

Java Card™ 2.1 Development Kit User's Guide



Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

Version 1.0P
November 15, 1999

Copyright © 1999 Sun Microsystems, Inc.

901 San Antonio Road, Palo Alto, CA 94303 USA

All rights reserved. Copyright in this document is owned by Sun Microsystems, Inc.

Sun Microsystems, Inc. (SUN) hereby grants to you at no charge a nonexclusive, nontransferable, worldwide, limited license (without the right to sublicense) under SUN's intellectual property rights that are essential to practice the Java Card™ technology to use this document for internal evaluation purposes only. Other than this limited license, you acquire no right, title, or interest in or to the document and you shall have no right to use the document for productive or commercial use.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government 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 227.7202-1(a).

SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.

TRADEMARKS

Sun, Sun Microsystems, the Sun logo, Java, Java Card, SunDocs, and SunExpress are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. UNIX® is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

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 vii

Who Should Use This Book vii

Before You Read This Book vii

How This Book Is Organized viii

Related Books viii

Ordering Sun Documents ix

What Typographic Changes Mean ix

1. Introduction to the Java Card™ 2.1 Development Kit 1

2. Installation 3

Installing the Java Card™ 2.1 Development Kit 3

 Setting up Your Environment 4

 Installed Directory Structure 5

Obtaining javax.comm 7

3. How to Use this Release 9

The Java Card Demonstration Applets 9

Running the Sample Applets 9

Building the Sample Applets 10

4. Using the Converter	13
Java Compiler Options	13
File and Directory Naming Conventions	14
Input Files	14
Output Files	14
Loading Export Files	15
Specifying an Export Map	16
Running the Converter	16
Command Line Arguments	17
Command Line Options	17
5. Using capgen	21
Command line for capgen	21
6. Using the JCWDE	23
Preliminaries	23
Running the JCWDE Tool	24
7. Using the Installer	25
Overview	25
Installer Applet AID	25
How to Use the Installer	25
Installer APDU protocol	26
Installer Error Response APDUs	27
Installer Limitations	27
8. Using the APDUTool	29
A. JCA Syntax Example	31

Figures

FIGURE 4-1 Calls between packages go through the export files 15

Preface

Java Card™ technology combines a subset of the Java™ programming language with a runtime environment optimized for smart cards and similar kinds of small-memory embedded devices. The goal of Java Card technology is to bring many of the benefits of Java software programming to the resource-constrained world of smart cards.

The Java Card API is compatible with international standards, such as ISO7816, and industry-specific standards, such as Europay/Master Card/Visa (EMV).

The *Java Card™ 2.1 Development Kit User's Guide* contains information on how to install and use the tools comprising this release.

Who Should Use This Book

The *Java Card™ 2.1 Development Kit User's Guide* is targeted at developers who are creating applets using the *Java Card™ 2.1 API Specification, Revision 1.1*, Sun Microsystems, Inc.

Before You Read This Book

Before reading this guide, you should be familiar with the Java programming language, object-oriented design, the Java Card technology specifications, and smart card technology. A good resource for becoming familiar with Java technology and Java Card technology is the Sun Microsystems, Inc. Web site, located at:
<http://java.sun.com>.

How This Book Is Organized

Chapter 2, “Installation,” describes how to install the tools included in this release.

Chapter 3, “How to Use this Release,” explains the intended use of the tools and sample applets in this release, by means of presenting the sample applets in the Java Card Demonstration.

Chapter 4, “Using the Converter,” provides an overview of the Converter and details of how to run it.

Chapter 5, “Using capgen,” describes how to use the capgen utility.

Chapter 6, “Using the JCWDE,” describes how to use the JCWDE applet simulator.

Chapter 7, “Using the Installer,” describes how to create applets using the installer.

Chapter 8, “Using the APDUTool,” describes using this tool to send APDUs to the JCWDE.

Appendix A, “JCA Syntax Example,” describes the JCA output of the Converter using a commented example file.

Related Books

References to various documents or products are made in this manual. You should have the following documents available:

- *Java Card™ 2.1 API Specification, Revision 1.1*, Sun Microsystems, Inc.
- *Java Card™ 2.1 Virtual Machine Specification, Revision 1.1*, Sun Microsystems, Inc.
- *Java Card™ 2.1 Runtime Environment (JCRE) Specification, Revision 1.1*, Sun Microsystems, Inc.
- *The Java Language Specification* by James Gosling, Bill Joy, and Guy L. Steele. Addison-Wesley, 1996, ISBN 0-201-63451-1.
- *The Java Virtual Machine Specification (Java Series)* by Tim Lindholm and Frank Yellin. Addison-Wesley, 1996, ISBN 0-201-63452-X
- *The Java Class Libraries: An Annotated Reference (Java Series)* by Patrick Chan and Rosanna Lee. Addison-Wesley, ISBN: 0201634589
- *ISO 7816 Specification* Parts 1-6

Ordering Sun Documents

The SunDocsSM program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpressTM Internet site at <http://www.sun.com/sunexpress>.

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%</code> You have mail.
<AaBbCc123>	Command-line placeholder: replace with a real name or value	To delete a file, type <code>rm <filename></code> .
<i>AaBbCc123</i>	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.

Introduction to the Java Card™ 2.1 Development Kit

Java Card™ 2.1 Development Kit is for smart cards and similar kinds of small-memory embedded devices. This release of the is intended to give you practice in taking Java sources (applets you write yourself or the provided samples), and running them in the JCWDE. (JCWDE stands for Java Card™ Workstation Development Environment, a simulator and related tools for Java Card applet development.)

