Development Kit User's Guide

Java Card™ Platform, Version 3.0.1
Connected Edition

5/21/09

Sun Microsystems, Inc.
www.sun.com

May 2009

Copyright © 2009 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at http://www.sun.com/patents and one or more additional patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights - Commercial Software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

Sun, Sun Microsystems, the Sun logo, Java, Solaris, Java Card, Java Developer Connection, Mozilla, Netscape, Javadoc, JAR, JDK, JVM, and NetBeans are trademarks or registered trademarks of Sun Microsystems, Inc. or its subsidiaries, in the U.S. and other countries

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

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2009 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, États-Unis. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plusieurs des brevets américains listés à l'adresse suivante: http://www.sun.com/patents et un ou plusieurs brevets supplémentaires ou les applications de brevet en attente aux États - Unis et dans les autres pays.

Droits du gouvernement des États-Unis - Logiciel Commercial. Les droits des utilisateur du gouvernement des États-Unis sont soumis aux termes de la licence standard Sun Microsystems et aux conditions appliquées de la FAR et de ces compléments.

Sun, Sun Microsystems, le logo Sun, Java, Solaris, Java Card, Java Developer Connection, Mozilla, Netscape, Javadoc, JAR, JDK, JVM, et NetBeans sont des marques de fabrique ou des marques déposées enregistrées de Sun Microsystems, Inc. ou ses filiales, aux États-Unis et dans d'autres pays.

UNIX est une marque déposée aux États-Unis et dans d'autres pays et licenciée exlusivement par X/Open Company, Ltd.

Les produits qui font l'objet de ce manuel d'entretien et les informations qu'il contient sont regis par la legislation americaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucleaires, des missiles, des armes biologiques et chimiques ou du nucleaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou reexportations vers des pays sous embargo des États-Unis, ou vers des entites figurant sur les listes d'exclusion d'exportation americaines, y compris, mais de maniere non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations de des produits ou des services qui sont regi par la legislation americaine sur le contrôle des exportations et la liste de ressortissants specifiquement designes, sont rigoureusement interdites..

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE ÀLA QUALITE MARCHANDE, À L'APTITUDE À UNE UTILISATION PARTICULIERE OU À L'ABSENCE DE CONTREFAÇON.

Please
Recycle

Contents

Figures

Tables

Preface

This document describes how to use the Java Card 3 Platform, Connected Edition, development kit to develop applications, servlets, and extended applets. The Connected Edition architecture uses a new virtual machine and a substantially different runtime environment from that of the classic platform (an update of the Java Card technology released in the 2.2.2 release).

Java Card technology for the Connected Edition targets devices that are less resource-constrained than previous Java Card technology devices. The Connected Edition includes new network-oriented features, such as support for web applications, including the Java Servlet APIs, and support for applets with extended and advanced capabilities.

Note – The Java Card 3 platform development kit is released in both binary and source bundles. Some bundles include cryptography extensions. Portions of this document are targeted toward specific release bundles and are identified as such throughout this book.

You must download the Java Card specifications bundle separately from the Sun Microsystems web site at:

`http://java.sun.com/javacard`

Apache Ant (Ant) tasks in the Development Kit are required to install and run the Development Kit.

Who Should Use This Document

The *Development Kit User's Guide, Java Card Platform, Version 3.0.1, Connected Edition* is written for developers who are:

- Creating Java Card 3 web or servlet applications, or classic or extended applet applications for the Connected Edition.
- Creating a vendor-specific framework based on the specifications for the Connected Edition.

Before You Read This Document

Before reading this guide, you should become familiar with the Java™ programming language, object-oriented programming, the specifications for the Connected Edition, and smart card technology. A good resource for becoming familiar with Java and Java Card technology is the Java Developer Connection™ web site located at `http://java.sun.com`.

How This Book Is Organized

The guide is divided into two parts. The Part I describes how to set up the development kit, how to use the samples, and how to use the development kit tools. Part II describes various programming issues for the Java Card 3 platform.

Part I: Setup, Samples and Tools

Chapter 1 provides an overview of the Development Kit for the Connected Edition.

Chapter 2 describes the procedures for installing the tools included in this release.

Chapter 3 provides a brief description of the steps involved in Java Card platform application development.

Chapter 4 describes the samples included with the Development Kit and provides the procedures used to run them.

Chapter 5 describes the reference implementation of the Connected Edition and provides the procedures used to start it.

Chapter 6 describes how to compile source files outside of an IDE by using the Compiler tool included with the Development Kit.

Chapter 7 describes how to use the Packager tool to create and validate a Java Card technology-based application module.

Chapter 8 describes how to use the Installer tool to perform card management tasks.

Chapter 9 describes how to use the the tools provided by the Development Kit to modify classic applets to run on the Java Card 3 platform.

Chapter 10 describes the APDU tool and how it is used when installing and running applets on a smart card.

Chapter 11 describes how to install and to use the Debugger tool in Java Card 3 platform applications development.

Part II: Programming With the Development Kit

Chapter 12 describes the options used to configure the RI.

Chapter 13 describes how developers can modify or add to source files of the RI including VM code, and all tools (such as the Packager and Installer) and build a cutomized Java Card 3 platform RI according to their specific requirements.

Chapter 14 describes how to use the Java Card RMI client-side API.

Chapter 15 describes the APDU I/O API, which is a library used by development kit components, such as `apdutool`, and the RMI client framework.

Chapter 16 describes how to generate and install SSL keys.

Appendix A describes the application module and library formats supported by the Java Card 3 platform card manager.

Appendix B lists and describes the files and directories installed as part of the bundle.

Appendix C is a reference of command line usage for the Development Kit tools.

Glossary describes key terms used in this document.

Related Documents

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

- *Application Programming Interface, Java Card Platform, Version 3.0.1,Connected Edition*
- *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition*
- *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Connected Edition*
- *Application Programming Interface, Java Card Platform, Version 3.0.1, Connected Edition*
- *Application Programming Notes, Java Card Platform, Version 3.0.1, Connected Edition*

- *ISO 7816 Specification* Parts 1-6
- *Java Card Platform, Version 3.0, White Paper*
- *Java Card Technology for Smart Cards: Architecture and Programmer's Guide* by Zhiqun Chen (Addison-Wesley, 2000)
- *Java Servlet Specification, Java Card Platform, Version 3.0.1, Connected Edition*
- *Off-Card Verifier,Java Card 2.2.2, White Paper*
- *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition*
- *The Java Programming Language (Java Series), Fourth Edition* by James Gosling, Ken Arnold, and David Holmes (Addison-Wesley, 2005)
- *The Java Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999)
- *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Connected Edition*

Specifications, Standards, Protocols and Technologies

The Connected Edition supports the following specifications, standards, protocols, and technologies:

- ETSI SCP and UICC specification for 3G mobile phones.
- ISO 7816-4:1995 Identification cards - Integrated circuit cards with contacts part 4, inter-industry commands for interchange.

  These specifications describe the communication transport and application protocol layer between the terminal and the card.
- ISO 7816-4:2004 Identification cards - Integrated circuit cards with contacts part 4, inter-industry commands for interchange.
- EMV 2000 Integrated Circuit Card specifications for payment systems.

  These standards enable the correct operation and interoperability of payment applications on terminals and smart cards.
- GlobalPlatform card specification

  These card specifications are built on top of the Java Card specifications to provide interoperable content and lifecycle management for multifunction payment cards.
- PCSC- Personal Computer Smart Card communication

  The standard communication interfaces used on personal computers to access smart card reader driver layers.

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
	Procedural steps	**1. Run `jcre` in a new window.**
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Accessing Documentation Online

The Java Developer Connection™ program web site enables you to access Java platform technical documentation on the web at

`http://java.sun.com/reference/docs`

Third-Party Web Sites

Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or

resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can submit your comments about this document to the following address:

`bandol-ri-feedback@sun.com`

Please include the following title of this document with your feedback:

Development Kit User's Guide, Java Card Platform, Version 3.0.1, Connected Edition

PART I Setup, Samples and Tools

This part of the user's guide describes how to install the development kit, use its tools and run its samples.

