaos` hierarchy:

`lib/html/about.html` (default file)

`lib/ja/html/aboutJavaOS.html` (Japanese version)

- 2. Modify the `build/javaos/java/java/Classfiles-LC.gmk` file.**

`Classfiles-LC.gmk` contains filenames organized into sections according to functionality. The standard ten languages are split into their own groups with any others currently grouped together under `other`.

The first category is `default`, which contains the default resource bundle files. These are the files that need to be translated. Once a translation is made, enter the new file with the correct underscore suffix in the section for that file's language. This organization allows for the separate compilation of different languages.

If new translations are added, update `Classfiles-LC.gmk` accordingly.

Building Localization Class Files

There are three classes of options for the localization build system: `build` script options, `gnumakefile` options, and `javac` command line options. Compilation options allow determination of the target languages and whether converters and keyboard tables will be compiled.

Usage: `build [compilation options] [gnumake options] [javac options]`

TABLE 2-1 `build` Script Options

Option	Description
<code>LOCALE=true</code>	Makes locales.
<code>CODEPAGE=true</code>	Makes character converters.
<code>KOREANLWE=true</code>	Makes Korean LWE IME.
<code>PRCPYLWE=true</code>	Makes PRC LWE IME.
<code>TWPHLWE=true</code>	Makes Taiwan LWE IME.
<code>KEYBOARD=true</code>	Makes keyboard tables.
<code>LANGUAGE=true</code>	Makes resource bundles for the languages.

By setting any or all of the above options, its functionality will be built. All of the classes that are built will be placed in the `/REMOTE` subdirectory on the server.

Running JavaOS With Localized Files

To run JavaOS with localized files, the files are placed on an NFS server. They are then loaded as needed.

`/REMOTE` is the file system where JavaOS searches for localized files, thus `/REMOTE` is the name to which the localized files should be mapped.

For additional information, see the `/REMOTE` section in *JavaOS for Business Network Operations*.

The method for reading localized HTML files is application-specific. The HotJava browser allows localized HTML files to be placed on an `http` server that is specified by the `doc.url dhcp` option. Here is an example entry:

```
-Ddoc.url=http://wombat.eng/JavaOS/localized/html
```

Set up the files by copying all the output from the build in the `lib` directory down into `/JavaOS/localized/html/lib/`.

The login screen contains a configurable list of locales. The contents of this list are specified at boot time as a `dhcp` option. JavaOS uses the following list of priorities to select the language the locales are displayed:

- The default locale (the new locale selected)
- The locale's language
- The locale's ISO codes

If the option is not specified, the list contains only English (United States). The option is a list of locales (separated by semicolons) specified using the ISO language and country codes with an underscore between them.

To work correctly, locales specified on the option line should have a matching `LocaleElements_*.class` file on the server for the locale's language or language/country. Without this file a locale may appear as the default (English (United States)).

The language codes are lowercase two-letter codes as defined by ISO-639. You can find a full list of these codes at a number of sites, such as:

```
http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt
```

The country codes are uppercase two-letter codes as defined by ISO-3166. You can find a full list of these codes at a number of sites, such as:

```
http://www.chemie.fu-berlin.de/diverse/doc/ISO\_3166.html
```

Here are some examples:

```
English, United States: en_US
```

```
French, France: fr_FR
```

```
French, Swiss: fr_CH
```

Managing a Font Server

JavaOS supports three font formats that can be built into the JavaOS binary. These font formats are described in the table below.

TABLE 3-1 JavaOS Font Formats

Format	Font List	Description
Bitmap	<code>src/share/sun/aw/awFontBitmaps.c</code>	For JavaOS 1.0 backward compatibility only.
TrueDoc	Dutch, a serif font similar to Times Roman Swiss, a sans-serif font similar to Helvetica Courier, a monospaced font	Bitstream format.
TrueType	The exact list of TrueType fonts depends on the terms of the JavaOS source license.	Can be stored in ROM or on a font server.