You can also run Java sources through the `converter` tool provided with this release to verify that the applet uses only the supported subset of the Java programming language. As an additional purpose, the `converter` tool outputs a JCA file containing a representation of the applet. Finally, you can input the JCA file into the `capgen` tool to convert the applet to a `cap` file conforming to the *Java Card™ 2.1 Virtual Machine Specification, Revision 1.1*, Sun Microsystems, Inc.

Installation

This release is provided for the Microsoft Windows NT 4.0 platform as a Zip archive, and for the UNIX® platform as a compressed tar archive.

Note – Be sure not to install this release into the same directory as a prior release.

Note – UNIX Users — The command line examples in this guide are written with the assumption that directory names in paths are separated with back slashes (\) in the style of Microsoft Windows. You, as a user of UNIX, will of course substitute forward slashes (/) in paths. Similarly, an executable file for Windows users will be shown with an .exe filename extension. No such extension is necessary for a user of UNIX. Batch scripts are referred in Microsoft Windows and shell scripts in UNIX.

The command lines shown here assume the C-shell (csh). For other shells, modify the command lines appropriately.

Installing the Java Card™ 2.1 Development Kit

NT Users

Unzip the file `java_card_kit2_1-win.zip`. Use your favorite unzip utility to unpack the file.

UNIX Users

Uncompress and untar the file `java_card_kit2_1-unix.tar.Z` with the following command line:

```
uncompress -c java_card_kit2_1-unix.tar.Z | tar xvf -
```

All Users

This creates a directory `jc21` (and its subdirectories) in whatever was your current location in your file system when unzipping or uncompressing.

Setting up Your Environment

The JC21BIN Environment Variable

Set the environment variable `JC21BIN` to the `jc21\bin` directory. This environment variable specifies the directory that the command scripts use to locate the jar files. For example, if you unzipped the release at the root of the C volume on an NT system, or your home directory (for example, the user name `doe`) on a UNIX system:

NT Users

```
set JC21BIN=c:\jc21\bin
```

UNIX Users

```
setenv JC21BIN /home/doe/jc21/bin
```

The PATH Environment Variable

Next, add the `JC21BIN` directory to your `PATH` environment variable:

NT Users

```
set PATH=%PATH%;%JC21BIN%
```

UNIX Users

```
setenv PATH ${PATH}:%JC21BIN
```

The CLASSPATH

The batch and shell scripts provided with the Version 1.0P assume that you have a CLASSPATH environment variable specifying the location of `classes.zip` or `rt.jar`, the Java API core class files. (For details, refer to your Java IDE or JDK installation instructions.) As supplied, the batch and shell script files append appropriate JAR files to the CLASSPATH when they are run.

Note – When compiling Java files or executing the JCWDE, ensure that the `classes.zip` file entry (in JDK version 1.0 or 1.1) or the `rt.jar` file entry (in JDK version 1.2) precedes the `javacard api jar` file (`api21.jar`) entry in the CLASSPATH.

Java VM Considerations

For the examples listed in this guide, you will need to be able to run the Java VM from the command line of your workstation.

The batch and shell scripts as provided with the Version 1.0P assume that the Java executable is in your PATH. To see whether it is, type `java -help`. If this does not return the Java Runtime Loader help message, you will need to add the directory where `java.exe` can be found to your PATH environment variable.

Installed Directory Structure

The result of unpacking is a directory `jc21` with the following subdirectories.

`api21`

This directory contains the following APIs:

`java.lang`

This package contains the source files for the Java Card language API. The classes in this package are fundamental to the design of the Java Card technology subset of the Java programming language.

`javacard.framework`

This package provides source files for the framework of classes and interfaces for the core functionality of a Java Card applet.

`javacard.security`

This package provides the source files for the classes and interfaces for the Java Card security framework.

`javacardx.crypto`

This extension package contains security classes and interfaces for export-controlled functionality.

The sources for the APIs provided with this release are for information only, and are not to be modified or compiled.

Note – this release does not include the `javacardx.crypto` API package.

`bin`

This directory contains the JAR files and batch or shell scripts for the APDU Tool, JCWDE, and the tools.

`demo`

This directory contains only one demonstration.

`demo1`

This demo contains the installer, the `JavaPurse`, `JavaLoyalty` and `Wallet` sample applets.

`doc`

This directory contains the following two documents in pdf format:

Java Card 2.1 Development Kit User's Guide

(this document).

Java Card 2.1 Development Kit Release Notes

The Java Card 2.1 Development Kit Release Notes provides new changes and updated information for this release of the Java Card 2.1 Development Kit.

`samples`

This directory contains the following sample applets: `HelloWorld`, `JavaLoyalty`, `JavaPurse`, `NullApp` and `Wallet` sample applets.

Obtaining javax.comm

The Java Communications API 2.0 contains a package, `javax.comm`, which is needed to run the Java Card™ 2.1 Development Kit. Please visit Sun's World Wide Web site at <http://java.sun.com/products/javacomm/index.html> to obtain the package. Follow the instructions there to install the package and correctly set up your environment to use it.

How to Use this Release

This release provides you demonstration applets and the tools to verify and simulate their execution. In practicing using the tools with the demonstration applets, you can also learn how to verify and simulate Java Card applets you might write.

The Java Card Demonstration Applets

The `demo1` demonstration contains the installer, and the `JavaPurse`, `JavaLoyalty` and `Wallet` applets as part of the masked JCWDE image. `demo1` runs in the JCWDE simulator.