CHAPTER 1

Introduction

The Java Card Platform, Version 3.0.1 consists of two editions, the Classic Edition and the Connected Edition.

- The Classic Edition is based on an evolution of the Java Card Platform, Version 2.2.2 and is backward compatible with it, targeting resource-constrained devices that solely support applet-based applications. Applets that run on the Classic Edition are referred to as classic applets. The classic applets have the same capabilities as applets in previous versions of the development kit.
- The Connected Edition contains a new architecture that enables developers to integrate smart cards within IP networks and web services architectures. The Connected Edition supports extended applets and servlets to allow for these new capabilities. In addition, the Connected Edition also supports classic applets.

This document applies to the Connected Edition. References to components, such as the Java Card runtime environment (RE), refer to the component as it exists in the Connected Edition.

The Java Card development kit ships in binary-only bundles or bundles with both binary and source versions of the kit. In addition, cryptography extensions are available in some bundles. This document pertains to both binary and source bundles, except where noted.

This chapter contains the following sections:

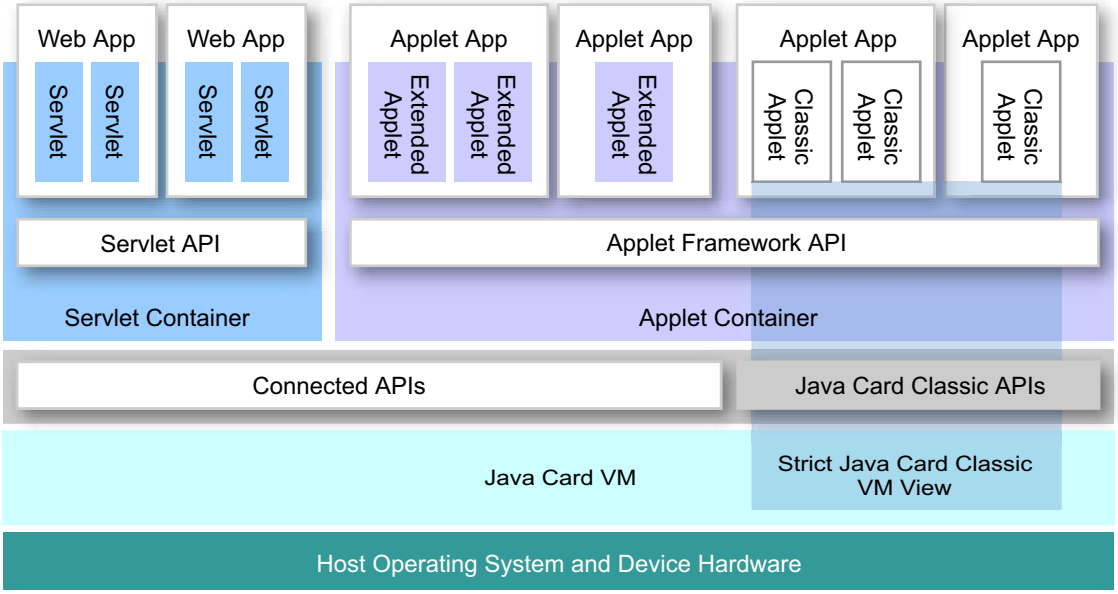
Platform Architecture

The Connected Edition contains a new architecture that enables developers to integrate smart cards within IP networks and web services architectures and features an enhanced runtime environment and virtual machine, with network-oriented features that support web applications. The Connected Edition supports both a web application model and an applet application model. The applet application model supports two types of applet applications - legacy applets and extended applets. Extended applets leverage the Connected Edition features while continuing to use the APDU communication model.

Java Card 3 platform, Connected Edition technology provides high-end smart cards with improved connectivity and integration into all-IP networks. A high-end, Java Card 3 technology-enabled smart card can act as a secure network node, capable of providing security services to the network or requesting access to network resources. It also allows for the convergence of smart-card services by handling multiple, concurrent communications through contact interfaces, using IP or ISO 7816-4 protocols, and through contactless interfaces, using the ISO 14443 protocol.

The high-level architecture of the Java Card 3 Platform, Connected Edition is illustrated in FIGURE 1-1. Notice the classic APIs in a Connected Edition are built on smart cards that implement a view of the strict, classic Java Card VM, which supports only classic applet applications. However, the Connected Edition Java Card VM also supports extended applets and servlets.

FIGURE 1-1 Architecture of Connected Edition



The development kit ships with a default Java Card RE that simulates a Java Card Platform, Connected Edition as it would be implemented onto a smart card. The default Java Card RE is the reference implementation (RI), and is invoked on the command line with `cjcre.exe`. The RI implements the ISO 7816-4:2005 specification, including support for up to twenty logical channels, as well as the extended APDU extensions as defined in ISO 7816-3. For more information on the RI, see Chapter 8.

The RI was designed to simulate a dual T=1 contacted and T=CL contactless concurrent interface implementation of the Java Card runtime environment, with the capability to operate on both interfaces simultaneously. The RI source code can be built and configured to support all the ISO 7816-3 and ISO 14443-4 smart card protocols, including T=0 single interface, T=1 single interface, T=CL single contactless interface and T=1/T=CL dual concurrent interface.

Development Kit Description

The Development Kit that enables creation of applications that utilize the Connected Edition new network-oriented features, such as support for web applications, including the Java™ Servlet APIs, as well as applets with extended and advanced capabilities. An application written for or an implementation of the Connected Edition may use features found in the Classic Edition.

Note – In this release, you will be able to use the Development Kit to create applications for both Classic and Connected Editions.

The Development Kit bundles include a suite of tools, a reference implementation, and the associated documentation for developers to use when developing Java Card technology-based applications (Java Card 3 platform applications), servlets, and extended applets for the Connected Edition. Developers use the Development Kit to create applications that fully utilize the features of the Connected Edition.

Connected Edition Features

Developers using the Development Kit to create implementations and applications for the Connected Edition should be aware of the following features of the Connected Edition that represent key security and usability characteristics of Java Card technology-based smart cards and ensure the backward-compatibility and scalability of the platform:

- Security for the Java Card 3 platform (Java Card security)
- Firewall mechanism
- Secure application update and upgrade
- Support for transactions, atomicity
- Card lifecycle-aware runtime environment
- Persistent memory model
- Standards alignment
- ISO 7816 compliance
- T=0, T=1, T=CL, USB, and MMC protocols support
- GP, ETSI/3GPP support
- Binary compatibility for Java Card 3 platform classic products
- Tools-automated application migration to Connected Edition products

- Legacy applications can be modified to use Connected Edition features
- Scalability
- Optional features optimize footprint
- Unified distribution file format
- TCK- enforced interoperability

Developers using the Development Kit to create applications for the Connected Edition should also be aware that the following features are exclusive to the Connected Edition:

- KVM-level VM technology
  - 32 bit VM
  - Dynamic `.class` file loading
  - Concurrent execution of applications
  - On-card and off-card bytecode verification
  - Automatic GC
- Network-oriented communication
  - Embedded web server
  - Service static and dynamic content through HTTP(s)
  - APDU and non-APDU comm support
  - Generic Communication API
  - Communication over USB, MMC
  - Management of concurrent contact/contactless card access
  - Client mode
- Connected Edition APIs
  - Support for additional Java language types `char` and `long`
  - String support
  - Multi-dimensional arrays
  - Object collections and large data structures
  - Generic event framework
  - Application code and data sharing enhancements

Connected Edition Security Model

The Connected Edition security model includes the following components and features:

- Class file verification

- Code isolation
- Context isolation (firewall)
- Policy-based access control
- Enhanced shareable interface mechanism
- Transport-level (SSL/TLS) web application security
- Web application client and card holder authentication
- Per-application declarative security
- Key and trust management

Application Models

The Connected Edition provides support for web applications, extended applets and legacy applet-based applications.