How JavaOS Locates Fonts

JavaOS looks for a font property file at boot time or whenever the locale is set or changed. There are two directories where the font property file can be found: the directory specified by the property `javaos.font.properties.home` (which defaults to `/FONTS`) and the directory specified by the property `java.home` (which has a default value of `/ROM`). The font property file is expected to be in the subdirectory `lib`. For example, if the default directory properties are not overridden, JavaOS will search in `/FONTS/lib/` and then in `/ROM/lib/` for a font property file.

The name of the font property file is the first match in the following list of file names.

```
<dir>/lib/font.properties.language_country_variant  
<dir>/lib/font.properties.language_country  
<dir>/lib/font.properties.language_variant  
<dir>/lib/font.properties.language  
<dir>/lib/font.properties
```

Both `/FONTS` and `/ROM` are aliases for pseudo file systems. `/FONTS` is usually mapped to some NFS location using the `javaos.mountlist` property to specify a font server for loading TrueType fonts. See page 16 for information on how to configure a font server.

`/ROM` is the internal ROM-based file system that is part of the JavaOS image. In the directory `src/javaos/lib` of the JavaOS source code, there is the font property file that ends up in `/ROM/lib`. It contains descriptions of the built-in Latin PFR fonts.

Each font directory requires a font property file and a `FONTS.LST` file.

font.properties File

The default `font.properties` file used in the JavaOS build is in `src/javaos/lib/font.properties`. The format of `font.properties` is described in:

<http://java.sun.com/products/jdk/1.1/docs/guide/intl/fontprop.html>.

FONTS.LST File

The `FONTS.LST` file maps font names to file names. `FONTS.LST` is a list of entries, 1 per line, containing

```
FontName style type FileName
```

`style` is one of `PLAIN`, `BOLD`, `ITALIC`, or `BOLDITALIC` and `type` is, for practical purposes, `truetype`.

The following sample entries set up a font named `times` that has outlines for all four styles:

```
times PLAIN          truetype times.ttf
times BOLD           truetype timesbd.ttf
times ITALIC         truetype timesi.ttf
times BOLDITALIC     truetype timesbi.ttf
```

The following sample entries set up a font named `arial` that has only `PLAIN` and `BOLD` variants; the slanted variant will be generated by the font scaler.

```
arial PLAIN          truetype arial.ttf
arial BOLD           truetype arialbd.ttf
```

Adding TrueDoc (PFR) Fonts

PFR format fonts can be added to the JavaOS binary by putting the font files in the `src/javaos/lib/fonts` directory and then updating the `src/javaos/lib/font.properties` file.

The PFR fonts must use Unicode encoding.

Adding TrueType Fonts

TrueType fonts may be stored in the JavaOS binary or on a font server.

To put TrueType fonts into the image, they need to be added to the ROM file system (for example, in `/ROM/lib/fonts`). Then modify `src/share/sun/type/jtm/jtm.c`.

The only requirement for the TrueType fonts that the scaler imposes is: all TrueType fonts should include a `<3,1> 'cmap'` table, which means that all fonts (except Symbol) should contain a Microsoft Unicode character to glyph encoding table.

Configuring a Font Server

A JavaOS client can load TrueType fonts from a directory on an NFS server. The JavaOS property `javaos.mountlist` indicates which NFS directory JavaOS uses to load fonts.

JavaOS mounts the directory locally as `/FONTS`. This directory must have `lib/font.properties.*` files and a `FONTS.LST` file, which maps font names to file names.

To add TrueType fonts to a font server, you need only to copy the fonts to the font directory and update the `font.properties` and `FONT.LST` files.

See Chapter 2, “Using the Localization Build System” and the companion document *JavaOS for Business Specification* for more information on building fonts into the JavaOS binary.

Managing an Input Method Server

JavaOS uses the Internet/Intranet Input Method Protocol (IIIMP) to provide distributed input method support for JavaOS clients. The JavaOS property `javaos.im.url` indicates the location of the input method server. See the reference documentation for IIIMP Agent/Bridge for network administration procedures for supporting an input method server.