Running the Sample Applets

To practice running the JCWDE and the APDUTool, use the sample files provided in this release. The `demo` directory contains three files `jcwde.app`, `demo1.scr`, and `demo1.scr.expected.out`.

Open two command windows. Make sure that `JC21BIN` environment variable is set to the `bin` subdirectory of the root of JC21 installation. Also verify that `java` is in the command path, and `CLASSPATH` is set up correctly. Then change to the `demo` directory in both windows. In one window, enter the following command to start JCWDE:

```
jcwde -p 9025 jcwde.app
```

(For more information on the JCWDE, refer to Chapter 6, “Using the JCWDE.”)

The `jcwde.app` file is a config file containing Java Card applet installation information. It lists all the sample applets provided in the Version 1.0P. The JCWDE responds with:

```
JavaCard 2.1 Workstation Development Environment (version 1.0).  
Listening for T=0 Adu's on TCP/IP port 9,025.
```

In the other window, start the APDU Tool:

```
apdutool demol.scr > demol.scr.out
```

The `demol.scr` file is an APDU script which contains the command APDUs to be sent to the JCWDE. The APDUTool creates the `demol.scr.out` file containing APDU Tool output.

Upon completion of the `demol.scr` execution, the message

```
jcwde exiting on receipt of power down command
```

is printed by JCWDE. The newly created `demol.scr.out` file should be the same as the `demol.scr.expected.out` file provided with this release.

Building the Sample Applets

Although the sample applets are provided pre-built to run in the JCWDE, you may want to rebuild them to learn how to build your own. The following commands and responses show how to compile the NullApp sample application, how to invoke the converter tool to verify that the applet uses the correct subset of the Java programming language, and how to create a CAP file using the `capgen` tool. You can follow similar steps for the other sample applets or applets you write yourself. Before entering the following commands, change to the `samples` directory.

```
> javac -classpath ../bin/api21.jar com\sun\javacard\samples\NullApp\*.java  
  
> converter -config com\sun\javacard\samples\NullApp\NullApp.opt  
  
Java Card 2.1 Class File Converter (version 1.0)  
Copyright (c) 1999 Sun Microsystems, Inc. All rights reserved.  
conversion completed with 0 errors and 0 warnings.  
  
> capgen -o NullApp.cap com\sun\javacard\samples\NullApp\javacard\NullApp.jca  
  
Java Card 2.1 CAP File Builder (version 1.0)  
Copyright (c) 1999 Sun Microsystems, Inc. All rights reserved.
```

Note – This is how the commands and responses will look on the Microsoft Windows NT platform. On the Solaris platform, backslashes '\ ' would be replaced by forward slashes '/').

Using the Converter

The converter tool is the front end of the conversion process. There is a separate backend called `capgen`. (Refer to Chapter 5, “Using `capgen`.”)

Usually, you instruct the Converter at the command line to output a `JCA` (Java Card Assembly) file, which you then input to `capgen` to produce a `CAP` file. A `JCA` file is a human-readable ASCII file to aid testing and debugging. The converter can also produce an `export` file. (Refer to “Loading Export Files” on page 15.)

Java Compiler Options

The `class` files should be compiled with `-g` option of the JDK Java compiler command line. Don't use the `-O` option.

This is because the Converter determines the local variable types by checking the `LocalVariableTable` attribute within the class file. This attribute is generated in the class file only if the `-g` option is used at the Java compiler command line.

The `-O` option is not recommended at the Java compiler command line, for two reasons. This option is intended to optimize execution speed rather than minimize memory usage. The latter is much more important in Java Card technology. Also, if the `-O` option is used, the `LocalVariableTable` attribute won't be generated even if the `-g` option is used.

File and Directory Naming Conventions

This section details the names of input and output files for the Converter, and gives the correct location for these files. With some exceptions, the Converter follows the Java naming conventions for default directories for input and output files. These naming conventions are also in accordance with the definitions in § 4.1 of the *Java Card™ 2.1 Virtual Machine Specification, Revision 1.1*, Sun Microsystems, Inc.

Input Files

The files input to the Converter are Java class files named with the `.class` suffix. Generally, there are several class files making up a package. All the class files for a package must be located in the same directory under the root directory, following the Java naming conventions. The root directory can be set from the command line using the `-classdir` option. If this option is not specified, the root directory defaults to be the directory from which the user invoked the Converter.

Suppose, for example, you wish to convert the package `java.lang`. If you use the `-classdir` flag to specify the root directory as `C:\mywork`, the command line will be:

```
converter -classdir C:\mywork java.lang package_aid package_version
```

The converter will look for all class files in the `java.lang` package in the directory `C:\mywork\java\lang`

Output Files

The name of the `export` file and the `JCA` file must be the last portion of the package specification followed by the extensions `.exp` and `.jca` respectively.

By default, the files output from the Converter are written to a directory called `javacard`, a subdirectory of the input package's directory.

In the above example, the output files are written by default to the directory `C:\mywork\java\lang\javacard`

The `-d` flag allows you to specify a different root directory for output.

In the above example, if you use the `-d` flag to specify the root directory for output to be `C:\myoutput`, the Converter will write the output files to the directory `C:\myoutput\java\lang\javacard`.

Loading Export Files

A Java Card export file contains the public API linking information (public classes, public and protected methods and fields) of classes in an entire package. The Unicode string names of classes, methods and fields are assigned unique numeric tokens.