Web Applications

The Connected Edition provides support for typical web applications including servlets, filters, and listeners. The web application model is only available on implementations for the Connected Edition.

Extended Applets and Legacy Applet-Based Applications

For developers, the extended applet application model of the Connected Edition provides a migration path for legacy applet-based applications to the Connected Edition.

Development Kit Contents

The Development Kit is delivered in executable Java archive (JAR) files or bundles. Each bundle includes the binaries of a Java Card virtual machine, APDU tool, Compiler tool, Converter tool, Debugger tool, Installer tool, Normalizer tool, Packager tool, ROMizer tool, and sample applications for the Development Kit. In addition to the binaries, the source bundles also include the source files used to build the binaries.

Reference Implementation

The Connected Edition reference implementation is located in the `bin` directory with a program name of `cjcre.exe`. See Chapter 5 for detailed information about running the reference implementation from the command line.

Development KitDevelopment KitTools

Chapter 3 describes the sequence of development activities and the tool chain used in developing Java Card 3 applications.

The Development Kit bundle contains the following tools:

- **Compiler Tool**- Compiles Java Card 3 platform application source files.

  See Chapter 6 for information about using the Compiler tool.

- **Packager Tool** - Packages application modules and libraries into a deployable application group.

  See Chapter 7 for information about using the Packager tool.

- **Installer Tool** - Interacts with the on-card card manager to install applications and applets.

  See Chapter 8 for information about using the Installer tool as a stand-alone application.

- **APDU Tool** - When loading an applet, reads a script file containing Application Protocol Data Unit (APDU) commands and sends them to the C Java Card Runtime Environment where each APDU command is processed and returned to `apdutool`, which displays both the command and response APDU commands on the console as a stand-alone application.

  See Chapter 10 for information about using the APDU tool.

- **Normalizer Tool** - Generates application modules for a Java Card 3 platform smart card from a converted applet format.

  See Chapter 9 for information about using the Normalizer tool.

- **Converter Tool** - Converts Java class files into a format that can be loaded onto and run on a Java Card 3 platform smart card.

  See Chapter 9 for information about using the Converter tool.

- **Debugger Tool** - Used during development of Java Card 3 platform applications to suspend the VM, step over source code, and inspect variables.

  See Chapter 11 for information about using the Debugger tool.

- **ROMizer Tool** - Creates a ROM image to use in building a custom `cjcre.exe`.

  See Chapter 13 for detailed information about creating a ROM image file and building a custom `cjcre.exe`.

Samples

The Development Kit bundle contains various samples to give an overview of Java Card 3 platform applications. These samples are organized to provide sample applications that demonstrate the features of the Connected Edition and sample source code that developers can use in creating custom applications. See Chapter 4 for a description of the contents of the samples directories and a description of how to run them.

System Requirements

This release of the Development Kit executes on the Microsoft Windows XP SP2 operating system with an IDE of the developer's choice.

Additional Software

The following additional software is required by the Development Kit. See Chapter 2 for download and installation information.

- **Apache ANT -** Apache Ant 1.6.5 or higher is required to run the samples from command line or to build the `cjcre.exe` from source code.
- **Firefox browser** - The trusted agent for running the RI.
- **Internet Explorer 7 browser** - Used as a remote client and not the trusted agent.
- **GCC compiler** - If you are using the source bundle, the Minimal GNU for Windows (MinGW) version 5.1.3 is required to build the `cjcre.exe` or tools from source code. If you are using a binary bundle, MinGW is not required.

Note – MinGW is not required to run or to develop applications.

- **Java Development Kit** - The commercial version of Java Development Kit (JDK™) version 6 update 10 or higher (JDK version 1.6) is required.
- **NetBeans IDE (optional)** - The NetBeans IDE 6.7 and the can be used to develop applications.

Java Card TCK

The Java Card Technology Compatibility Kit (Java Card TCK) is a portable, configurable automated test suite for verifying the compliance of your implementation with the Java Card specification. To be in compliance, an implementation of the Java Card 3 platform, Connected Edition specification must pass the Java Card TCK 3.0.1 tests as described in *Java Card Technology Compatibility Kit, Version 3.0.1 User's Guide*.

CHAPTER 2

Installation

This chapter describes the prerequisites you need to install on your system before you use the development kit, how to install the development kit, how to set system variables, and how to uninstall the development kit. You can run both a Classic and Connected development kit simultaneously.

Binary and source code development kits are available for the Microsoft Windows XP SP2 operating system. Source code bundles allow you to change the development kit's reference implementation, whereas the binary bundles allow you only to use the reference implementation.

Each development kit is provided in an executable JAR file bundle. See Chapter 1 for a description of this development kit bundle and Appendix B for a list of all the files installed by this development kit.

Note – The Java Card specifications are not included in the Development Kit bundle. The specifications must be downloaded separately.

Prerequisites to Installing the Development Kit

The following software must be installed before installing the Development Kit:

- **Apache ANT -** download and install Apache Ant version 1.6.5 or higher from `http://ant.apache.org`.
- **Firefox browser** - download the Firefox browser from `http://www.mozilla.com`.

- **GCC compiler** - download and install MinGW from `http://sourceforge.net/projects/mingw` and install it according to the instructions on the `http://www.mingw.org` web site.
- **Java Development Kit** - download the JDK software from `http://java.sun.com/javase/downloads` and install it according to the instructions on the web site.
- **NetBeans IDE (optional)** - download NetBeans IDE 6.7 from `http://www.netbeans.org/downloads` and install it according to the instructions on the web site.

Install and Setup the Development Kit

This section describes how to install and set up the development kit.

▼ Install the Development Kit

1. **Verify that the additional software required by the Development Kit is installed on the development system.**

   See "Prerequisites to Installing the Development Kit" on page 11 for the download location and installation instructions of the required additional software.

2. **Download an appropriate Development Kit JAR file to a directory of your choice.**

3. **Launch the Development Kit installer.**

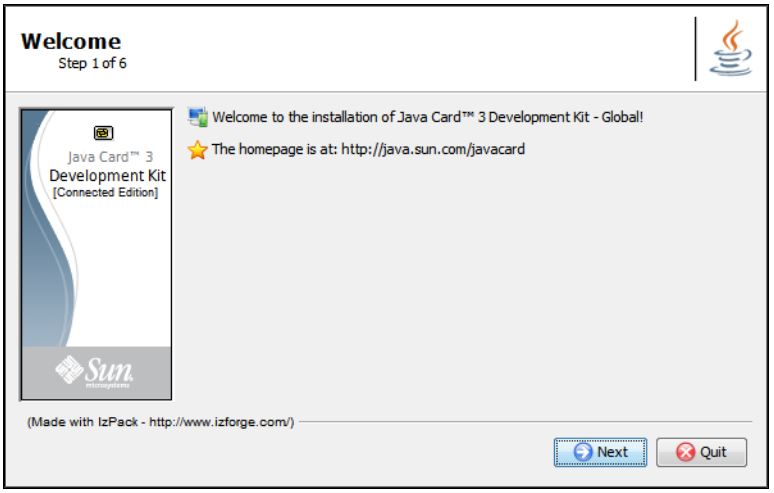
   The Development Kit can be launched automatically when you download the JAR file or by using the Windows file manager tool to navigate to the directory containing the Development Kit JAR file and double clicking the file name or icon.

   The Development Kit can also be launched by opening a Command Prompt window, navigating to the directory containing the Development Kit JAR file, and executing the following command from the command line:

   `java -jar` *Bundle-Filename*

   In the command, *Bundle-Filename* is the name of the downloaded Development Kit JAR file.

