



JavaOS™ for Business™



JavaOS™ for Business™ Version 2.0
Application Development Guide

JavaOS™ for Business™ Version 2.0
Application Development Guide

©Copyright 1998 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, 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, 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, the IBM Logo, OS/2 are trademarks or registered trademarks of IBM Corp. in the United States and other countries, and are used under license by Sun Microsystems.

Microsoft, Windows, Windows NT, and the Windows logo are registered trademarks of Microsoft Corporation.

Intel and EtherExpress are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

The OPEN LOOK and Sun(TM) 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., 901 San Antonio Road, Palo Alto, California 94303, Etats-Unis.; IBM Corporation, Old Orchard Road, Armonk, New York 10504. Tous droits réservés.

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 décompilation. 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 Etats-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 écrites de Sun. L'accord du gouvernement américain est requis avant l'exportation du produit.

Contents

Chapter 1. Introduction to JavaOS™ for Business™	1
Network computers	1
Features of JavaOS for Business	1
Chapter 2. Introduction to the JavaOS™ for Business™ Software Development Kit (JSDK)	3
Chapter 3. Development Environment	5
Software Requirements	5
Harddisk Requirements	5
Installing the JSDK on a Development System	5
JSDK Directory Structure	6
Configuring the JSDK	6
JSDK Documents	7
JSDK Tools	7
JAR	7
ICAT Debugger	7
Bldlevel	8
Public methods/APIs	8
Building samples	9
Chapter 4. Running an Application on JavaOS for Business	11
Chapter 5. Packaging the application	13
Packaging the application using the Application Load Settings JAR file	13
Packaging the application using your own load JAR file	13
Chapter 6. Loading an application	15
Using the Application Load Settings JAR file provided by JavaOS for Business	15
Implementing your own load JAR file	18
Creating a load JAR file	18
Making the load JAR file available to the JCT	24
Adding the loadable service to the application	24
Chapter 7. Customizing an application	27
Appendix A. JavaOS Service Loader	29
Appendix B. JavaOS Configuration Tool	31
Appendix C. JavaOS System Database	33
Temp namespace	33
Interface namespace	33
Alias namespace	33
Software namespace	34
Device namespace	34
Appendix D. Building the sample application	35
Appendix E. Building device driver samples	37

Index 39

About this book

This book provides the following information:

- How to install the JavaOS for Business Development Kit (JSDK)
- How to build the sample application and device drivers
- How to run applications on JavaOS for Business
- JavaOS for Business methods and classes for loading and configuring

This book does not provide fundamental information about JavaOS for Business. See the *JavaOS for Business Reference* for more information on JavaOS for Business.

For information on how to write device drivers see the *JavaOS for Business Device Driver Guide* .

Who should read this book

The JavaOS for Business Application Development Guide is intended for application programmers who plan to run applications on the JavaOS for Business operating system.

You should already be familiar with:

- Java programming
- Java Development Kit (JDK) 1.1.4
- Microsoft Windows NT 4.0

Conventions and terminology used in this information

This information uses the following typographical conventions:

italics Used to specify a variable. Substitute your variable for the word in italics. For example, The *user_ID* can be up to 24 bytes long.

bold Used to specify a command. For example, type **dump** to start a system dump.

monospace

Used to show coding samples. For example,

* @(#) FirstFileCustomizer.java is used to show a code fragment.

Required and related information

The following information should be used with this book:

- *JavaOS for Business Reference*
- *JavaOS for Business Network Operations*

Chapter 1. Introduction to JavaOS™ for Business™

The JavaOS™ for Business™ operating system is designed for network computers and based on Java technology produced by Sun Microsystems, Inc. While other Java environments run on top of existing operating systems, JavaOS for Business provides just enough operating system support services so that a network computer can manage its resources and support a 100% Pure Java environment.

JavaOS for Business provides better performance and a substantially reduced memory footprint because it was designed from the beginning to support Java applications and does not need to provide backward compatibility with legacy workstation applications.

Network computers

As corporations moved from mainframe computing to personal workstations, the size and complexity of enterprise networks increased proportionally. The cost of purchasing a full-function workstation is only a small fraction of the total cost of ownership, which includes the support and maintenance expenses that are incurred over the workstation's lifetime, such as installing and updating system and application software, technical support and troubleshooting, and end-user education. Add to these costs the cost of the networking infrastructure necessary to interconnect the workstations and the result is an expensive and complex operation over which enterprises need better control.

The network computer reduces the total cost of owning workstations, particularly for large corporations. A network computer contains no hard disk or application software. When the network computer boots, it loads its operating system and application software over the network and then runs them locally.

As a result, system and network administrators do not have to maintain each workstation individually. Because the system and application software resides on one or more servers, the software needs to be updated only once on the servers. The next time the network computer needs to run an application, the most recent version is obtained from the server, making software updates to network computers automatic. The JavaOS for Business operating system software and device drivers are also maintained this way, allowing network computers to refresh their operating system software simply by rebooting from the server.

Users benefit also. Because they are freed from backing up their own data and managing new operating system and application software updates, they can concentrate on their important tasks.

Features of JavaOS for Business

JavaOS for Business builds upon the prior versions of JavaOS by providing the following new and improved features:

- Support for JDK 1.1.4 applications and applets
- Improved performance and memory management
- Dynamically loadable device drivers and applications
- A layered architecture allowing pieces of the operating system to be independently updated and replaced
- Centralized administration of network computers and associated applications from the server
- Improved reliability, availability, and serviceability of the JavaOS for Business operating system and applications running on the network computers

Chapter 2. Introduction to the JavaOS™ for Business™ Software Development Kit (JSDK)

The JavaOS for Business Software Development Kit (JSDK) is a JavaOS development kit that runs on the Windows NT 4.0 operating system.

The JSDK provides the class files, documentation, samples, and tools needed to develop applications and downloadable system components that use services that are unique to JavaOS for Business.

Note: If you want to write Java™ code that uses only the Java Development Kit (JDK) classes—that is, 100% Pure Java code—you do not need to use the JSDK.

JavaOS for Business supports Java APIs and is capable of running any 100% pure Java application or applet. In addition, there are APIs specific to JavaOS for Business that an applet or application could use. However, applications or applets that use APIs specific to JavaOS for Business can run only on JavaOS for Business.

The JSDK Version 1.0 contains:

- Documentation
- Class/Methods Library
- Samples
- Java Development Kit (JDK) 1.1.4
- JavaOS for Business Tools

Chapter 3. Development Environment

This chapter describes how to install and use the development environment provided with the JSDK.

Software Requirements

The JSDK requires that the following software be installed:

- Microsoft Windows NT 4.0 with service pack 3
- Java Development Kit (JDK) 1.1.4 (supplied by the JSDK)

Harddisk Requirements

JSDK requires approximately 150 MB of hard disk space. The space requirement for each of the JSDK files (that is, the size of each file after unpacking) is:

JSDK100.ZIP 24 MB

ICATJOT.ZIP 16 MB

JDK114.ZIP 25 MB

JDK114DOC.ZIP 11 MB

Installing the JSDK on a Development System

The JSDK.ZIP file contains the following files:

- JavaOS for Business Software Development Kit main ZIP file (JSDK100.ZIP)
- ICAT Debugger for Microsoft Windows NT (ICATJOT.ZIP)
- Java Development Kit 1.1.4 (JDK114.ZIP)
- Java Development Kit 1.1.4 Documentation (JDK114DOC.ZIP)

All files are compressed into the ZIP format. Currently, you can obtain an unzip utility from Info-ZIP at <http://www.cdrom.com/pub/infozip/Info-Zip.html>. IBM makes no representation or warranty that an unzip utility will continue to be available from Info-ZIP.

You must have JDK 1.1.4 installed on your system before using the JSDK. It is provided as JDK114.ZIP.

To install the JDK, unzip the JDK114.ZIP.