Export files are not used directly on a device that implements a Java Card virtual machine. However, the information in an export file is critical to the operation of the virtual machine on a device. An export file is produced by the Converter when a package is converted. This package's export file can be used later to convert another package that imports classes from the first package. Information in the export file is included in the CAP file of the second package, then is used on the device to link the contents of the second package to items imported from the first package.

During the conversion, when the code in the currently converted package references a different package, the Converter loads the export file of the different package.

The following figure illustrates how an applet package is linked with the `java.lang`, the `javacard.framework` and `javacard.security` packages via their export files.

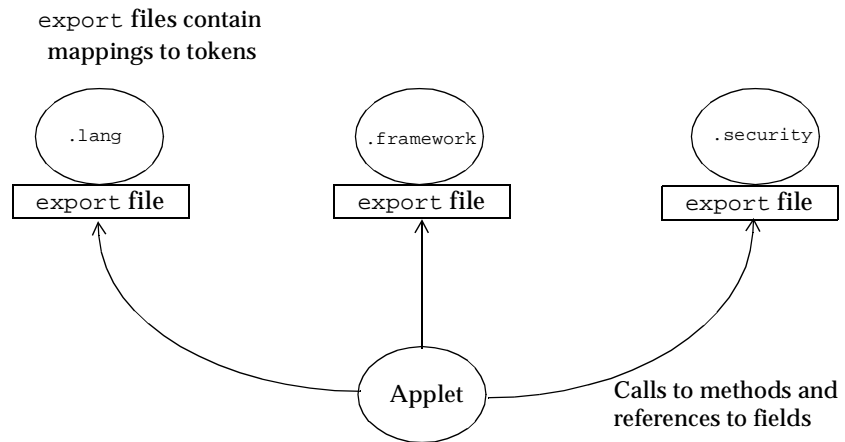


FIGURE 4-1 Calls between packages go through the export files

You can use the `-exportpath` command option to specify the locations of export files. The path consists of a list of root directories in which the Converter looks for export files. Export files must be named as the last portion of the package name followed by the extension `.exp`. Export files are located in a subdirectory called `javacard`, following the Java Card directory naming convention.

For example, to load the export file of the package `java.lang`, if you have specified `-exportpath` as `c:\myexportfiles`, the Converter searches the directory `c:\myexportfiles\java\lang\javacard` for the export file `lang.exp`.

Specifying an Export Map

You can request the Converter to convert a package using the tokens in the pre-defined export file of the package that is being converted. Use the `-exportmap` command option to this.

Note – This command option exists for licensed users who might modify one of the Java Card packages. The current release does not permit any such modification of source code. The documentation of this command option is provided for information only.

For example, if a licensed user re-implemented the `javacard.framework` package, he or she must convert the package using the export file `framework.exp` provided by Sun. By specifying the `-exportmap` option, he or she instructs the Converter to convert your implementation of the `javacard.framework` package according to the tokens defined in `framework.exp`.

The converter loads the pre-defined export file of the currently-converted package the same way it loads other export files.

Running the Converter

Command line usage of the Converter is:

```
converter [options] package_name package_aid
                                major_version.minor_version
```

The file to invoke the Converter is a shell script (`converter`) on the UNIX[®] platform, and a batch file (`converter.bat`) on the Microsoft Windows NT platform.

Command Line Arguments

The arguments to this command line are:

`package_name`

the fully-qualified name of the package to convert.

`package_aid`

5 to 16 decimal, hex or octal numbers separated by colons. Each of the numbers must be byte-length.

`major_version.minor_version`

user-defined version of the package.

Command Line Options

The options in this command line are:

`-classdir` <the root directory of the class hierarchy>

Set the root directory where the Converter will look for classes.

If this option is not specified, the Converter uses the current user directory as the root.

`-i`

Instruct the Converter to support the 32-bit integer type

`-exportpath` <List of directories>

These are the root directories in which the Converter will look for export files. The separator character for multiple paths is platform dependent. It is semicolon (;) for the Microsoft Windows NT platform and colon (:) for the UNIX[®] platform.

If this option is not specified, the Converter sets the `exportpath` to the Java `classpath`.

`-exportmap`

Use the token mapping from the pre-defined `export` file of the package being converted. The converter will look for the `export` file in the `exportpath`.

`-applet <AID> <class_name>`

Set the default applet AID and the class that defines the install method for the applet.

If the package contains multiple applet classes, this option must be specified for each class.

`-d <the root directory for output>`

Set the root directory for output

`-out [JCA] [EXP]`

Tell the Converter to output the JCA file and/or the export file.

By default (if this option is not specified), the Converter outputs a JCA file and an export file.

`-V, -version`

Print the Converter version string

`-v, -verbose`

Enable verbose output

`-mask`

Indicates this package is for mask, so restrictions on native methods are relaxed

`-help`

Print out help message

`-nowarn`

Instruct the Converter to not report warning messages

`-nobanner`

Suppress all messages to standard output

Command Configuration File

You could also include all the command line arguments and options in a configuration file. The syntax to specify a configuration file is:

`converter -config <configuration file name>`

The `<configuration file name>` argument contains the file path and file name of the configuration file.

Using capgen

`capgen` is a backend to the Converter. It produces a CAP file from a JCA file.

Command line for capgen

The file to invoke `capgen` is a shell script (`capgen`) on the UNIX platform, and a batch file (`capgen.bat`) on the Microsoft Windows NT platform.

Command line usage of `capgen` is:

```
capgen [-options] <infile>
```

The flag `-o` allows you to specify an output file. If the output file is not specified with the `-o` flag, output defaults to the file `a.jar` in the current directory.