   The installation wizard displays the following screen.



4. **Complete each action requested by the installer.**

   By default, the Development Kit for the Connected Edition is installed in:

   `C:\JCDK3.0.1_ConnectedEdition`

   If you specify a different installation directory, the names of the installation directory and its parent must not contain a space.

   For example, the installation directory cannot be located in `C:\program files` because of the space in the `program files` directory name.

Note – The installation directory (either the default directory or the alternate installation directory you specify) is referred to as *JC_CONNECTED_HOME*.

---

5. **Click the Finish button to complete installation.**

   The bundle installs files and directories containing the binary files and source code described in Appendix B. The files and directories are installed under the root installation directory, either `C:\JCDK3.0.1_ConnectedEdition` or the directory you specified during installation. The root installation directory is referred to as *JC_CONNECTED_HOME* in this document.

▼ Setting Up the System Variables

1. **Set the JAVA_HOME system variable to the JDK root directory.**

   Before running the Development Kit, you must set the `JAVA_HOME` environment variable permanently in the Windows Control Panel or temporarily from the command line:

   - To permanently set `JAVA_HOME`, go to Windows Control Panel > System > Advanced > Environment Variables dialog and either create or edit a System variable named `JAVA_HOME` with the literal value of the JDK root directory on your system. For example, in the System variables box enter the following:

Variable	Value
`JAVA_HOME`	`C:\JAVA\jdk1.6.0_10`

   - To temporarily set `JAVA_HOME`, enter the following command in a Command Prompt window:

     `set PATH=C:\`*`java_home_path`*`;%PATH%`

     For example, if the Java platform software is stored in the `c:\jdk6` directory, enter:

     ```
     set PATH=C:\jdk6;%PATH%
     ```

Note – If using the Category view, choose Windows Control Panel > Performance and Maintenance > System > Advanced to open the Environment Variables panel.

---

2. **Set the ANT_HOME system variable to the Ant root directory.**

   Before running the Development Kit, you must set the `ANT_HOME` environment variable permanently in the Windows Control Panel or temporarily from the command line:

   - To permanently set ANT_HOME, go to Windows Control Panel > System > Advanced > Environment Variables dialog and either create or edit a System variable named ANT_HOME so that its value is the Apache Ant folder. For example, in the System variables box enter the following:

Variable	Value
`ANT_HOME`	`C:\ant\apache-ant-1.6.5`

   - To temporarily set ANT_HOME, enter the following command in a Command Prompt window:

     `set PATH=C:\`*`ANT_HOME`*`;%PATH%`

     For example if the Ant was installed in `C:\ant\apache-ant1.6.5`, enter:

     ```
     set PATH=C:\ant\apache-ant1.6.5;%PATH%
     ```

3. **Set the JC_CONNECTED_HOME system variable to the development kit root directory.**

   Before running the development kit, you must set the `JC_CONNECTED_HOME` environment variable permanently in the Windows Control Panel or temporarily from the command line:

Note – Some of the command line tools as well as running samples from the command line require that the `JC_CONNECTED_HOME` variable is set correctly.

   - To permanently set `JC_CONNECTED_HOME` , go to Windows Control Panel > System > Advanced > Environment Variables dialog and either create or edit a system variable named JC_CONNECTED_HOME variable so that its value is either `C:\JCDK3.0.1_ConnectedEdition` or the directory you specified during installation. For example, in the System variables box enter the following:

Variable	Value
`JC_CONNECTED_HOME`	`C:\JCDK3.0.1_ConnectedEdition`

   - To temporarily set `JC_CONNECTED_HOME`, enter the following command in a Command Prompt window:

     `set PATH=C:\`*`JC_CONNECTED_HOME`*`;%PATH%`

     For example if you installed in `C:\JCDK3.0.1_ConnectedEdition`, enter:

     ```
     set PATH=C:\JCDK3.0.1_ConnectedEdition\bin;%PATH%
     ```

4. **Add `%JAVA_HOME%\bin, %JC_CONNECTED_HOME%,` and `%ANT_HOME%\bin` to the `Path` variable displayed in the Environment Variables panel.**

5. **Add MinGW to the `Path` variable.**

   MinGW is not required if only the Development Kit binary bundle is installed. If the Development Kit source bundle is installed, set the `MinGW` environment variable permanently in the Windows Control Panel or temporarily from the command line:

   - To permanently set the MinGW path, edit the `Path` variable in the System variables box to include the location of `MinGW\bin`:

     ```
     ;C:\MinGW\bin;
     ```

   - To temporarily set the MinGW path, enter the following command in a Command Prompt window:

     `set PATH=C:\`*`MinGW_path`*`;%PATH%`

     For example, if MinGW is installed in the `C:\mingw` directory, enter:

     ```
     set PATH=C:\mingw\bin;%PATH%
     ```

Note – If you choose to set the `JAVA_HOME` variable and MinGW `PATH` each time you run the Development Kit, place the appropriate `JAVA_HOME` variable and MinGW `PATH` commands in a batch file.

Uninstalling the Development Kit

To uninstall the Development Kit, delete the *JC_CONNECTED_HOME* directory and all of its contents.

CHAPTER 3

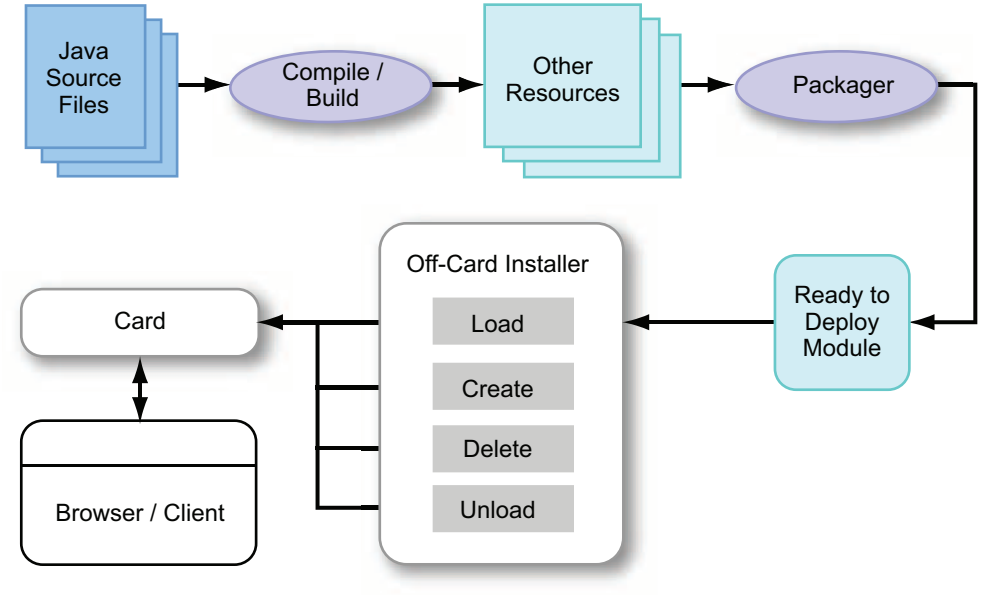
Developing Java Card 3 Platform Applications

This chapter provides a brief description of the activities and Development Kit tools involved in developing applications for the Java Card 3 platform. If you are enabling classic applets to run on Connected Edition and Classic Edition cards, see Chapter 9.

Development Steps

The steps described in FIGURE 3-1 illustrate the sequence of activities completed by a developer when creating an application for the Java Card 3 platform. See the *Application Programming Notes, Java Card Platform, Version 3.0.1, Connected Edition* for additional, advanced information not provided in this guide about creating applications for the Java Card 3 platform.

FIGURE 3-1 Java Card 3 Platform Application Development



1. **Source files** - Write the source code and create the descriptor files.