Configuration instructions are provided in the JDK documentation. Unzip the `jdk114doc.zip` file and use your favorite browser or HTML viewer to view the JDK 1.1.4 documentation.

To install the JSDK, unzip the JSDK100.ZIP file.

Leaving the JAVAOS.ZIP, OCFBIN.JAR files, and the JSDK documentation in the same JSDK100 directory (the directory tree created when you unzipped the JSDK100.ZIP file), ensures that HTML links in the JSDK and demo files work properly.

Note: Included in the unpacked files are the files `JSDK100\lib\JAVAOS.ZIP` and `JSDK100\lib\OCFBIN.JAR`. *Do not unzip these files.* These files contain the JavaOS for Business class files, and must remain in their packed form for the JSDK to use them.

JSDK Directory Structure

The following directory structure is created when you unzip the `JSDK100.ZIP` file.

Directory	Contents
<code>JSDK100\bin\</code>	JSDK tools such as JAR and <code>bdlevel</code> .
<code>JSDK100\demo\</code>	Sample programs for an application and device drivers. See “Building samples” on page 9 for more information.
<code>JSDK100\lib\</code>	<code>Javaos.zip</code> and <code>ocfbin.jar</code> files for the JavaOS for Business classes.
<code>JSDK100\docs\</code>	Documentation for the JSDK. See “JSDK Documents” on page 7 for a list of the documents.
<code>JSDK100\readme</code>	Latest breaking news about JavaOS for Business Software Development Kit.
<code>JSDK100\changes</code>	(No changes for this release.)
<code>JSDK100\copyright</code>	ASCII file that contains the copyright statement for the JSDK.
<code>JSDK100\license</code>	ASCII file that contains the license agreement for the JSDK.
<code>JSDK100\index.html</code>	A roadmap with descriptions and links to each of the files in the JSDK.

The organization of the JSDK files roughly parallels the file organization of JDK 1.1.4.

Configuring the JSDK

Configuring the JSDK involves updating your `CLASSPATH` environment variable to include the JSDK class file.

To setup `CLASSPATH` for this kit, you must add these items to your `CLASSPATH`:

- **JSDK100\bin**
- **JSDK100\lib\JAVAOS.ZIP**

Use the following command to add these items to the end of your `CLASSPATH`: (NOTE: this assumes the JSDK is installed in `d:\JSDK100`)

```
set CLASSPATH=%CLASSPATH%;d:\JSDK100\bin;d:\JSDK100\lib\JAVAOS.ZIP;
```

If you also wish to use the OpenCard framework APIs, you must also add the following item to the `CLASSPATH`:

```
set CLASSPATH=%CLASSPATH%;d:\JSDK100\bin;d:\JSDK100\lib\JAVAOS.ZIP;  
d:\JSDK100\lib\OCFBIN.JAR
```

Use the following command to add these items to the end of your `CLASSPATH`: (NOTE: this assumes the JSDK is installed in `d:\JSDK100`)

```
set CLASSPATH=%CLASSPATH%;d:\JSDK100\bin;d:\JSDK100\lib\javaos.zip;
```

d:\JSDK100\lib\ocfbin.jar

Use the following command to verify that CLASSPATH has been setup correctly:

set CLASSPATH

You should see the JSDK paths at the end of the path.

JSDK Documents

The JSDK provides the following books from the JavaOS for Business library:

JavaOS for Business Network Operations System and network administrators who need to plan for, install, and maintain the JavaOS for Business system on a day-to-day basis.

JavaOS for Business Reference Programmers who need information on the classes and methods unique to JavaOS for Business.

JavaOS for Business Device Driver Development Guide Programmers who want to add a new device driver or modify an existing device driver and make that driver available to JavaOS for Business network computer users.

JavaOS for Business Application Development Guide Programmers wishing to create and deploy a new desktop application that takes advantage of the features in the JavaOS for Business operating system.

JSDK Tools

The JSDK provides the following JavaOS for Business tools:

- JAR
- Interactive Code Analysis Tool (ICAT) Debugger for Windows NT
- Bldlevel

JAR

The JAR tool provided in the JSDK accepts the same parameters as the JAR tool provided in the JDK. To invoke the version of jar provided in the JSDK, you need your CLASSPATH setup to point to the **JSDK100\bin** directory and to use the Java interpreter by typing the following at the command line:

```
java ibm.tools.jar.Main <parameters>
```

ICAT Debugger

The Microsoft Windows NT 4.0 version of the Interactive Code Analysis Tool (ICAT) for JavaOS™ for Business™ provides debugging of the JavaOS for Business microkernel, its Java Virtual Machine (JVM), the JVM Java applications, and their (potential) native-method calls (C or C++ code collected in load libraries).

For more information see the documentation provided in:

JSDK100\docs\icatnt

Bldlevel

Bldlevel is a server-side tool that extracts version information from a JavaOS file. In the case of a JAR or ZIP file, it locates the manifest entry and parses and displays the version information that it contains. In case of the JavaOS binary image file, the manifest file is part of the ROM filesystem. Bldlevel reads the JavaOS binary image file, attempts to locate the manifest file entries, then parses and displays the build level information that it contains.

Bldlevel tool relies on the manifest file to adhere to a format. The following example shows the file format and the information contained in each field.

```
Manifest-version:  
Name: =  
Specification-Title: =  
Specification-Version: =  
Specification-Vendor: =  
Package-Title: =  
Package-Version: =  
Package-Vendor: =
```

If the tool is unable to locate the manifest entries, it outputs an error message: Unable to find Version or Build information.

To access the bldlevel tool, point your CLASSPATH to the BLDLEVEL.ZIP file provided in the **JSDK100\bin\com\ibm\jossrv\versioning** directory. To invoke the bldlevel tool, type the following from the command line:

```
java com.ibm.jossrv.versioning.bldlevel <parameters>
```

where <parameters> can be either “?” to display help or the path to the file from which you want to extract version information.

Public methods/APIs

The JSDK contains the JAVAOS.ZIP file in the lib directory. This file contains the JavaOS for Business extension classes that you can use to build device drivers and applications.

The kinds of classes present in the javaos.zip file are:

- JavaOS Service Loader (JSL)
- JavaOS Device Interface (JDI)
- Event manager
- JavaOS System Database (JSD)
- Networking
- I/O
- Kernel level (memory, interrupts, DMA, etc)

Documentation on these classes is provided in the *JavaOS for Business Reference* in the **JSDK100\docs** directory.

The OCFBIN.JAR file provides the OpenCard framework APIs. For more information see the OpenCard API documentation at <http://www.opencard.org/OCF/1.0/nc>

Building samples

The JSDK provides code samples for an application and for device drivers in the **JSDK100\demo** directory.

See Appendix D, “Building the sample application” on page 35 for information on how to build the sample program.

See Appendix E, “Building device driver samples” on page 37 for a list of the device driver samples provided and for information on how to build each sample.

Chapter 4. Running an Application on JavaOS for Business

This chapter provides methods for running an application in the JavaOS for Business environment.

A network computer, by definition, has no persistent storage and no installed software other than for the BIOS or microcode that allows it to boot from a network server. Before a network computer can be used, the appropriate network services must be set up and configured on the server; the selected main application must be defined on the server to become available to the network computer.

JavaOS for Business supports the execution of one application which is referred to as the main application. The main application is the application that the system loads and runs after a user logs in. By default, JavaOS for Business provides the HotJava Browser as the main application. You must define any application you use, including the HotJava Browser, to JavaOS for Business.

Defining the application consists of storing the loading and configuration information for the application in the JavaOS System Database (JSD)—the system repository that contains the information about the JavaOS for Business system.

JavaOS for Business provides a graphical user interface tool, the JavaOS Configuration Tool (JCT), to add, modify, and change the system information stored in the JSD.