The flag `-version` outputs the version information.

The flag `-help` displays online documentation for this command.

The flag `-nobanner` suppresses all messages to standard output.

Using the JCWDE

The JCWDE tool kit allows the simulated running of a Java Card applet as if it were masked in ROM. It emulates the card environment.

The JCWDE tool kit executable consists of the `jcwde.jar`, `api21.jar`, and `apduio.jar` files. Also provided are the sample applets in the `samples.jar` file. The main class for JCWDE is `com.sun.javacard.jcwde.Main`.

A sample batch and shell script are provided to start JCWDE.

Preliminaries

Make sure that the `CLASSPATH` and `JC21BIN` environment variables are set, as detailed in “Setting up Your Environment” on page 4.

Configuring the Applets in the JCWDE Mask

The applets to be configured in the mask during JCWDE simulation need to be listed in a configuration file that is passed to the JCWDE as a command line argument. In this release, the sample applets are listed in a configuration file called `jcwde.app`. Each entry in this file contains the name of the applet class, and its associated AID.

The configuration file contains one line per installed applet. Each line is a white space(s) separated `{CLASS NAME, AID}` pair, where `CLASS NAME` is the fully qualified Java name of the class defining the applet, and `AID` is an Application Identifier for the applet class used to uniquely identify the applet. `AID` may be a string or hexadecimal representation in form `0xXX[:0xXX]`¹. Note that `AID` should be 5 to 16 bytes in length.

1. Repeat the construct `:0xXX` as many times as necessary.

For example:

```
com.sun.javacard.samples.wallet.Wallet 0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6:0x1
```

Note – The installer applet must be listed first in the JCWDE configuration file.

If you write your own applets for public distribution, you should obtain an AID for each of your packages and applets according to the directions in §4.2 of the *Java Card™ 2.1 Virtual Machine Specification, Revision 1.1*, Sun Microsystems, Inc., and in *ISO 7816 Specification* Parts 1-6.

Running the JCWDE Tool

The general format of the command to run the JCWDE is as follows:

```
jcwde [-p port] [-version] [-nobanner] <config-file>
```

where:

- the flag `-p` allows you to specify a TCP/IP port other than the default port;

- the flag `-version` specifies prints the JCWDE version number;

- the flag `-nobanner` suppresses all messages to standard output; and

- `<config-file>` is the configuration file described above.

When started, JCWDE starts listening to APDUs in T=0 format on the TCP/IP port specified by the `-p` port parameter. The default port is 9025.

Using the Installer

Overview

The Java Card installer's role in the JCWDE simulation is to create instances of the applets previously configured in the JCWDE mask file. (See "Configuring the Applets in the JCWDE Mask" on page 23.)

The APDU command sequence for creation is shown below in "Create Only" on page 26.

For more information about the installer, please see the *Java Card™ 2.1 Runtime Environment (JCRE) Specification, Revision 1.1*, Sun Microsystems, Inc.

Installer Applet AID

The on-card installer applet AID is: 0xa0, 0x0, 0x0, 0x0, 0x62, 0x3, 0x1, 0x8, 0x1

How to Use the Installer

The installer is invoked using the APDUtool. (See Chapter 8, "Using the APDUTool.")

Applet creation is the only scenario supported by the installer in the JCWDE mode:

Create Only

In this scenario, the applet from the set configured in the mask is instantiated. (Refer to “Configuring the Applets in the JCWDE Mask” on page 23). Steps to perform this creation of the JavaPurse applet are:

1. Determine the applet AID.
2. Create an APDU script similar to this:

```
powerup;
// Select the installer applet
0x00 0xA4 0x04 0x00 0x09 0xA0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01
0x7F;
// begin installer command
0x80 0xB0 0x00 0x00 0x00 0x7F;
// create JavaPurse
0x80 0xB8 0x00 0x00 0x0b 0x09 0xA0 0x00 0x00 0x00 0x62 0x03 0x01 0x04
0x01 0x00
0x7F;
// end installer command
0x80 0xB1 0x00 0x00 0x00 0x7F;
// APDU commands to select and test your applet go here
powerdown;
```

3. Invoke APDUTool with this APDU script file path as the argument.

Installer APDU protocol

Select APDU to invoke the on-card installer:

Table 1:

00, 0xA4, 04, 00	Lc field	Installer AID	Le field
------------------	----------	---------------	----------

Response APDU from the on-card installer:

Table 2:

[optional response data]	SW1SW2
--------------------------	--------

Create Applet:

Table 3:

0x80, 0xb8, 0x00, 0x00	Lc field	AID length field	AID	parameter length field	[parameters]	Le field
------------------------	----------	------------------	-----	------------------------	--------------	----------

Installer Error Response APDUs

```
/**
 * Response status : Installer in error state = 0x6021
 */
static final short ERROR_STATE = 0x6021;

/**
 * Response status : Exception occurred = 0x6024
 */
static final short ERROR_EXCEPTION = 0x6024;

/**
 * Response status : Applet not found = 0x6043
 */
static final short ERROR_APPLET_NOT_FOUND = 0x6043;

/**
 * Response status : Applet creation failed = 0x6044
 */
static final short ERROR_APPLET_CREATION = 0x6044;
```

Installer Limitations

- The maximum length of the parameter in applet creation APDU command is 14.
- The maximum number of applets which can be configured is 16 minus the number of ROM applets.
- The maximum length of data in the installer APDU commands is 32.