   The Development Kit also provides sample application source code that developers can use in creating custom applications. See Chapter 4 for a description of the samples provided in the Development Kit.

2. **Compile/build** - Compile the source code.

   See Chapter 6 for a description of using the Java Card 3 platform Compiler tool (`javacardc.bat`) as a stand-alone application.

3. **Packager** - Package the compiled source code.

   See Chapter 7 for a description of using the Packager tool to create and validate application modules.

4. **Off-Card Installer** - Load the application and create instances on the card by using the Installer tool.

   See Chapter 8 for a description of using the Off-Card Installer (Installer) tool and the associated on-card installer used to load an application module onto the card, create an instance of an application, delete (deactivate) an instance of an application, remove a module or application from the card, and display information about loaded applications and instances.

5. **Browser/Client** - Access the application on the card by using a client (browser or APDU tool).

   See Chapter 10 for description of using the APDU tool to display command and response APDU commands on the console.

CHAPTER 4

Running the Samples

The samples directory under *JC_CONNECTED_HOME* contains various samples that demonstrate the features of the Java Card API, Connected Edition. The samples include simple web applications, extended applet applications, classic applet applications, and reference applications. Reference applications are blue print-like applications that demonstrate the interactions between various applications on the card using advanced features such as SIO and events.

This chapter describes the procedures for running the samples and contains the following sections:

General Procedures for Running Samples

This section contains the following general procedures that developers can use to run a sample from the command line.

▼ Run Samples from the Command Line

Each sample has a `build.xml` at the root level of sample folder. This `build.xml` can be run from command line using the `ant` tool. To run any sample from command line perform the following steps:

1. **Open a command window.**
2. **Make sure ant can be run from command line.**

   See Chapter 2 for information about installing and running ant.
3. **Verify `JC_CONNECTED_HOME` is set to the Development Kit home.**

Note – `JC_CONNECTED_HOME` represents the directory in which the Development Kit was installed.

If `JC_CONNECTED_HOME` is not set permanently by using the Windows Control Panel (see Chapter 2), you can temporarily set `JC_CONNECTED_HOME` by entering the following command:

```
set JC_CONNECTED_HOME=c:\path
```

4. **Go to the appropriate sample folder.**

   This example uses the HelloWorld sample.
5. **Enter the `ant run` command.**

   The following figure illustrates the command window used to run the HelloWorld sample.

```
C:\JCDK3.0.1_ConnectedEdition\samples\web\HelloWorld>ant run
Buildfile: build.xml
```

The **`ant run`** command starts `cjcre.exe` in a new command window, loads the built application, and, if used, opens the browser to access the application.

When running, most sample applications open a browser and display a web page that serves as the primary user interface. Follow the instructions for each sample (explained in later sections) to interact with the application.

▼ Accepting an Untrusted Certificate

When running a sample that uses HTTPS to establish a secure connection with a web server, the Firefox browser might report that the sample uses an untrusted certificate and not allow you to accept the certificate required to open the web page. If `cjcre.exe` is still running, you can use the browser Certificate Manager to add an exception for the server certificate by performing the following procedure.

1. **In the browser menu bar select the Tools > Options menu items.**
2. **In the Options dialog box, select the Advanced icon in the tool bar and click the View Certificates button to open the Certificate Manager.**
3. **Select the Servers tab and click the Add Exception button.**
4. **In the Add Security Exception dialog box, enter the URL of the local host that is displayed in the web browser.**

   For example, https://localhost:50245
5. **Click the Get Certificate button and accept the certificate loaded by the Certificate Manager.**

   In some cases, you may need to restart the browser for the certificate to be accepted.

Running Web Application Samples

The following sections describe the individual web application samples contained in the `samples\web` folder and provide specific procedures used to run them. The following list of samples is ordered based on their complexity with the simplest `HelloWorld` sample listed first and the more complex `CardHolderAuthorization` sample listed last:

- `HelloWorld` - Demonstrates the base structure of a Java Card 3 platform application that developers can use to develop, deploy, create, execute, delete, and unload a stand-alone module. It is a minimal application utilizing the simplest source code and meta-files. See "Running the `HelloWorld` Sample" on page 24.
- `ContainerManagedAuthentication` - Demonstrates basic web container authentication with a PIN authenticator using JC-FORM with Biometric password. See "Running the `ContainerManagedAuthentication` Sample" on page 26.
- `StaticSecureWebHosting` - Demonstrates secure hosting on a static port. See "Running the `StaticSecureWebHosting` Sample" on page 27.

- `DynamicSecureWebHosting` - Demonstrates secure hosting on a dynamic port. See "Running the `DynamicSecureWebHosting` Sample" on page 29.
- `GCFClient` -Demonstrates the Generic Connection Framework (GCF) functionality of a http client, a socket client, and a socket server. See "Running the `GCFClient` Sample" on page 30.
- `DynamicallyLoadedClasses` - Demonstrates dynamic class loading using the `Class.forName()` method. See "Running the `DynamicallyLoadedClasses` Sample" on page 31.
- `Persistence` - Demonstrates the persistence of user entered information when a session on a card is resumed. See "Running the `Persistence` Sample" on page 33.
- `RestartableTasks` - Demonstrates how a registered task can be automatically available after the card is reset. See "Running the `RestartableTasks` Sample" on page 34.
- `Transactions` - Demonstrates transaction types and functions. See "Running the `Transactions` Sample" on page 36.
- `SIOFacility` - Demonstrates `isCLientInRole()` usage and usage of the transferable API. See "Running the `SIOFacility` Sample" on page 37.
- `EventFacility` - Demonstrates a servlet that registers a listener for events and a servlet that enables users to fire custom events. See "Running the `EventFacility` Sample" on page 39.
- `CardHolderAuthorization` - Demonstrates a remote client unable to access a resource and a card holder authorizing a second servlet to enable access by the remote client. See "Running the `CardHolderAuthorization` Sample" on page 41.

Note – See *Programming Notes, Java Card 3 Platform, Connected Edition* for information about writing web applications that run on the Java Card 3 platform.

Running the `HelloWorld` Sample

This application demonstrates the basic structure of a Java Card 3 platform application that developers can use to develop, deploy, create, execute, delete, and unload a stand-alone module. It is a minimal application utilizing the simplest source code and meta-files. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details.

This sample contains one web applications that demonstrates using a basic web form to collect and display information provided by the user. The project is located in the *JC_CONNECTED_HOME*`\samples\web` folder and is named `HelloWorld`.

Running the sample consists of using the command-line interface to start the `HelloWorld` application, entering a name in the web page, clicking the Say Hello button on the page, and then displaying a greeting.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

▼ Run `HelloWorld`

1. **Start the `HelloWorld` application.**

   a. **Go to the `HelloWorld` folder.**

   b. **Enter the `ant run` command.**

   When running, a browser displays the following page:

2. **Enter a name in the Enter Name field and click the Say Hello button.**

   The browser displays a greeting similar to the following illustration.

Running the `ContainerManagedAuthentication` Sample

This sample demonstrates basic web container authentication with a PIN authenticator using JC-FORM with a Biometric password. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details.

This sample contains a web application that demonstrates running a servlet that requires user authentication to access a web page. The project is located under the *JC_CONNECTED_HOME*`\samples\web` folder and is named `ContainerManagedAuthentication`.