The steps to define an application on JavaOS for Business are:

- Defining HotJava as the main application
 - Load the application

Use the JCT to load and configure the HotJava Load Settings file. See *JavaOS for Business Network Operations* for more information.
 - Configure the application

Use the JCT to load and configure the HotJava Configuration file. See *JavaOS for Business Network Operations* for more information.
- Defining an application other than HotJava as the main application
 - Packaging the application

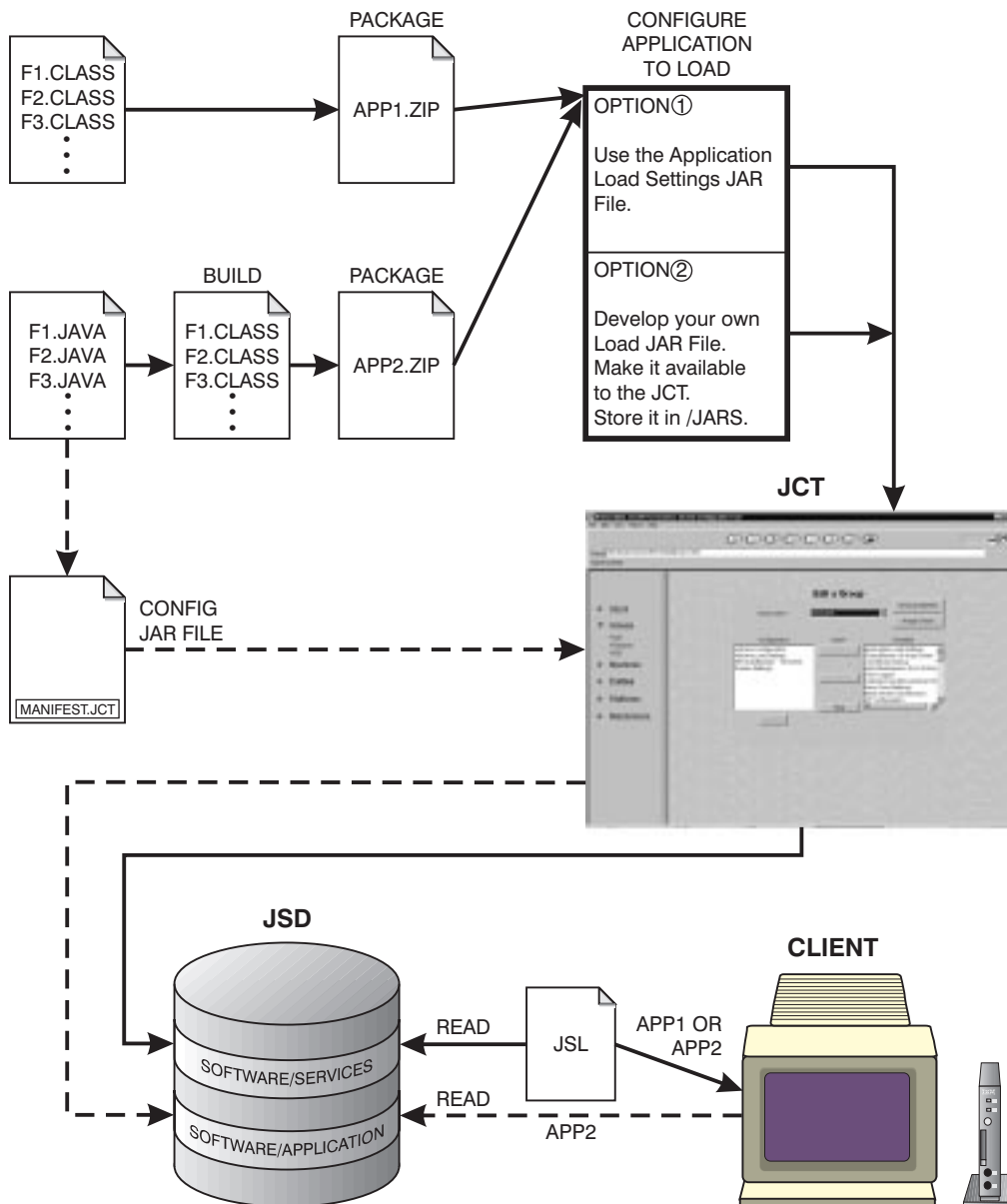
The application's class files are bundled into a ZIP or JAR file. See Chapter 5, “Packaging the application” on page 13 for more information.
 - Loading the application

This step consists of using the JCT to add the application information to the JSD.

There are two ways to load an application on JavaOS for Business. The methods are described in Chapter 6, “Loading an application” on page 15.
 - Customizing an application

You can only customize an application when you have access to the application source code. Use the JCT to load and configure your configuration JAR file. See Chapter 7, “Customizing an application” on page 27 for more information.

The following figure depicts two applications, App1 and App2, and the procedures involved to run each application. Note that there is no source code available for App1, but source code is available for App2. Customization, the config JAR file, is available only for App2 because it requires change to the source code.



Chapter 5. Packaging the application

There are some packaging requirements to load an application, other than the HotJava browser, on JavaOS for Business.

Packaging requirements depend on the procedures you adapted for running the application:

- Packaging the application to use the Application Load Settings JAR file provided by JavaOS for Business
- Packaging the application to use your own load JAR file

Packaging the application using the Application Load Settings JAR file

The Application Load Settings JAR file requires the application runtime to be packaged in a ZIP file.

The Application Load Settings JAR file does not take advantage of the dynamically loading class files from the application server; therefore, all files must be included in the ZIP archive.

Applications are typically made up of classfiles and other files such as .gifs, .jpegs, .txt, .html, properties, and other files that make up your application contents. Many application hierarchies are organized so that all the class files are in the **classes** directory, and all other files are in the **lib** directory. The Application Load Settings JAR file requires that all classes must live in the **classes** directory. You can place all other files in the **lib** directory. If your application includes classes packaged into JAR files, these should be placed in the **classes** directory. They will be unbundled when the application is loaded.

See “Using the Application Load Settings JAR file provided by JavaOS for Business” on page 15 for information on how to use the Application Load Settings file.

Note: This procedure only supports the ZIP file format.

Packaging the application using your own load JAR file

When you create your own load JAR file, you can take advantage of the Dynamic Loading Services of JavaOS for Business.

Your application needs to be packaged in a ZIP or a JAR archive format to be loaded by your load JAR file. The archived file must at least include the class files that implement the Service class and the ServiceInstalnce class. All other application files may remain on ther server. The location of the application files is specified in the Business Card parameters that are defined when you create an application load JAR file.

At runtime these packages have to be retrieved, unbundled, and instantiated. These services are performed by the JavaOS Service Loader, which uses unbundlers to unbundle a package into its separate class and data files. Default unbundlers are provided that support the ZIP bundle types.

Applications that provide their own unbundling services must implement the Java Unbundle interface and advertise their services. The JavaOS Service Loader will query the /Interfaces namespace to identify providers of these services.

public void doUnbundle(Url bundleUrl, String appPrefix, String rsvd) throws IOException The implementor retrieves the service package specified by the URL argument and uses FileOutputStream JDK methods to store the separate class and data files into the local file system. In the case of JavaOS for Business, the local file system is defined to be /SYSTEM.

If during the unbundling process errors are encountered that prevent the loading of a service bundle into the File Loader's virtual file system, error log events are generated. The encountered errors could be the result of bundle problems, such as configuration errors, or system problems, such as low-memory conditions. The behavior of the JavaOS Service Loader to the application requiring a class that cannot be unbundled is identical to file-open errors that are reported when a file cannot be found.

Chapter 6. Loading an application

A brief overview of the methods for loading an application on JavaOS for Business is provided below.

- Using the Application Load Settings file provided by JavaOS for Business

This method of loading is basically for the quick loading of applications for testing and for trial applications. However, this procedure is restricted and it does not allow the class files to be dynamically loaded from the server. This procedure is documented in “Using the Application Load Settings JAR file provided by JavaOS for Business.”