Using the APDUTool

The APDUTool reads a script file containing command APDUs and sends them to the JCWDE. Each command APDU (C-APDU) is processed by the JCWDE and returned to the APDUTool, which displays both the command and response APDUs on the console. Optionally, the APDUTool can write this information to a log file.

The file to invoke the APDUTool is a shell script (`apdutool`) on the UNIX platform, and a batch file (`apdutool.bat`) on the Microsoft Windows NT platform. For example:

```
apdutool example.scr
```

This command runs the APDUTool with the file `example.scr` as input. Output goes to the console.

```
apdutool -o example.scr.out example.scr
```

This command runs the APDUTool with the file `example.scr` as input. Output is written to the file `example.scr.out`.

The APDU script file is a protocol-independent APDU format containing comments, script file commands, and C-APDUs. Script file commands and C-APDUs are terminated with a `;`. Comments may be of any of the three Java style comment formats (`//`, `/*` or `/**`)

APDUs are represented by decimal, hex or octal digits, UTF-8 quoted literals or UTF-8 quoted strings. C-APDUs may extend across multiple lines.

C-APDU syntax for the APDUTool is as follows:

```
<CLA> <INS> <P1> P2> <LC> [<byte 0> <byte 1> ... <byte LC-1>] <LE> ;
```

where

<CLA> :: ISO 7816-4 class byte.

<INS> :: ISO 7816-4 instruction byte.

<P1> :: ISO 7816-4 P1 parameter byte.

<P2> :: ISO 7816-4 P2 parameter byte.
<LC> :: ISO 7816-4 input byte count.
<byte 0> ... <byte LC-1> :: input data bytes.
<LE> :: ISO 7816-4 expected output length byte. 0 implies 256.

The following script file commands are supported:

powerUp;

Send a power up command to the reader. A powerUp command must be executed prior to sending any C-APDUs to the reader.

powerDown;

Send a power down command to the reader.

echo "string";

Echo the quoted string to the output file. The leading and trailing quote characters are removed.

delay <Integer>;

Pause execution of the script for the number of milliseconds specified by <Integer>.

JCA Syntax Example

This appendix contains an annotated JCA file output from the Converter. The comments in this file are intended to aid the developer in understanding the syntax of the JCA language, and as a guide for debugging Converter output.

```
/*+
 * Copyright (c) 1999 Sun Microsystems, Inc. All rights reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information"). You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE
 * SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
 * PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES
 * SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING
 * THIS SOFTWARE OR ITS DERIVATIVES.
-*/

/*
 * JCA (Java Card Assembly) annotated example. The code contained within this example
 * is not an executable program. The intention of this program is to illustrate the
 * syntax and use of the JCA directives and commands.
 *
 * A JCA file is textual representation of the contents of a CAP file. The contents
 * of a JCA file are heirachly structured. The format of this structure is:
 *
 * package
 *     package directives
 *     imports block
 *     applet declarations
 *     constant pool
 *     class
 *         field declarations
```

```

*         virtual method tables
*         methods
*         method directives
*         method statements
*
* JCA files support both the Java single line comments and Java block comments.
* Anything contained within a comment is ignored.
*
* Numbers may be specified using the standard Java notation. Numbers prefixed
* with a 0x are interpreted as
* base-16, numbers prefixed with a 0 are base-8, otherwise numbers are interpreted
* as base-10.
*
*/

/*
* A package is declared with the .package directive. Only one package is allowed
* inside a JCA
* file. All directives (.package, .class, et.al) are case insensitive. Package,
* class, field and
* method names are case sensitive. For example, the .package directive may be written
* as .PACKAGE,
* however the package names example and ExAmPle are different.
*/
.package example {

    /*
    * There are only two package directives. The .aid and .version directives declare
    * the aid and version that appear in the Header Component of the CAP file.
    * These directives are required.

    .aid 0:1:2:3:4:5:6:7:8:9:0xa:0xb:0xc:0xd:0xe:0xf; // the AIDs length must be
                                                    // between 5 and 16 bytes inclusive
    .version 0.1;                                // major version <DOT> minor version

    /*
    * The imports block declares all of packages that this package imports. The data
    * that is declared
    * in this section appears in the Import Component of the CAP file. The ordering
    * of the entries
    * within this block define the package tokens which must be used within this
    * package. The imports
    * block is optional, but all packages except for java/lang import at least
    * java/lang. There should
    * be only one imports block within a package.
    */

    .imports {
        0xa0:0x00:0x00:0x00:0x00:0x62:0x00:0x01 1.0;
        // java/lang aid <SPACE>  java/lang major version <DOT> java/lang minor version
    }
}

```

```

    0:1:2:3:4:5 0.1;           // package test2
    1:1:2:3:4:5 0.1;           // package test3
    2:1:2:3:4:5 0.1;           // package test4
}

/*
 * The applet block declares all of the applets within this package. The data
 * declared within this block appears
 * in the Applet Component of the CAP file. This section may be omitted if this
 * package declares no applets. There
 * should be only one applet block within a package.
 */

.applet {
    6:4:3:2:1:0 test1; // the class name of a class within this package which
    7:4:3:2:1:0 test2; // contains the method install([BSB)V
    8:4:3:2:1:0 test3;
}

/*
 * The constant pool block declares all of the constant pool's entries in the
 * Constant Pool Component. The positional
 * ordering of the entries within the constant pool block define the constant pool
 * indices used within this package.
 * There should be only one constant pool block within a package.
 *
 * There are six types of constant pool entries. Each of these entries directly
 * corresponds to the constant pool
 * entries as defined in the Constant Pool Component.
 *
 * The commented numbers which follow each line are the constant pool indexes
 * which will be used within this package.
 */

.constantPool {

    /*
     * The first six entries declare constant pool entries that are contained in
     * other packages.
     * Note that superMethodRef are always declared internal entry.
     */
    classRef      0.0;      // 0 package token 0, class token 0
    instanceFieldRef 1.0.2; // 1 package token 1, class token 0,
                          // instance field token 2
    virtualMethodRef 2.0.2; // 2 package token 2, class token 0,
                          // instance field token 2
    classRef      0.3;      // 3 package token 0, class token 3
    staticFieldRef 1.0.4;    // 4 package token 1, class token 0,
                          // field token 4
    staticMethodRef 2.0.5;  // 5 package token 2, class token 0,