Running the sample consists of using the command-line interface to start the `ContainerManagedAuthentication` application and login to the page by entering the name **`admin`** and password **`1234`**.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

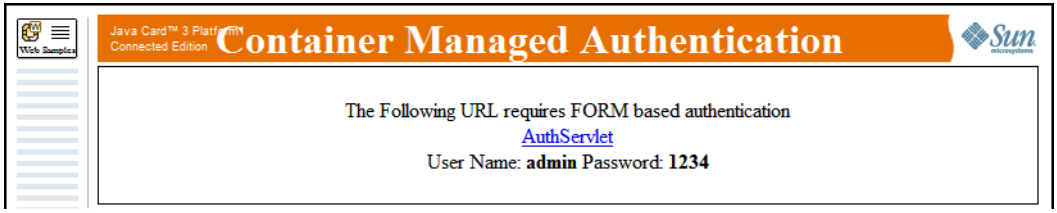
▼ Run `ContainerManagedAuthentication`

1. **Start the `ContainerManagedAuthentication` application.**

   a. **Go to the `ContainerManagedAuthentication` folder.**

   b. **Enter the `ant run` command.**

   When running, a browser opens that displays the following dialog:



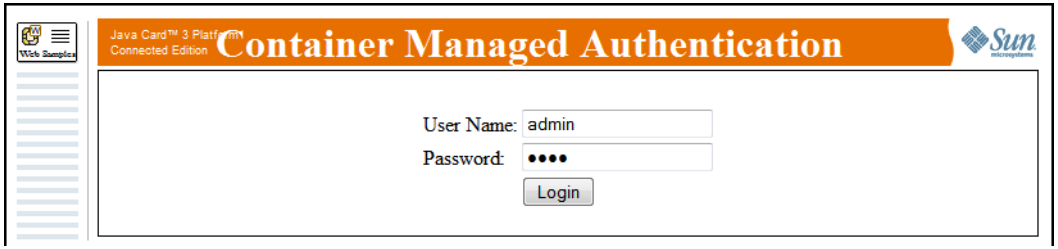
2. **Click the AuthServlet hyperlink.**

   The browser displays the login page.

3. **Enter the following values in the login dialog:**

   Login: **admin**

   Password: **1234**



- If the login is incorrect, the browser displays the following error page:

- If the login is correct, the browser displays the following protected page:

Running the `StaticSecureWebHosting` Sample

This sample demonstrates secure hosting with user access on a static port. The sample contains one web application that accesses a secure website hosted on a static port defined by the sample's `manifest.mf` file. Each time the application is run, access to the web site is made through this port. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details. The project is located under the *JC_CONNECTED_HOME*`\samples\web` folder and is named `StaticSecureWebHosting`.

Running the sample consists of using the command-line interface to start the `StaticSecureWebHosting` application, check the website's security certificate, and display the web page.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

▼ Run `StaticSecureWebHosting`

1. **Start the `StaticSecureWebHosting` application.**

   a. **Go to the `StaticSecureWebHosting` folder.**

   b. **Enter the `ant run` command.**

Note – If the browser displays a warning that the security certificate is not issued by a trusted certificate authority, disregard it and choose to continue to the web site. The security certificate for this sample is used for demonstration purposes only and cannot be used for developing deployable samples.If you are unable to continue to the web page, perform the procedure described in "Accepting an Untrusted Certificate" on page 23.

The browser displays the Secure hosting static port entry page.

2. **Enter a name in the Name field and click the Say Hello button.**

   The browser displays the static port web page.

Running the `DynamicSecureWebHosting` Sample

This sample demonstrates secure hosting with user access on a dynamic port. The sample contains one web application that accesses a secure website hosted on a dynamic port determined by the Java Card 3 platform. Each time the application is run, the Java Card 3 platform determines the port used to access the website. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details. The project is located under the *JC_CONNECTED_HOME*`\samples\web` folder and is named `DynamicSecureWebHosting`.

Running the sample consists of using the command-line interface to start the `DynamicSecureWebHosting` application, check the website's security certificate, and display the web page.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

▼ Run `DynamicSecureWebHosting`

1. **Start the `DynamicSecureWebHosting` application.**

   a. **Go to the `DynamicSecureWebHosting` folder.**

   b. **Enter the `ant run` command.**

Note – If the browser displays a warning that the security certificate is not issued by a trusted certificate authority, disregard it and choose to continue to the web site. The security certificate for this sample is used for demonstration purposes only and cannot be used for developing deployable samples. If you are unable to continue to the web page, perform the procedure described in "Accepting an Untrusted Certificate" on page 23.

The browser displays the Secure hosting dynamic port entry page.

2. **Enter text in the name field and click the Say Hello button.**

   The browser displays a greeting page that uses the text entered in the name field.

Running the `GCFClient` Sample

The GCF application demonstrates the Generic Connection Framework (GCF) functionality. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details.

Running the sample consists of using the command-line interface to start the `GCFClient` application, entering a URL (such as `http://www.sun.com`) in the URL field, and then clicking the Get Content button on the page to display the contents of the web page.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

▼ Run `GCFClient`

1. **Start the `GCFClient` application.**

   a. **Go to the `GCFClient` folder.**

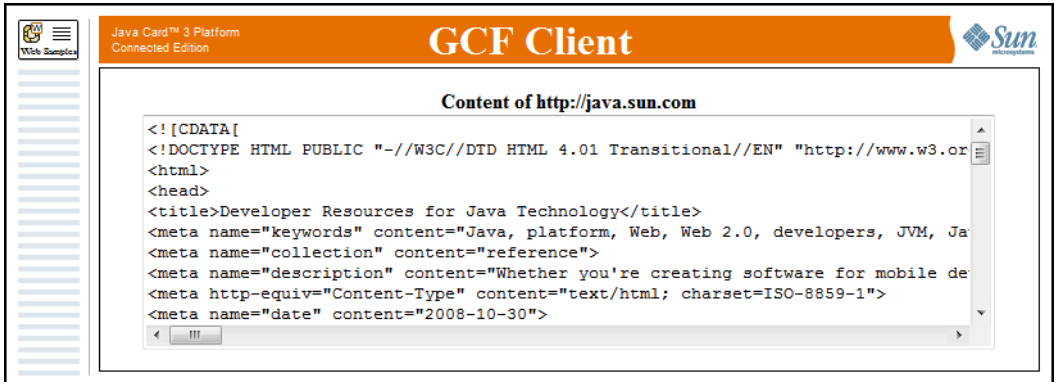
   b. **Enter the `ant run` command.**

   When running, a browser opens and displays the `GCF` Client Page.

2. **Enter a URL in the URL field and click the Get Content button.**

   The application establishes an http connection with the website and displays content from the web site in the browser.

Note – The sample works correctly only when a URL field contains a URL in a full form (such as `http://www.sun.com`). The sample does not work when the URL is provided in an abbreviated form (such as `www.sun.com`).



Running the `DynamicallyLoadedClasses` Sample

This sample demonstrates dynamic class loading using the `Class.forName()` method. The sample contains one web application that displays a web form enabling the user to enter a text string and to select a greeting type from a dropdown list. The application displays a text string that concatenates the greeting type with the text string entered by the user. Refer to the *Runtime Environment Specification, Java Card*

Platform, Version 3.0.1, Connected Edition for details. The project is located under the *JC_CONNECTED_HOME*`\samples\web` folder and is named `DynamicallyLoadedClasses`.

Running the sample consists of using the command-line interface to start the `DynamicallyLoadedClasses` application, entering text in the web page Name field, and clicking the Greet button to display the greeting page.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

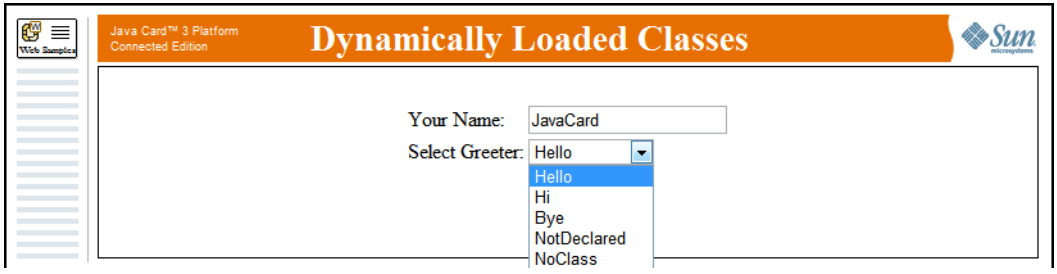
▼ Run `DynamicallyLoadedClasses`

1. **Start the `DynamicallyLoadedClasses` application.**

   a. **Go to the `DynamicallyLoadedClasses` folder.**

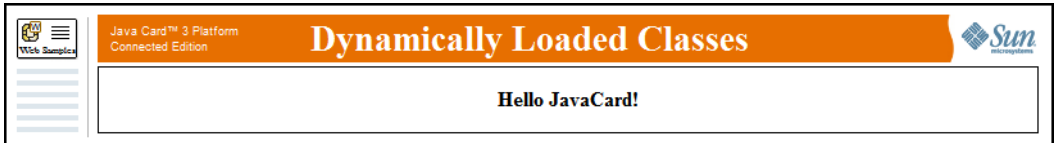
   b. **Enter the `ant run` command.**

   When running, a browser opens and displays the Dynamically Loaded Classes page.