- Implementing your own load JAR file

This procedure provides the steps to create your own load JAR file. When you use this procedure, you can use the dynamic loading service that provides the capability to retrieve the needed classes over the network dynamically, on an as needed basis, freeing up critical system resources and optimizing performance. The steps to implement your own load file are documented in “Implementing your own load JAR file” on page 18.

This procedure requires the following:

- Creating a JAR file
- Adding the loadable service to the application

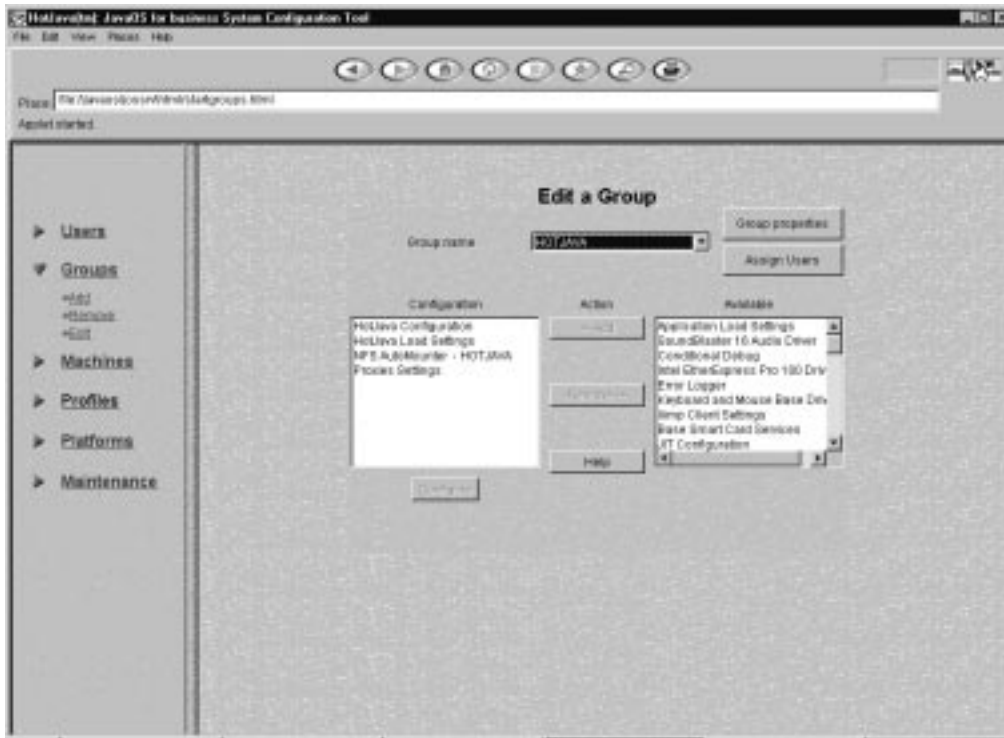
Using the Application Load Settings JAR file provided by JavaOS for Business

To load an application as the main application on JavaOS for Business, the application must be written in Java. The following are also required:

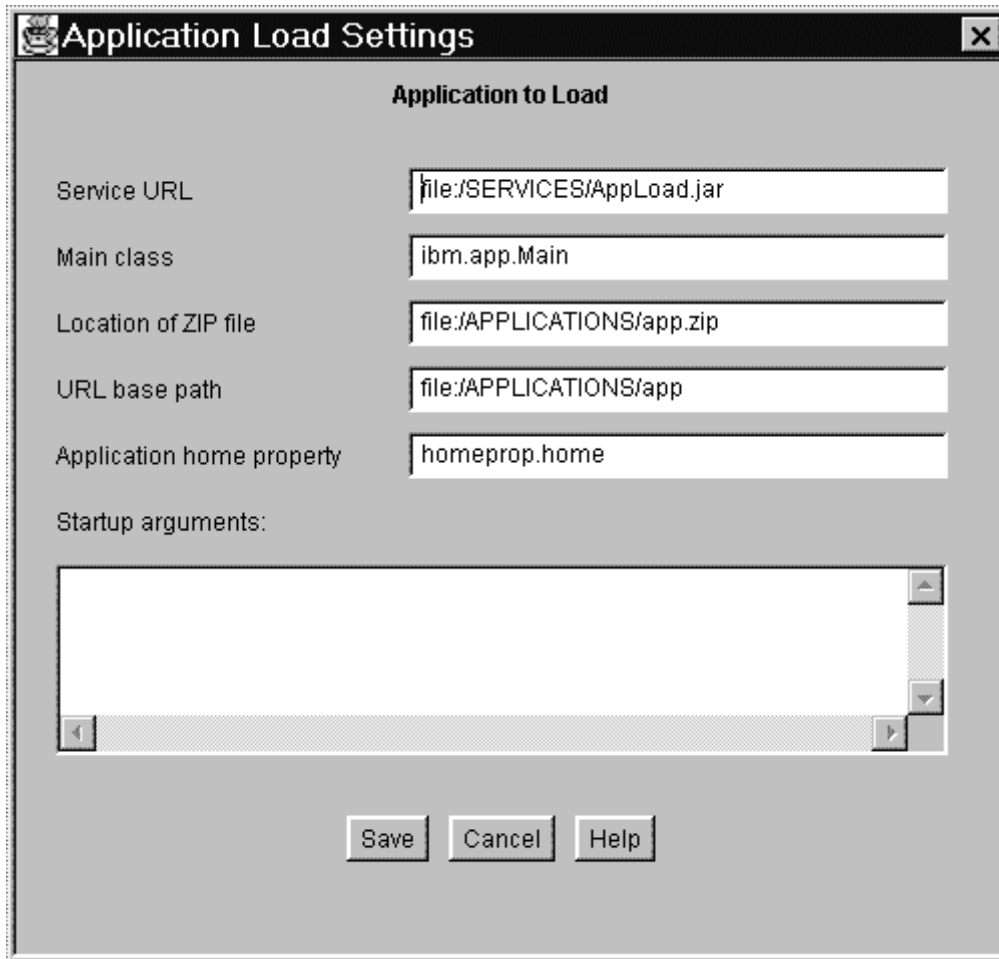
- The application must run on Java Developer's Kit (JDK) 1.1.4
- The application class files must be archived into a ZIP file.

You can also include the JavaOS for Business extensions in your application.

Use the JavaOS Configuration Tool edit panels to add the Main Application from the **Available** list. Select Application Load Setting from the Available list and press **Add**.



The following panel is displayed:



Specify the settings for the application:

- Service URL

The name of the loadable service. The provided figure has `file: /SERVICES/AppLoad.jar` in this field. You can change `/SERVICES/` to point to the NFS mount point where your loadable services are located. However, do not change `AppLoad.jar`. This is the name of the Load JAR file provided by JavaOS for Business

- Main class of the Java application

Enter the name of the main class of your application.

- Location of the ZIP file

Applications are bundled as ZIP archives. The system refers to them with URLs. You can store your applications anywhere that a URL can be stored; on the network or locally at the network computer in flash ROM (read-only memory). Because the archive file is a URL, the system can easily download it from many places.

- URL base path

Enter the path of the URL.

- Application home property

If your application uses this property, enter the name of your application home property. For example, for HotJava browser, this is `hotjava.home`. In the application startup properties text area, which is described below, you must identify where the home property is located.

- Startup arguments

Enter arguments to be passed to the application. These can be in the format of key/value pairs separated by the equal sign (=) or as individual arguments. For example, the location of the home property might be:

```
hotjava.home = file:/HOME/hotjava
```

Enter each argument on a separate line.

Implementing your own load JAR file

This section describes the steps for loading an application on JavaOS for Business by implementing your own load JAR file. If you are not interested in creating a load JAR file, refer to the information in “Using the Application Load Settings JAR file provided by JavaOS for Business” on page 15.