```

```

//      method token 5

/*
 * The next five entries declare constant pool entries relative to this class.
 */
classRef      test0;                // 6
instanceFieldRef test1/field1;      // 7
virtualMethodRef test1/method1()V; // 8
superMethodRef test9/equals(Ljava/lang/Object;)Z; // 9
staticFieldRef test1/field0;        // 10
staticMethodRef test1/method3()V;   // 11
}

/*
 * The class directive declares a class within the Class Component of a CAP file.
 * All classes except java/lang/Object should extend an internal or external
 * class. There can be
 * zero or more class entries defined within a package.
 *
 * for classes which extend a external class, the grammar is:
 * .class modifiers* class_name class_token? extends packageToken.ClassToken
 *
 * for classes which extned a class within this package, the grammar is:
 * .class modifiers* class_name class_token? extends className
 *
 * The modifiers which are allowed are defined by the Java Card language subset.
 * The class token is required for public and protected classes, and should not be
 * present for other classes.
 */

.class final public test1 0 extends 0.0 {

    /*
     * The fields directive declares the fields within this class. There should
     * be only one fields
     * block per class.
     */

    .fields {
        public static int field0 0;
        public int field1 0;
    }

    /*
     * The public method table declares the virtual methods within this classes
     * public virtual method
     * table. The number following the directive is the method table base (See the
     * Class Component specification).
     */

```

```

    * Method names in declared in this table are relative to this class. This
    * directive is required even if there
    * are not virtual methods in this class. This is necessary to establish the
    * method table base.
    */

    .publicmethodtable 1 {
        equals(Ljava/lang/Object;)Z;
        method1()V;
        method2()V;
    }

    /*
    * The package method table declares the virtual methods within this classes
    * package virtual method
    * table. The format of this table is identical to the public method table.
    */

    .packagemethodtable 0 {}

    .method public method1()V 1 { return; }
    .method public method2()V 2 { return; }
    .method protected static native method3()V 0 { }
    .method public static install([BSB)V 1 { return; }
}

.class final public test9 9 extends test1 {

    .publicmethodtable 0 {
        equals(Ljava/lang/Object;)Z;
        method1()V;
        method2()V;
    }
    .packagemethodtable 0 {}

    .method public equals(Ljava/lang/Object;)Z 0 {
        invokespecial 9;
        return;
    }
}

.class final public test0 1 extends 0.0 {

    .Fields {
        // access_flag, type, name [token [static Initializer]] ;
        public static byte field0 4 = 10;
        public static byte[] field1 0;
        public static boolean field2 1;
        public short field4 2;
        public int field3 0;
    }
}

```

```

}
.PublicMethodTable 1 {
    equals(Ljava/lang/Object;)Z;
    abc()V;                // method must be in this class
    def()V;
    labelTest()V;
    instructions()V;
}
.PackageMethodTable 0 {
    ghi()V;                // method must be in this class
    jkl()V;
}
// if the class implements more than one interface, multiple
// interfaceInfoTables will be present.
.InterfaceInfoTable 0.0 {
    0;                    // index in public method table of method
    1;                    // index in public method table of method
}
.InterfaceInfoTable 0.0 {
    1;                    // index in public method table of method
}

/*
 * Declaration of 2 public visible virtual methods and two package visible
 * virtual methods..
 */
.method public abc()V 1 {
    return;
}
.method public def()V 2 {
    return;
}
.method ghi()V 0x80 {      // per the CAP file specification, method tokens
                          // for package visible methods
    return;                // must have the most significant bit set to 1.
}
.method jkl()V 0x81 {
    return;
}

/*
 * This method illustrates local lables and exception table entries. Lables
 * are locall to each
 * method. No restrictions are placed on label names except that they must
 * begin with an alphabetic
 * character. Label names are case insensitive.
 *
 * Two method directives are supported, .stack and .locals. These
 * directives are used to

```

```

* create the method header for each method. If a method directive is omitted,
* the value 0 will be used.
*
*/

.method public static install([BSB)V 0 {
    .stack 0;
    .locals 0;
10:    nop;
11:    nop;
12:    nop;
13:    nop;
14:    nop;
15:    nop;
    return;

    /*
     * Each method may optionally declare an exception table. The start offset,
     * end offset and handler offset
     * may be specified numerically, or with a label. The format of this table
     * is different from the exception
     * tables contained within a CAP file. In a CAP file, there is no end
     * offset, instead the length from the
     * starting offset is specified. In the JCA file an end offset is specified
     * to allow editing of the
     * instruction stream without having to recalculate the exception table
     * lengths manually.
     */

    .exceptionTable {
        // start_offset end_offset handler_offset catch_type_index;
        10 14 15 3;
        11 13 15 3;
    }
}

/*
 * Labels can be used to specify the target of a branch as well.
 * Here, forward and backward branches are
 * illustrated.
 */

.method public labelTest()V 3 {
L1:    goto L2;
        nop;
        nop;
L2:    goto L1;
        nop;
        nop;