2. **Enter a name in the name field, select a greeting, and click the Greet button.**

   The browser displays text that consists of the greeting type and the name entered in the name field.



Running the `Persistence` Sample

This sample demonstrates the persistence of user entered information when a session on a card is resumed. The sample stores user-entered information as history and, when the card resumes functioning, the previous entries are made available. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details.

Running the sample consists of using the command-line interface to start the `Persistence` application, clicking the hyperlink on the Persistence web page to open the database form, using the form to add and delete items in the database list, and stopping and restarting the server. When the server is stopped and restarted the items in the list should not change.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

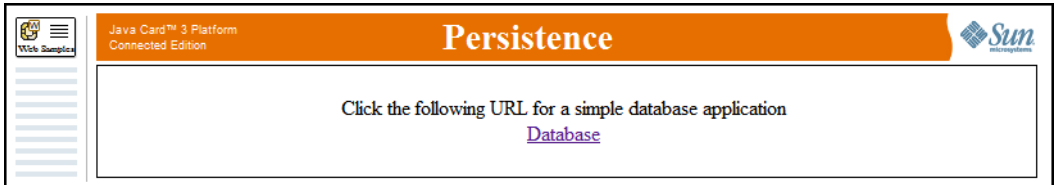
▼ Run `Persistence`

1. **Start the `Persistence` application.**

   a. **Go to the `Persistence` folder.**

   b. **Enter the `ant run` command.**

   When running, a browser opens that displays the Persistence page.

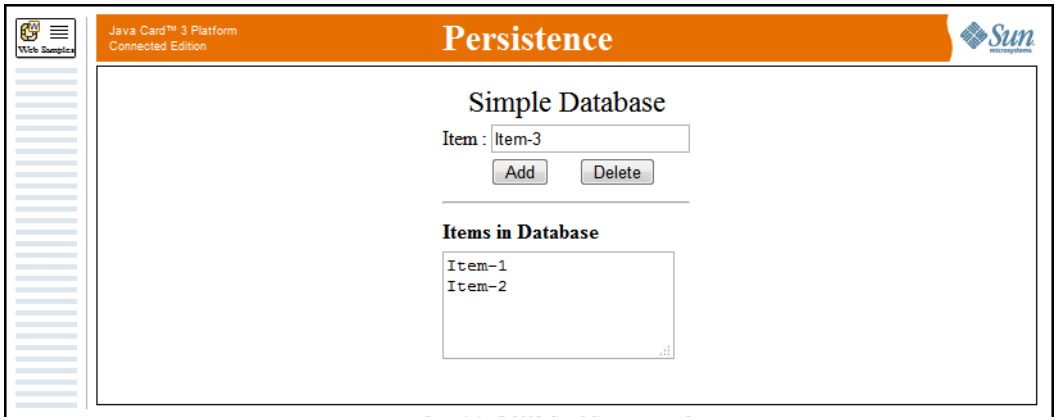


2. **Click the Database hyperlink.**

   The browser displays the database entry form.

3. **Enter text in the Item field and click the Add button.**

   The browser displays the added text in the Items in Database list. The following figure illustrates an entry form with two items added to the list.



4. **Delete an item from the list.**

   a. **Type the name of the item into the Item text field.**

   b. **Click the Delete button.**

   The item is removed from the Items in Database list.

5. **Stop and resume the server.**

   a. **In the `cjcre.exe`. window, kill the server by using `ctrl` + `C`.**

   b. **Open a new Command Prompt window and navigate to the `JC_CONNECTED_HOME\bin` directory.**

   c. **Restart the server from the new window by using, `cjcre.exe -resume`.**

6. **Verify that the content in the database list is unchanged.**

7. **Add a new item to the list to verify that the database is still functional.**

Running the `RestartableTasks` Sample

This sample demonstrates how a registered task can be automatically available after the card is reset. The sample registers a task that serves data via an HTTP connection to the client. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details.

This sample contains two web applications that demonstrate a registered task that is available after the card is reset. The projects are located under the *JC_CONNECTED_HOME*`\samples\web\RestartableTasks` folder. One project is named `InfoClient` and the other project is named `InfoServer`.

Running the `RestartableTasks` sample consists of using the command-line interface to start the `InfoClient` application and the `InfoServer` application, and clicking the Register Ping Service hyperlink to display the JCPingServer page. The server is stopped and restarted and the JCPingServer page.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

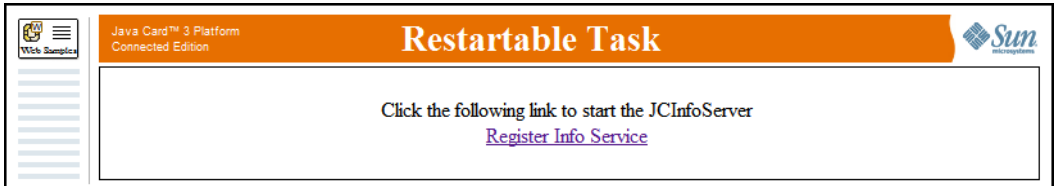
▼ Run `RestartableTasks`

1. **Start the `InfoServer` application.**

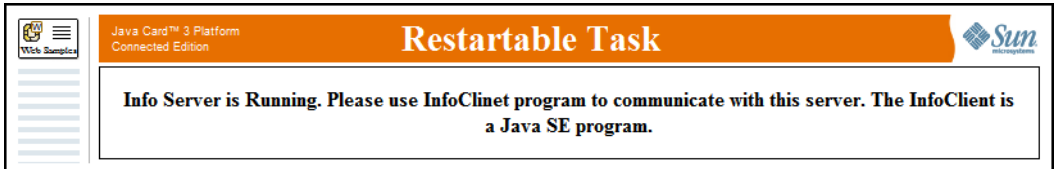
   a. **Go to the `InfoServer` folder.**

   b. **Enter the `ant run` command.**

   When running, a browser opens and displays the Restartable Tasks page.



2. **Click the Register Info Service link in the web page to start the JCInfoServer and display the server page.**



3. **Start the `InfoClient` application.**

   Each time the InfoClient application runs, the hit number is incremented in the output and a different text message is displayed.

4. **Stop and resume the server.**

   a. **In the `cjcre.exe`. window, kill the server by using `ctrl` + `C`.**

   b. **Open a new Command Prompt window and navigate to the `JC_CONNECTED_HOME\bin` directory.**

c. **Restart the server from the new window by using,** `cjcre.exe -resume`**.**

5. **Run the** `InfoClient` **application.**

   The hit number in the output text continues to increment from the previous value.

Running the `Transactions` Sample

The `Transactions` sample application contains a module that performs updates and either commits the transaction gracefully or aborts by throwing exception within code to simulate rollback. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details.

This sample contains a web application that demonstrates the event mechanism. The `Transactions` project is located under the *JC_CONNECTED_HOME*`\samples\web` folder.