The JavaOS System Database (JSD) stores the loading information about JavaOS for Business components such as devices and system software services that are installed, user and group attributes, device drivers, and application-specific information. The set of parameters that contains the loading information for an application is called Business Card. See Appendix C, “JavaOS System Database” on page 33 for more information. The JavaOS Configuration Tool (JCT) is used to place the Business Card information in the JSD. See Appendix B, “JavaOS Configuration Tool” on page 31 for more information. The JavaOS Service Loader (JSL) handles the loading and unloading of services (application) using the Business Card information in the JSD. See Appendix A, “JavaOS Service Loader” on page 29 for more information.

1. Create a load JAR file.

The load JAR file contains the loading information for the application. The JCT uses this file to place the loading information for an application in the JSD. See “Creating a load JAR file” for more information.

A load JAR file contains the following:

- Business Card

This is the set of parameters that contains loading information; it is stored in the JSD for each set of loadable services. See “Creating the information for the Business Card” on page 19 for more information.

- A MANIFEST.JCT file

This file is used by the JCT to determine the location of the Business Card information in the JSD. See “Creating a manifest file for the JCT” on page 22 for more information.

2. Make the load JAR file available to the JCT.
3. Add the loadable service to the application.

This step includes a start method that gets the JSD entry and performs static initialization of the service

4. Package the application for loading.

See Chapter 5, “Packaging the application” on page 13 for more information.

Creating a load JAR file

A sample load JAR file is provided in the directory:

JSDK100\demo\SampleApplication\LoadBean

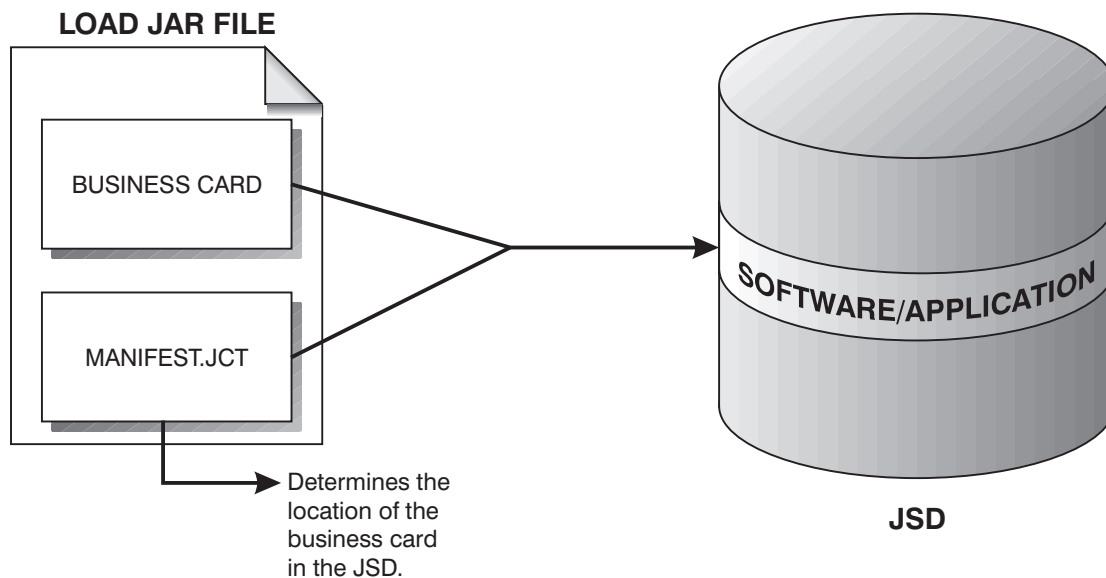
The provided sample AppLoadSampleEditor.java shows how the load JAR file is configured for the JCT to read it.

The JAR file contains the information that the JSL needs in order to load the application. This set of parameters is called a Business Card.

The Dynamic Loading Service is a JavaOS for Business feature that is used to retrieve classes over the network on an as needed basis. This service frees up critical resources and improves performance. This is an optional feature; JavaOS for Business supports the loading of applications with or without the use of the Dynamic Loading Service.

To use the Dynamic Loading Service, you must include specific parameters in the JAR file. These parameters are marked with an * in the section “Creating the information for the Business Card.”

You also need to create a manifest file for the JCT, the MANIFEST.JCT file, to determine the location of the load JAR file in the JSD. See “Creating a manifest file for the JCT” on page 22 for more information.



Creating the information for the Business Card

The parameters of the Business Card provide the configuration information necessary to retrieve the interface and advertise the interfaces it supports. BusinessCard configuration parameters are contained in the JSD/Software/Services namespace, in nodes uniquely identified by a META-property.

The Business Card interface is a set of configuration parameters comprised of both private and framework parameters. The private parameters are used by the application to configure itself, and the framework parameters are used by the JavaOS Service Loader to discover, load, advertise, and instantiate the service.

The following code fragment shows the Business Card parameters used in a load JAR file:

```

bundleURL = "";
bundleType = "ZIP";
bundleInstanceInfo = "ApplicationSample,ApplicationInstance,
                    sun.javaos.Application,Desktop";
bundleInitClassName = "ApplicationService";
bundlePackageName = "";
bundleClassPath = "";
bundlePropertyName = "";
isBusinessCard = true;
loadsUponConnection = true;
loadsWhenDiscovered = false;
loadsWhenMatchedToEntry = false;

```

Business Card framework parameters are described below.

The parameters marked with an * are required only if you are using the dynamic loadable services.

bundleURL URL-formatted string that contains the name and location of the service.

Note: File URLs must reference a mount point that is already started by JavaOS for Business.

bundleType String specifying the bundle packaging type. This parameter is used to match a service with an unbundler. If this parameter is not present and a bundleURL exists, a default bundler type of JAR is assumed.

* **bundleResourceBundleURL** URL-formatted string that contains the location of the bundle's resource bundle.

Note: The resource bundle contains all the translatable text used by the service, including error logging and message box services.

bundleInitClassName Name of the class within the package that implements the Java Service interface. If this parameter is not present, static initialization of the package is not performed.

bundlePackageName Prefix name for the package. This is a required parameter that is used internally by the JSL to manage the File Loader.

Note: The classes and data files contained in each package must descend from a unique Java package prefix.

* **bundleClassPath** URL-formatted string that contains the path prefix used to search for a class that belongs to this package but was not included in the bundle. The File Loader uses this parameter to populate the logical file system, as the files are needed.

bundleInstanceInfo String that contains the information necessary to advertise and create a service instance. Each service instance described must, at a minimum, specify a logical name and the name of the class implementing the ServiceInstance interface. It can optionally contain an advertised interface and an alias name. The format of this string is as follows:

```
"logicalName,InstanceName,InterfaceName,AliasName"
```

Multiple entries of this type can be specified, delimited by a ";" as shown in the following example:

```
InstanceInfo="myFax, myFaxInstance, javaos.util.fax,DefaultFax:
             myfax2, myFaxInstance, javaos.util.fax";
```

Note: In addition to the advertised interface that the service implements, it must also implement the ServiceInstance interface, which is used by the JavaOS Service Loader to create, delete, and manage the driver instances.

* **bundlePropertyName** This parameter is required only if the package contains data files. In this case, it is necessary to add a unique prefix to the file name when using JDK methods to access the file. This parameter specifies a string that is used by the JSL to store a unique prefix in the System Properties. A

service must retrieve this prefix from System Properties, using this parameter value as the properties keyword.

isBusinessCard This parameter must be set to true for the JavaOS Service Loader to discover, load, advertise, and instantiate the service.

loadsWhenMatchedToEntry During the discovery phase of the Device Namespace manager the *matchingName* and *compatibleMatchNames* parameters are used to search for a matching service. If the entry matches and this boolean is true, the service is loaded.

loadsWhenDiscovered During the load phase the Device Namespace manager runs through all the BusinessCard entries. If an entry is found with this property set, the service is loaded.

loadsUponConnection This service is loaded only when an application requests a ServiceConnection to the service.