JavaOS Control Applets

JavaOS includes a few applets for controlling system resources and preferences. The applets are available through different menus in HotJava Browser, which uses HTML pages to organize the layout of the different applets. Licensees who use an alternate main application can organize the JavaOS control applets differently. TABLE 1-2 on page 5 describes where the source code and resource bundles for the JavaOS control applets are kept.

The following sections are organized by the location of the Applets in the HotJava Browser user-interface.

Edit->Preferences->Audio Video...

There are three applets associated with this menu item. The HTML page for these applets is located in `/lib/html/preferences-av.html`.

AudioControlApplet

The Volume slider controls the volume of the audio player. The Set button confirms a change.

ScreenBlankApplet

A single check box that controls automatic screen blanking.

FrameBufferResizerApplet

Controls the current video mode for the frame buffer. A video mode includes three attributes:

- Scan dimensions (for example, 640 x 480)
- Scan frequency (for example, 60Hz)
- Bit depth (for example, 16-bit)

These attributes are combined into a single list. The Set button confirms a selection.

Edit->Preferences->Select Printers...

The HTML page for the PrinterChooserApplet is located in `/lib/html/preferences-printers.html`.

PrinterChooserApplet

The `PrinterChooserApplet` manages the user's list of favorite printers from a list of printers available on the network. The applet displays two lists, the list of printers available on a given server and the list of selected applets. The server pop-up menu indicates a list of servers that are available. Use the Add button to add selected printers from the available list to the selected list.

The Apply button adds the selected printers to the list of printers available from the File->Print dialog. The Reset button reverts the available and selected lists to their initial states.

View->Monitor->Memory

The HTML page for `HeapMapApplet` is located in `/lib/html/heapMap.html`.

HeapMapApplet

This is a diagnostic applet that displays a graphical representation of the VM's heap. It also shows the amount of free space left in the heap.

- refresh - redraws the heap map
- perform gc - forces garbage collection
- up & down arrow - navigates current position in the heap
- zoom in & zoom out - increases or decreases the level of detail in map
- K free - indicates free memory
- K total - indicates total memory

View->Monitor->Printer Queue

The HTML page for `PrinterQueueApplet` is located in `/lib/html/printerQueue.html`.

PrinterQueueApplet

The `PrinterQueueApplet` displays a list of print jobs. Each printer has a separate list; the Printer pop-up menu selects which print job list is displayed. The View my jobs only checkbox filters the list of jobs in the print queue to display only the current user's print jobs. Cancel Selected Jobs removes the selected jobs from the print queue.

Help->About JavaOS

The Help->About JavaOS menu item displays an HTML page with a button that has a button labeled System Information. Pressing this button will display a table with a list of information about the current state of JavaOS. The HTML page for `SystemInfoApplet` is located in `/lib/html/sysinfo.html`.

SystemInfoApplet

The `SystemInfoApplet` displays the information in the following list. It does not have any user-interface features for controlling the behavior or configuration of JavaOS.

- System Information
- Machine Name
- Memory (k)
- IP Address
- Ethernet Address
- Home Directory Server
- Home Directory
- DNS Domain Name
- NIS Domain Name
- Platform Name
- JavaOS Version
- Build Information
- Language
- Country
- Variant
- Keyboard
- Time Zone

Adding New Keyboard Layouts

This chapter describes how to add new keyboard layouts to JavaOS. (This procedure is used by IBM for PC keyboards.) You should include a keyboard driver for each keyboard supported in JavaOS.

Overview

When a key is pressed, the keyboard emits a scan code and a keyboard driver converts the scan code into a key code, as defined by `KeyEvent` in JDK. A key can be pressed with keyboard modifiers, such as Shift, to type other characters. For example, if a user presses the modifier key Shift and a, the driver should interpret this combination as uppercase A. Using modifiers such as Shift, Control, and Alt, users can enter more characters than the number of keys on the keyboard.

The keyboard driver sends the character a if no modifier is pressed, or A if the a key is pressed while the Shift key is down.

The following list contains terms that you should know:

Modifier	A key (such as Shift, Control, and Alt) that maps a keystroke into a different character.
Group	Characters that can be typed by the same keystroke with different modifiers. For example, on a U.S. keyboard, the characters a and A are in one group.