```

```

        goto_w L1;
        nop;
        nop;
        goto_w L3;
        nop;
        nop;
        nop;
L3:      return;
    }

    /*
    * This method illustrates the use of each Java Card 2.1 instruction. Mnemonics
    * are case insensitive.
    * See the Java Card Virtual Machine Specification for the specification of
    * each instruction.
    */

    .method public instructions()V 4 {

        aaload;
        aastore;
        aconst_null;
        aload 0;
        aload_0;
        aload_1;
        aload_2;
        aload_3;
        anewarray 0;
        areturn;
        arraylength;
        astore 0;
        astore_0;
        astore_1;
        astore_2;
        astore_3;
        athrow;
        baload;
        bastore;
        bipush 0;
        bspush 0;
        checkcast 10 0;
        checkcast 11 0;
        checkcast 12 0;
        checkcast 13 0;
        checkcast 14 0;
        dup2;
        dup;
        dup_x 0x11;
        getfield_a 1;
        getfield_a_this 1;
    }

```



```

getfield_a_w 1;
getfield_b 1;
getfield_b_this 1;
getfield_b_w 1;
getfield_i 1;
getfield_i_this 1;
getfield_i_w 1;
getfield_s 1;
getfield_s_this 1;
getfield_s_w 1;
getstatic_a 4;
getstatic_b 4;
getstatic_i 4;
getstatic_s 4;
goto 0;
goto_w 0;
i2b;
i2s;
iadd;
iaload;
iand;
iastore;
icmp;
iconst_0;
iconst_1;
iconst_2;
iconst_3;
iconst_4;
iconst_5;
iconst_m1;
idiv;
if_acmpeq 0;
if_acmpeq_w 0;
if_acmpne 0;
if_acmpne_w 0;
if_scmpeq 0;
if_scmpeq_w 0;
if_scmpge 0;
if_scmpge_w 0;
if_scmpgt 0;
if_scmpgt_w 0;
if_scmpne 0;
if_scmpne_w 0;
if_scmlt 0;
if_scmlt_w 0;
if_scmpne 0;
if_scmpne_w 0;
ifeq 0;
ifeq_w 0;
ifge 0;

```

```

ifge_w 0;
ifgt 0;
ifgt_w 0;
ifle 0;
ifle_w 0;
iflt 0;
iflt_w 0;
ifne 0;
ifne_w 0;
ifnonnull 0;
ifnonnull_w 0;
ifnull 0;
ifnull_w 0;
iinc 0 0;
iinc_w 0 0;
iipush 0;
iload 0;
iload_0;
iload_1;
iload_2;
iload_3;
ilookupswitch 0 1 0 0;
impdep1;
impdep2;
imul;
ineg;
instanceof 10 0;
instanceof 11 0;
instanceof 12 0;
instanceof 13 0;
instanceof 14 0;
invokeinterface 0 0 0;
invokespecial 3;// superMethodRef
invokespecial 5;// staticMethodRef
invokestatic 5;
invokevirtual 2;
ior;
irem;
ireturn;
ishl;
ishr;
istore 0;
istore_0;
istore_1;
istore_2;
istore_3;
isub;
itableswitch 0 0 1 0 0;
iushr;
ixor;

```

```

jsr 0;
new 0;
newarray 10;
newarray 11;
newarray 12;
newarray 13;
newarray boolean[]; // array types may be declared numerically or
newarray byte[]; // symbolically.
newarray short[];
newarray int[];
nop;
pop2;
pop;
putfield_a 1;
putfield_a_this 1;
putfield_a_w 1;
putfield_b 1;
putfield_b_this 1;
putfield_b_w 1;
putfield_i 1;
putfield_i_this 1;
putfield_i_w 1;
putfield_s 1;
putfield_s_this 1;
putfield_s_w 1;
putstatic_a 4;
putstatic_b 4;
putstatic_i 4;
putstatic_s 4;
ret 0;
return;
s2b;
s2i;
sadd;
saload;
sand;
sastore;
sconst_0;
sconst_1;
sconst_2;
sconst_3;
sconst_4;
sconst_5;
sconst_m1;
sdiv;
sinc 0 0;
sinc_w 0 0;
sipush 0;
sload 0;
sload_0;

```

```

        sload_1;
        sload_2;
        sload_3;
        slookupswitch 0 1 0 0;
        smul;
        sneg;
        sor;
        srem;
        sreturn;
        sshl;
        sshr;
        sspush 0;
        sstore 0;
        sstore_0;
        sstore_1;
        sstore_2;
        sstore_3;
        ssub;
        stableswitch 0 0 1 0 0;
        sushr;
        swap_x 0x11;
        sxor;
    }
}

.class public test2 2 extends 0.0 {

    .publicMethodTable 0 {}
        equals(Ljava/lang/Object;)Z;
    .packageMethodTable 0 {}
    .method public static install([BSB)V 0 {
        .stack 0;
        .locals 0;
        return;
    }
}

.class public test3 3 extends test2 {
    .publicMethodTable 0 {}
        equals(Ljava/lang/Object;)Z;
    .packageMethodTable 0 {}
    .method public static install([BSB)V 0 {
        .stack 0;
        .locals 0;
        return;
    }
}

.interface public test4 4 extends 0.0 {

```

```
}  
}
```