* **matchingName** A string representing a unique device identification. This field must follow the formatting rules specified in IEEE 1275 (that is, "pci1234,9999").

* **compatibleMatchingNames** A string array of compatible device-matching names that can also be supported by this driver. These strings are supplied to handle the case where a primary driver to support a device cannot be found.

Version String containing the version of this bundle.

Creating explicit advertisements: Normally, advertisements are created directly from a service's Business Card. However, there can be situations in which services will want to add their own advertisements. The following JavaOS Service Loader Class methods are available for this purpose:

- **public static ServiceAdvertisement Add Advertisement(String logicalName, String instanceName, String advertisedInterface, Entry businessCardEntry)**

This method explicitly requests the creation of a ServiceAdvertisement.

- **public static boolean RemoveAdvertisement(Service Advertisement)**

This method explicitly removes existing ServiceAdvertisements and returns true if advertisements are successfully removed.

How applications find and connect to a service: Applications use the Service Loader Class to find and collect information on services that have advertised support for an interface. The advertised interface is published by the service's Business Card bundleInstanceInfo parameter. The following methods are provided:

- **public static Enumeration findAdvertisements(String interface) throws ServiceException**

Given the interface string, (for example, jav.util.fax), this method returns an Enumeration of ServiceAdvertisement objects. The enumeration includes one object for each service that has advertised its support for that interface. Each ServiceAdvertisement object is used on other ServiceLoader methods to retrieve the associated Business Card or to call findService() to obtain a Service Connection instance.

- **public static Entry getBusinessCardEntry(ServiceAdvertisement) throws ServiceException**

Given the ServiceAdvertisement reference, this method creates a ServiceConnection instance. The ServiceConnection's connect() method can then be used to obtain an instance of the Service associated with the ServiceAdvertisement argument.

- **public static ServiceConnection findService(String aliasName) throws ServiceNotFoundException**

Given the system alias name, this method locates the associated ServiceAdvertisement reference and creates a ServiceConnection instance.

Applications use the Service Connection Class to connect to an instance of an advertised device driver or system service. The following methods are provided:

- **public void final connect(Object parentCallback) throws ServiceNotFoundException**
Creates an instance of the service that implemented the advertised interface specified in the ServiceAdvertisement that is associated with this ServiceConnection instance.
- **public final ServiceInstance getService()**
Returns the ServiceInstance created by a previous connect call.
- **public void final disconnect() throws Service Exception**
Deletes the instance created on the connect call.

Creating a manifest file for the JCT

A manifest file is packaged with a JAR file and describes its contents. The MANIFEST.JCT file is used by the JCT to determine a bean location in the JSD.

MANIFEST.JCT is located in the META-JCT directory and has the following format:

```

PackageName=<product name>
PackageType=<type>
PackageAbout=<descriptive text>
PackageHelp=<descriptive text>
JSDEntryName=<entry path name>
JSDStoreFormat={ENUMERATE, BEAN, HASH}
JSDEntryNameFactory=class/method
ResourceBundle=<Bundle name>
META-Property=<name value>
Name=<bean.class>
DisplayName=<item name>
JSDPropertyName=<key name>
JSDPropertyMap=<name value>

```

The following code shows the MANIFEST.JCT file. Note that this file contains the location of the configuration information in the JSD. This file is provided as a sample in the directory:

JSDK100\demo\SampleApplication\LoadBean\META-JCT\MANIFEST.JCT

```

/*
 * @(#) MANIFEST.JCT
 *
PackageType=SERVICE
JSDEntryName=/software/service/com.ibm.javaos/applicationsample
    /properties
JSDStoreFormat=ENUMERATE

ResourceBundle=mri/AppLoadSample

Name=AppLoadSample.class
DisplayName=Application Sample Load Bean

```

Package attributes: Following are the attributes that describe a service package:

PackageName: Specifies the package name and is used for display and selection purposes. If a ResourceBundle attribute is supplied, the resource bundle is used to retrieve the locale-specific package name. If PackageName is not specified, the default for the package name is the JAR file name.

PackageType: Specifies the classification of the software and is used for selection purposes. This field is required and can be any of the following: SERVICE, APPLICATION, or SYSTEM.

PackageAbout: Specifies the versioning, licensing, and author information that is displayed for the JAR file when the system administrator requests specific information about the software. This field is optional. If nothing is supplied, this field defaults to the PackageName. If a ResourceBundle is supplied, the resource bundle is used to retrieve the locale-specific "About" text.

PackageHelp: Specifies the help information that is available when configuring any of the beans. This field is optional. If a ResourceBundle is supplied, then it will be used to retrieve the locale-specific help text.

JSDEntryName: Specifies the relative location of the configuration information. This is a required field that represents the path of the beans in the JSD. The Network Computer client uses the path to obtain specific configuration information for the JAR file, for example: /software/service/com.ibm/touchscreen.

JSDStore Format: Specifies the actual storage format of the configuration information being placed in the location JSDEntryName. The storage format can be any of the following: ENUMERATE, BEAN, or HASH.

If ENUMERATE is chosen, the data from each bean's get methods is stored in the JSD, with the accessor pair name used as the property name.

If BEAN is chosen, each bean is wholly stored in the JSD with a property name of JSDPropertyName.

If HASH is chosen, data from each bean's get methods is stored in a hash table, and then the hash table is stored in the JSD with a property name of JSDPropertyName. Any cross-bean accessor-pair name collisions can be resolved using the JSDPropertyMap field.

Note: The JSDStoreFormat field represents one of three different storage techniques that is used for the entire package, not for each bean. This field defaults to ENUMERATE if an option is not specified. Beans that use the ENUMERATE option should use built-ins or primitive Java types as their return values to ensure serialization of the objects. In addition, beans that use primitive types and a storage format of ENUMERATE have their properties stored as the primitives object counterpart; that is, int=integer.

JSDEntryNameFactory: Specifies the method for dynamic JSD entry name generation, which provides the capability for multiple instances of a single JAR file within the same tree. This optional field is used when it is necessary to construct a dynamic JSD entry name before storing the configuration object in the JSD. If a JSDEntryNameFactory is specified, the JSDEntryName is built by appending JSDEntryName with the result from calling the specified class-method. The call result must be a string object.

ResourceBundle: Specifies the bundle name used to localize PackageName, PackageAbout, and DisplayName. If the ResourceBundle name is specified, and the appropriate resource bundle is found, the appropriate attributes will be localized. If the ResourceBundle name is not found, the default are the strings provided. If the ResourceBundle is specified and a default basename is supplied, it is not necessary to provide the PackageName, PackageAbout or DisplayName in the MANIFEST.JCT. It can be supplied in the properties file.

META-Property: Specifies additional properties for a JSD entry. The META-Property is used by the system to give JSD entries special characteristics. These properties can be defined by the system or by a JAR File and are placed in the JSD entry. System-defined properties begin with a period ".". The ".JAR" system property and the ".Origin" system property are managed by the JCT and cannot be defined by the user. In addition, a META-property is never seen by the system administrator. They are handled and used by the JCT and optionally by the client.

Bean and applet attributes: Following are the attributes that describe a bean or applet:

Name: Specifies the name of the bean (or applet) to which the attributes that follow the name apply. Whenever the name attribute appears, each attribute following it is applied to the bean's class file (or the applet's HTML file). The name attribute must match the Jar Entry name of the file. This field is optional.

DisplayName: Displays information about the bean or applet and is used when building a user interface to provide a text representation of the bean's usage. The display name is optional and defaults to the package name. If a ResourceBundle is specified, it is used to retrieve the locale-specific display name.

JSDPropertyName: Specifies the property name to be used as the key for the BEAN or HASH JSDStoreFormat. If the storage format is BEAN or HASH, the JCT stores the bean data using this property name as the key. The client uses the property name to retrieve the configuration object from the JSD. This attribute has no meaning for the ENUMERATE storage format. This field is optional and defaults to the bean class name if BEAN or HASH is the storage format.