Required Keyboard Data

You will need the scan code map for the new keyboard. An example appears in the following figure:

70	65	01	02	03	04	05	06	07	08	09	10	11	12	13	15	90	95	100	105
71	66	16	17	18	19	20	21	22	23	24	25	26	27	28	91	96	101		
72	67	30	31	32	33	34	35	36	37	38	39	40	41	42	43	92	97	102	106
73	68	44	45	46	47	48	49	50	51	52	53	54	55	56	57	93	98	103	
74	69	58	60	61										62	64	99	104	108	

FIGURE 6-1 Scan Code Map

You will also need to map the scan code into virtual key codes defined in `KeyEvent` in JDK1.1. An example appears in the following figure:

F1	F2	ESC	1	2	3	4	5	6	7	8	9	0	-	=	BKSP	NUM	/	*	-
F3	F4	TAB	q	w	e	r	t	y	u	i	o	p	[]	ENT	7	8	9	
F5	F6	CAPS	a	s	d	f	g	h	j	k	l	;	'	{		4	5	6	+
F7	F8	SHIFT	<	z	x	c	v	b	n	m	,	.	/		SHIFT	1	2	3	
F9	F10	CTL	ALT	SPACE										ALT	CTL	0	.	ENT	

FIGURE 6-2 Key Code Map

The keyboard driver is expected to pass a key code value for every keypress and keyrelease event. When a key is pressed and released, the key code map table will be used for this purpose. It should also supply the Unicode character value of the

key pressed. It returns different Unicode character values for the `keypress` event depending on which key masks, if any, are pressed (such as the Caps Lock or Shift key).

The set of Unicode characters formed by the same keystroke for different keymasks forms a group. For example, the list of groups could include `{a,A}`, `{1,!}`, where the first element of the group is the character with no key masks. The second character is returned if Shift is pressed or Caps Lock is on.

Developers must create a group for each key on the keyboard.

Adding a New Keyboard

The keyboard class uses the following naming conventions:

- The country name and keyboard number is based on the ISO3166 code.
- The keyboard name is formed by concatenating with `KeyTable`. For example, the Russian 443 keyboard is named `RU443KeyTable.java`.

```
class US103KeyTable extends
KeyTable {
}
```

To add a new keyboard:

1. Create an array for scan code -> key code mappings.

This class should be located in the `sun.javaos` package.

```
class US103KeyTable extends KeyTable {
private int[][] scancodeTokeycode = {
VK_UNDEFINED, //Scancode 0
VK_ESC, // Scancode 1, ...};
```

2. Create a two-dimensional array containing the list of groups.

In this code snippet you will find the first group has only one element and the second group has two elements. The first group has only one element as key has no special meaning even if it is pressed with different modifiers.

```
private char [] [] charmapArray = { {VK_ESC}, //Scancode 0
  {1,!}, // Scancode 1
  {2,@}, //Scancode 2
```

3. Implement two methods to publish the two tables to JavaOS.

```
protected int [][] getScanCodeArray() { return scancode_array};
protected char [][] getCharMapArray() { return charMapArray; };
```

The keyboard driver should return a lowercase a if a is pressed, but it should return an uppercase A if Caps Lock is on (since this is equivalent to pressing Shift and typing a). In order to implement this feature, the driver should implement the `getShiftIndex` method.

```
protected int getShiftIndex (KbdState kstate, int NLSFlag, int
vkeyNumber) {
  if (kstate.isCapsLock()){
    if ((vkeyNumber > 13 && vkeyNumber < 28) ||
        (vkeyNumber > 30 && vkeyNumber < 43) ||
        (vkeyNumber > 45 && vkeyNumber < 56) ) {
      return 1; // index in the group
    }
    return 0; //index in the group
  }
}
```

Adding Compose Files

This chapter describes how to add new compose files for European (including eastern European) locales in JavaOS.

Overview

Most European and Asian languages translate keystrokes according to the current state. Users can change from one state to another using a toggle key. The examples in the following sections are part of a compose file for the Russian locale.