JSDPropertyMap Maps bean property names to JSD property names. This attribute allows you to assign the same property name to several beans in a jar and then map them to different JSD entry property names. This field is optional.

Making the load JAR file available to the JCT

The JCT displays the available JAR files (Java Beans). Install the load JAR file in the /JARS directory for the JCT to display it in the available list.

Adding the loadable service to the application

A service, in this case an application, must provide support to the JavaOS Service Loader, so that the loader can perform static initialization and subsequent instance creation and deletion of the service (application).

The following code fragment shows how the service class provides the start, stop, suspend, and resume methods for the application sample program provided in the directory:

JSDK100\demo\SampleApplication\Application\ApplicationService.Java

```
import javax.system.services.*;
import javax.system.database.*;

public class ApplicationService extends Service {

    public static void start(Entry bc) throws ServiceException {
        // DEBUG:: System.err.println("ApplicationService::start, entry="+bc);
    }

    public static void stop() {
    }

    public static void suspend() throws ServiceException {}
    public static void resume(Entry e) throws ServiceException {}
}
```

To support static initialization by the JavaOS Service Loader, the application has to implement the Abstract Service Class that is used by the JavaOS Service Loader to perform one-time startup and teardown of the service.

The *bundleInitClassName* parameter in the service's BusinessCard entry identifies the class name that implements the Abstract Service Class. Following are descriptions of its methods:

public static void start(Entry businessCard) throws ServiceException Called by the ServiceLoader as part of the processing for the loadService() method. The entry passed in is the Service's BusinessCard entry from the JSD namespace.

public static void stop() throws ServiceException Called by the JavaOS Service Loader as part of bytecode unloading. This method allows the service to do termination processing before its bytecodes are unloaded from the system.

public static void suspend() throws ServiceException Called by the Service Manager to notify the bundle to suspend its services. It is up to the creator of the service to perform appropriate processing.

public static void resume(Entry) throws ServiceException Called by the Service Manager to notify the service to resume. It is up to the creator of the service to perform appropriate processing.

To provide instance creation and deletion capabilities, each service must include a class that implements the *ServiceInterface* interface. The *bundleInstanceClassName* parameter of the *BusinessCard* contains the name of the class that provides this implementation. The interface is used by the JavaOS Service Loader to obtain and release *ServiceInstance* instances. Following are its methods:

public static ServiceInstance createInstance(String logicalName, Object parentCallback, Entry matchingEntry, Entry aliasEntry, Object cookie) throws ServiceException The provided sample:

jdk100\demo\SampleApplication\Application\ApplicationInstance.java

shows the use of this method. If this method call is made as a result of matching a service marked *loadsWhenMatchedToEntry* to a device, the matching entry argument contains a reference to the entry that was matched; otherwise, it is null. If this method call is made as the result of a *ServiceLoader.findService(String aliasName)* call, the *aliasEntry* argument contains a reference to the corresponding alias JSD entry; otherwise, it is null. The *logicalName* argument identifies the *logicalName* within the *BusinessCard* that the instance should be returned for. The *cookie* argument is provided on both the *createInstance()* and *deleteInstance()* calls to maintain the logical relationship of the instance.

public void deleteInstance(Object cookie) throws ServiceException Tells the service that the instance returned on the previous *createInstance()* call is no longer needed.

Chapter 7. Customizing an application

Application customization is possible only when you have access to the application source code.

Although it is optional, storing configuration information in the JSD integrates the application into JavaOS for Business and allows you to change the application properties through the JavaOS Configuration Tool (JCT).

To store the application configuration information in the JSD, create a configuration JAR file. This file contains the application properties such as background colors, window size, etc.

A sample configuration JAR file is provided in the directory:

JSDK100\demo\SampleApplication\ConfigBean\META-JCT

You also need to create a manifest file for the JCT, the MANIFEST.JCT file, to specify the name and the location of the configuration JAR file in the JSD. For more information see “Creating a manifest file for the JCT” on page 22 .

The following code shows the format of the MANIFEST.JCT file. Note that this file contains the location of the configuration information in the JSD. This program is provided as a sample in the directory:

JSDK100\demo\SampleApplication\ConfigBean\META-JCT

```
# MANIFEST.JCT
PackageType=APPLICATION
JSDEntryName=/software/application/com.ibm/application/properties
JSDStoreFormat=ENUMERATE

ResourceBundle=mri/Application

Name=FirstFile.class
DisplayName=Application Bean
```

Appendix A. JavaOS Service Loader

The JavaOS Service Loader (JSL) is the control point for managing, loading, and instantiation of JavaOS for Business services such as applications, device drivers, and system services. When the Service Loader receives a request from the Service Connection classes to connect to an instance of an advertised service, it takes the steps necessary to load and instantiate the service. When a disconnect is requested, the service instance is dereferenced and its cached bytecodes become eligible for garbage collection.

The JavaOS Service Loader supports the following components:

- Service Connection classes
- File loader
- Unbundle interface
- Service configuration classes

Service Connection Classes provide the public methods used by applications to create and delete instances of an advertised interface.

File Loader provides logical file system services. Upon request from the JavaOS Service Loader, this component retrieves and unbundles a file. The unbundled data and classes are placed in a logical file system, where they can be accessed using normal JavaOS for Business file system operations.

The *Unbundle* interface provides unbundling services to the File Loader. An unbundler that supports ZIP and JAR is provided. Other unbundle types can also be dynamically loaded and can advertise their services. The File Loader provides a logical file system hash table that is populated by the Unbundler with the individual class and data files that make up the service package.

BusinessCard is the set of configuration parameters provided by a service (application). These parameters are contained in the JSD /Software/Services namespace in nodes uniquely identified by a META-property.

Appendix B. JavaOS Configuration Tool

The JavaOS Configuration Tool (JCT) provides a framework to systems, device drivers, and applications for defining information to the configuration server so that the the system administrator can use it for configuration purposes. The information includes properties that can be managed and modified by the system administrator, assigned to machines or users, and then downloaded to the appropriate network computers. The JCT runs locally, using the file:// protocol, in the process of the JSD server, and in the context of a Java 1.1-enabled Web browser that allows local applets unrestricted access to local resources.

The configuration information must be provided in a standard Java JAR (Java ARchive) file. The JAR file includes a manifest file, which describes the contents of the JAR file. The configuration information can take one of two forms: a set of Java Beans that contain the properties to be included in the JSD entry or an HTML/Java applet combination. If Java Beans are provided, the JCT uses the beans to create a dynamic user interface through Java Introspection and Property Editors. Introspection enables the retrieval of each property value from each Java Bean. If an HTML/Java applet is provided, it will be launched. The applet must provide the user interface and also insert the proper JSD entries into the JSD, using the JSD interfaces provided on the server.

Appendix C. JavaOS System Database

The JavaOS System Database (JSD) is organized as a hierarchy of entries, which descends from a single super root entry into namespaces entries. A namespace is a specially designated tree of entries that name like kinds of objects. Namespaces are always direct descendants of the super root. Each namespace is managed by a namespace manager. System database initialization includes the creation of the following JSD namespaces, in the order shown, which are used by the JavaOS Service Loader to load and track.

Dynamically Loading Services:

- Temp
- Interface
- Alias
- Software
- Device

The order of namespace creation is enforced because some namespaces initialized later must create nodes in the namespaces created earlier. The portions of the Temp, Interface, Alias, and Software namespaces that are described in the following sections contain information that is produced by the JavaOS for Business framework. This information is used by the Device Namespace manager to instantiate each driver.

Temp namespace

A `/Temp/BusinessCardTracking` entry is created for each business card entry in the Software namespace. Each `BusinessCardTracking` entry contains a `ServiceTracking` object instance. The `ServiceTracking` object contains the non-persistent data associated with its business card. This information, which is used by the JavaOS Service Loader, includes data on whether the bytecodes comprising the service are loaded and what the active connections are.

Interface namespace

The Interface namespace contains entries that publish interfaces being advertised by a service, as identified in its business card. When a new business card entry is added, an entry in the `/Interface` namespace for each interface listed in the `bundleInstanceInfo` parameter is created. Applications and other JavaOS for Business components use the `/Interface` namespace to discover and connect to instances of a service that implements the advertised interface they want to use.

Alias namespace

An Alias namespace entry is set up for each alias name listed in the `bundleInstanceInfo` parameter of the Business Card. Each alias name is associated with an `/Interface` entry. Applications and other JavaOS for Business components can use `/Alias` entries to request an instance of the service that has implemented that interface. For example, the assignment of a Network computer `/DESKTOP` would appear in this namespace.

Software namespace

The Software namespace manager sets up the Software namespace branches. It creates the subtree containing the Business Card entries. Each entry under the /Software/Service branch contains the persistent configuration parameters that comprise the Business Card for a service package.

Device namespace

During its initialization, the Device Namespace manager first populates the namespace with physical bus and device information gathered through the JBI APIs. The minimum information will be an entry representing the processor device that corresponds to the particular platform.

After entries have been created for the discovered devices, the Device Namespace manager attempts to locate and assign a device driver to each device. Matching of device to device driver is done using the information in the Business Cards. The Device Namespace manager uses the JavaOS Service Loader *loadServices(Entry BusinessCardTracking)* method to lead the bytecodes into the client's file system and to perform a one-time driver initialization. As each driver is loaded it is possible for additional entries in the device namespace to be created. The device namespace is traversed until no new entries are created.

After driver loading is completed, the conflict detection and arbitration phase begins. During this phase, the configured system resources in the business card and the information in the Device namespace entries must be used to assign conflict-free system resources.

After resources are assigned, the Device namespace manager instantiates each driver and updates the ServiceTracking instance in the BusinessCardTracking entry to reflect the driver's new state.

Appendix D. Building the sample application

To aid application developers with integrating their application into JavaOS for Business, the JSDK includes a sample program with its configuration JAR file and load JAR file. The application uses the JSD to set and retrieve application data, and is configurable using the JCT. The JSDK samples come with batch files that can be used to build samples.

The sample application consists of 3 pieces:

- The application itself
- The configuration JAR file
- The load JAR file

You can build each sample using the build.bat batch file found in the **JSDK100\demo\SampleApplication** directory.

To use this file, go into the **jskd100\demo\SampleApplication** directory and type **build** (no arguments).

Note: You must have already set the CLASSPATH. See “Configuring the JSDK” on page 6 for more information.

Appendix E. Building device driver samples

JSDK provides the following samples for device drivers:

- PC3Com–Ethernet driver for the 3ComFast Etherlink XL PCI 3C905

To build the PC3Com device driver, go to the **JSDK100demo\PC3Com\sun\javaos\PC3Com** directory and type **build PC3Com**.

- NfsFileSystem–Network File System service

The NfsFileSystem sample consists of 2 pieces:

- Loadable service
- Configuration JAR file

To build the NfsFileSystem loadable service, go to the **JSDK100\demo\NfsFileSystem\sun\javaos\NfsFileSystem** directory and type **build NfsFileSystem**.

To build the configuration JAR file, go to the **JSDK100\demo\Config\com\ibm\joscfg\NfsFileSystemcfg** directory and type **build NfsFileSystemcfg**.

Note: An NFS connection between JavaOS for Business Network computers and Windows NT server hosts is not supported by this sample code.

- EEPro100 – Ethernet driver for the Intel™ EtherExpress PRO 100+

The EEPro100 sample consists of 2 pieces:

- Device driver
- Configuration JAR file

To build the EEPro100 device driver, go to the **JSDK100\demo\EEPro100\sun\javaos\EEPro100** directory and type **build EEPro100**.

To build the configuration JAR file, go into the **JSDK100\demo\Config\com\ibm\joscfg\EEPro100cfg** directory and type **build EEPro100cfg**.

- Parallel1284 – Parallel port driver

The Parallel1284 sample consists of 2 pieces:

- Device driver
- Configuration JAR file

To build the Parallel1284 device driver, go to the **JSDK100\demo\Parallel1284\sun\javaos\Parallel1284** directory and type **build Parallel1284**.

To build the configuration JAR file, go to the **JSDK100\demo\Config\com\ibm\joscfg\Parallel1284cfg** directory and type **build Parallel1284cfg**.

- PCITr – IBM PCI token-ring adapter

The PCITr sample consists of 2 pieces:

- Device driver
- Configuration JAR file

To build the PCITr device driver, go to the **JSDK100\demo\PCITr\sun\javaos\PCITr** directory and type **build PCITr**.

To build the configuration bean, go to the **JSDK100\demo\Config\com\ibm\joscfg\PCITrcfg** directory and type **build PCITrcfg**.

Note: This code is provided for illustrative purposes only. This release of JavaOS for Business does not support client machines with token-ring adapters.

- Smart card

The smart card sample consists of card layout files and device drivers.

The following card layout file are included in the directory:

JSDK100\demo\SmartCard

- joslogon.clt – Smart card layout description
- m3tfixdc.inc – Declarations and environment specification for Multi Formation Card (MFC) 3.51
- m4tfixdc.inc – Declarations and environment specification for Multi Formation Card (MFC) 4.0

The following smart card device drivers samples are provided in the directory:

JSDK100\demo\SmartCard

- sc8002 – Smart card driver
- iso7816 – Base smart card services

Note: The smart card device drivers are not currently supported on the Intel platform. They are provided as an illustration of a smart card driver implementation.

You can obtain the IBM Smart Card ToolKit at <http://www.chipcard.ibm.com/sc12tkit.htm>.

The OpenCard APIs can be obtained at <http://www.opencard.org/OCF/1.0/nc>

For more information about integrated circuit cards, see ISO7816 – “Integrated Circuit Cards with Contacts” (part 1-7.)

Index

A

- adding the loadable service
 - JavaOS Service Loader 24
- applet attributes 23
- application
 - packaging 13

B

- bean attributes 23
- bldlevel 8
- business card
 - /Software/Service 34
 - /Temp/BusinessCardTracking 33
 - bundleInstanceInfo 33

C

- Class
 - Service Connection 21
 - Service Loader 21
- CLASSPATH 6, 7
- configuration JAR file 27
- Creating a Business Card 19
- customizing an application
 - configuration JAR file 27

D

- device namespace 34

I

- ICAT debugger 5, 7
- implementing a load JAR file
 - Business Card 18
 - Dynamic Loading Service 19
 - JavaOS Configuration Tool 18
 - JavaOS Service Loader 18
 - JavaOS System Database 18
 - MANIFEST.JCT 18

J

- Java Development Kit 5
- JavaOS Configuration Tool 31
- JavaOS Service Loader 29
 - loadServices(Entry BusinessCardTracking) 34
- JavaOS System Database (JSD) 33
- JSD
 - alias namespace 33
 - interface namespace 33

- JSD namespace
 - temp 33
- JSDK
 - API 8
 - overview 3

M

- MANIFEST.JCT 19
 - creating 22
 - format 22
 - package attributes 22

N

- namespace 33
 - device 34
 - Ssoftware 34

P

- packaging
 - JavaOS Configuration Tool 14
 - JavaOS Service Loader 14
 - JavaOS System Database 14

S

- Service Connection Class 21, 29
- Service Loader Class 21
- software namespace 34
- storing information in the JSD
 - configuration JAR file 27
 - MANIFEST.JCT 27

T

- tool
 - bldlevel 8
 - ICAT debugger 5, 7
 - JAR 7
 - JDK 5

U

- unbundle interface 29