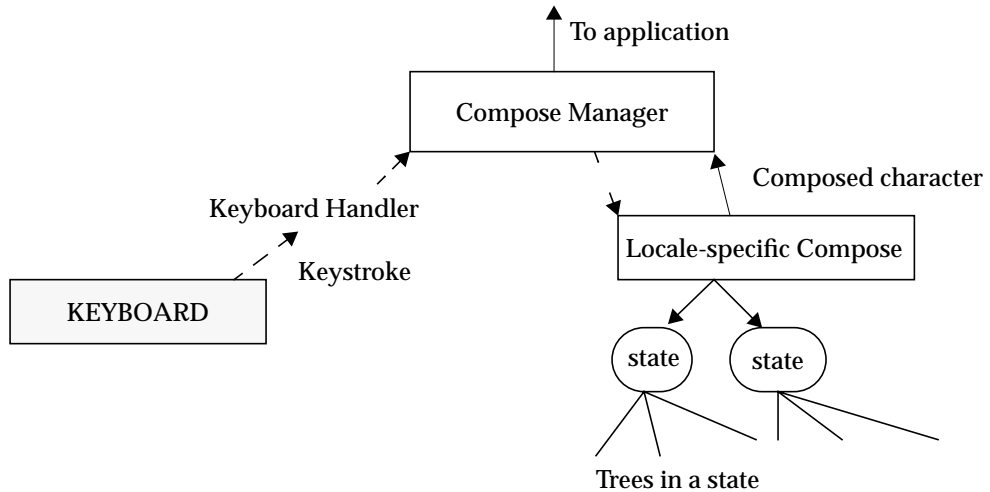
The following list contains some terms that you should know:

state	The active input mode. For example, Russian has two states: ASCII and Russian. In ASCII state, keystrokes are entered as is. In Russian state, each keystroke is mapped to a Russian character.
rule tree	The sequence of keystrokes for typing a character.

Architecture

As the following figure shows, when a user presses a key, the keyboard handler receives the scan code and converts it into virtual key constants. The compose file deals with only virtual key constants to avoid any keyboard dependency. The keyboard handler passes to the keystroke to the Compose Manager. The Compose

Manager then binds with the locale-specific compose class and passes the keystroke. The locale-specific `compose` class returns the appropriate state or composed character. The Compose Manager returns the composed character to the application.



Adding a New Compose File to a Locale

Compose files are Java files that contain the state and tree of the language input. A compose file can be written by implementing the interface `sun.javaos.im.local.compose.ComposeIM`. The resulting class file should be in the package `sun.javaos.im.local.compose`.

For example, assume you want to implement a Russian compose file named `ISO8859_5.java`.

1. Make sure the program starts with the following lines:

```
package sun.javaos.im.local;
public class ISO8859_5 implements ComposeIM{
    ...
}
```

2. Identify the number of distinct states the input method can have.

3. Register the number of distinct states with the `IMStateManager` class.

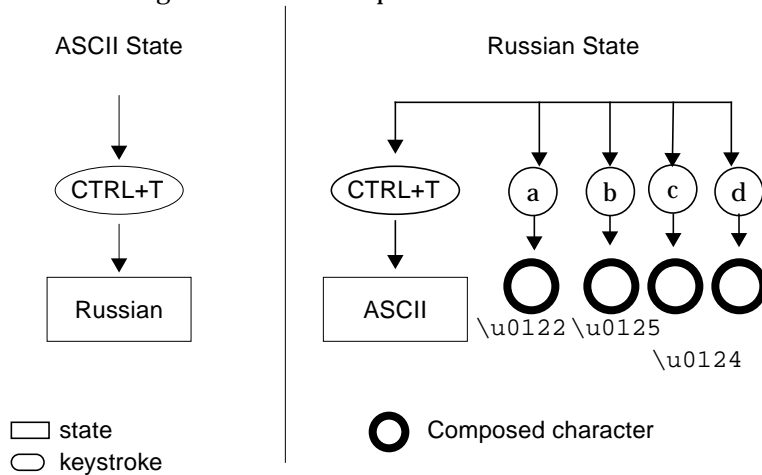
In this example there are two states: ASCII and Russian. The following sample shows how to register the two states:

```
IMStateManager imState = IMStateManager.getInstance();
IMState englishIM = imState.getState("[ASCII]");
IMState ruIM = imState.getState("[Russian]");
englishIM.setStatusVisible (true); //make label [ASCII] visible
ruIM.setStatusVisible (true) ; // make label [Russian] visible
```

If the input mechanism does not support more than one state, you need not show the state.

Say a user types a series of keystrokes to enter a character or change the state. The list of characters and change state is described using `IMTree` object. The following example shows the keystroke `Control+T`, which toggles the state from ASCII to Russian. In ASCII mode all keystrokes are mapped to the same characters. In Russian mode the characters a, b, c, and d are mapped to Russian characters.

The state diagram for this example would look as follows:



Each path to an end node (either a composed character or a state) should be programmed as an `IMTree` object in the `compose` class. The code to program this state is explained as follows:

Consider the ASCII state. It has only one path and one node with Control+T and it results in an end state, Russian. It will be coded as shown below:

```
char[] toggle = { '\u0014' }; // Ctrl+T toggle character
ImTree buf[] = new ImTree[10];
buf[0] = new ImTree();
buf[0].keycode = new String (toggle,0,toggle.length);
buf[0].keychar = new String (toggle, 0, toggle.length);
buf[0].modifier = InputEvent.CTRL_MASK;
buf[0].modifier_mask = InputEvent.CTRL_MASK;
buf[0].target_state = ruIM; // target state
englishIM.putTrees (buf, 1, // number of elements
null , // Composed character or string, status only changed
"[Russian]", buf[0].keycode );// State it should goto
```

The `ImTree` object contains information about the keycode and modifier associated with the keystroke. After the object is constructed, use the `putTrees` method to insert the compose rules.

In the Russian example, a total of five trees must be added: four for processing characters and one to toggle the state. You can create a table for the four characters. The following sample shows the result of using `putTree` and shows the code for toggling the state from Russian to ASCII.

```
char[] toggle = { '\u0014' }; // Ctrl+T toggle character
ImTree buf[] = new ImTree[10];
buf[0] = new ImTree();
buf[0].keycode = new String (toggle,0,toggle.length);
buf[0].keychar = new String (toggle, 0, toggle.length);
buf[0].modifier = InputEvent.CTRL_MASK;
buf[0].modifier_mask = InputEvent.CTRL_MASK;
buf[0].target_state = englishIM; // target state
ruIM.putTrees (buf, 1, // number of elements
null , // Composed character or string, status only changed
"[Russian]", buf[0].keycode );// State it should goto
```

The following code fragment covers only the processing of four characters.

```
char keys[][] = { {'a' , 0, '\u0122'},
{'b',0,'\u0123'}, {'c',0,'\u0124'}, {'d',0,'\u0125'}}
//character sequence terminated by null character and last element
is the composed character
for (int i=0; i<4;i++) {
int n = 0;
buf[n]=new ImTree();
while (keys[i][n] != 0) {
buf[n].keycode = new String (keys[i][n],n,1);
buf[n].keychar = new String (keys[i][n],n,1)
// Add control mask if required
n++;
}
ruIM .putTrees (buf,n,keys[i][n],//Composed char
null, null); // no state change
}
```

In addition to these steps, you need to implement the dummy method `getState()` to satisfy the interface requirements. This method may be used in future releases.

Setting Up and Testing the Program

Use the following procedure to set up and test the program:

- 1. Compile the program and add this class into the `REMOTE` directory or to the JavaOS source tree.**
- 2. Set the property `javaos.im.compose_<localname>=<classname>` using dhcp settings.**
For example, `djavaos.im.compose_th=ISO8859_5`.
- 3. Boot the Java station in the given locale.**
You should see [ASCII] in the status window of the login dialog.
- 4. Press Control+T to toggle between Russian and ASCII states.**
- 5. In Russian mode, type a b c d and check the Russian characters.**