Running the `Transactions` sample consists of using the command-line interface to start the `Transactions` application, entering data in the Transactions web page and clicking the Set Value button on the page to commit the transaction.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

▼ Run `Transactions`

1. **Start the** `Transactions` **application.**

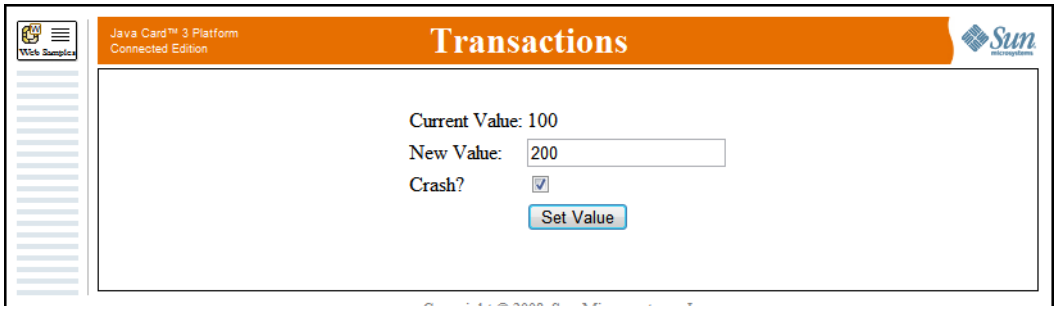
   a. **Go to the** `Transactions` **folder.**

   b. **Enter the** `ant run` **command.**

   When running, a browser opens and displays the Transactions page.

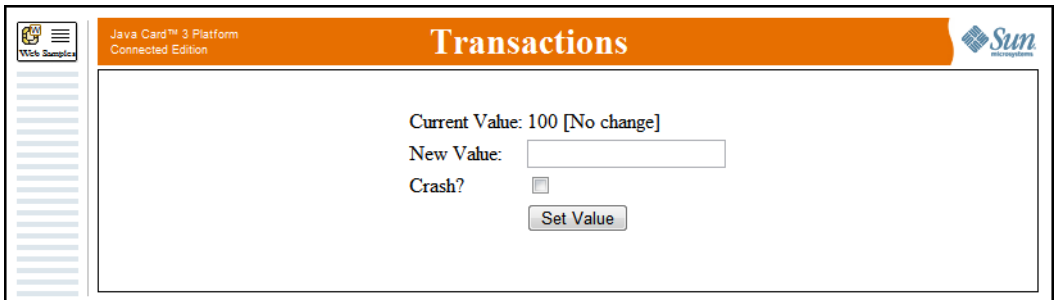
2. **Enter a set of characters in the New Value field and mark the Crash? checkbox.**

   The browser displays a page containing the contents of the value field. The page is similar to the following.



3. **Click the Set Value button.**

   The browser displays a page containing an exception message with the contents of the value field. The page is similar to the following.



4. **In the Transactions page, enter a set of characters in the Value field, do not mark the Crash? checkbox, and click the Set Value button.**

   The browser displays the Transactions page with the Current Value updated to the new value.

Running the `SIOFacility` Sample

This sample demonstrates inter-application communication using shared interface objects (SIOs). One servlet updates a shared object while the other can read updated values. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details.

This sample contains two web applications that demonstrate inter-application communication using SIOs. The projects are located under the *JC_CONNECTED_HOME*`\samples\web\SIOFacility` folder. One project is named `SIOservice` and the other project is named `SIOclient`.

Running the `SIOFacility` sample consists of using the command-line interface to start both the `SIOservice` application and the `SIOclient` application, entering values in the SIO Service web page, clicking the Set Value button on the page, and then using a hyperlink on the SIO page to display a list of received events.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

▼ Run `SIOFacility`

1. **Start the `SIOService` application.**

   a. **Go to the `SIOService` folder.**

   b. **Enter the `ant run` command.**

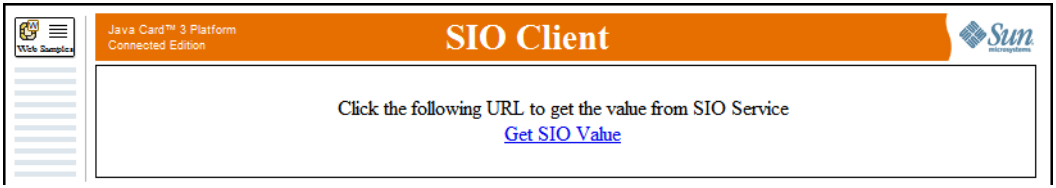
   When running, a browser opens that displays the SIO Service page.

2. **Enter a value in the New Value field and click the Set Value button.**

   The browser displays a page with the new value set.

3. **Use the command line to start the `SIOClient` application.**

   When running, a browser opens that displays the SIO Client page.



4. **Click the link in the servlet page to display the SIO value.**

   The value displayed on the page is the same as that set in the SIO Service page.

Running the `EventFacility` Sample

This sample demonstrates an Event facility containing two servlets, in which the first servlet registers a listener for events fired by another application. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details.

This sample contains two web applications that demonstrate the event mechanism. The projects are located under the *JC_CONNECTED_HOME*`\samples\web\EventFacility` folder. One project is named `EventSender` and the other project is named `EventListener`.

Running the `EventFacility` sample consists of using the command-line interface to start both the `EventListener` application and the `EventSender` application, entering data in the Event Sender web page, clicking the Fire Event button on the page, and then using a hyperlink on the Event Listener page to display a list of received events.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

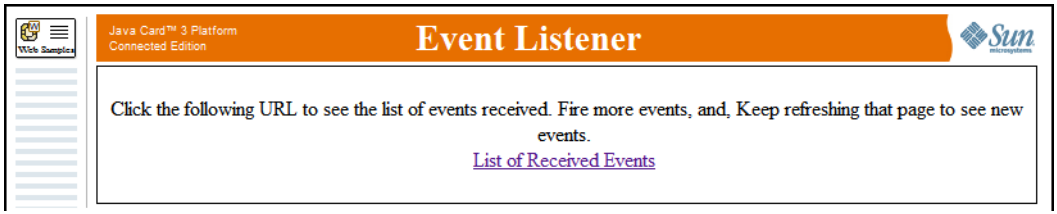
▼ Run `EventFacility`

1. **Start the `EventListener` application.**

   a. **Go to the `EventListener` folder.**

   b. **Enter the `ant run` command.**

   When the sample is running, a browser opens and displays the following web page containing a hyperlink to the List of Received Events.



2. **Start the `EventSender` application.**

   Depending on the browser settings, the application opens a new browser window or a new tab and displays the following Event Sender form:

3. **Enter some text in the Data field and click the Fire event button.**

   The browser displays a page with the new text.

4. **In the Event Facility Listener page, click the List of Received Events hyperlink.**

   The application displays the following list of events fired by the custom event servlet:

Running the `CardHolderAuthorization` Sample

This sample demonstrates how the authentication of a card holder to a locally accessible servlet grants a non-card holder access to a remotely accessible servlet. Card-holder-user authentication is tracked globally (card-wide). Authorization to access resources is protected by globally authenticated card-holder-user identity. Authorization to access resources can be granted by the card holder to other users. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details.

In this sample, servlets (`CardHolderApp` and `RemoteUserApp`) run on a local desktop that is networked with a remote desktop. After `CardHolderApp` is deployed and instantiated on the local desktop, the remote user attempts but fails to access `RemoteUserApp` on the local desktop.

After the login attempt fails, the card holder uses the `CardHolderApp` on the local desktop to authenticate and enable the remote user to access the `RemoteUserApp` on the local desktop.

In this sample, the URL for the `RemoteUserApp` is `http://`*IP Address*`:8020/RemoteUserApp` and the URL for the `CardHolderApp` is `http://localhost:8020/CardHolderApp`.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

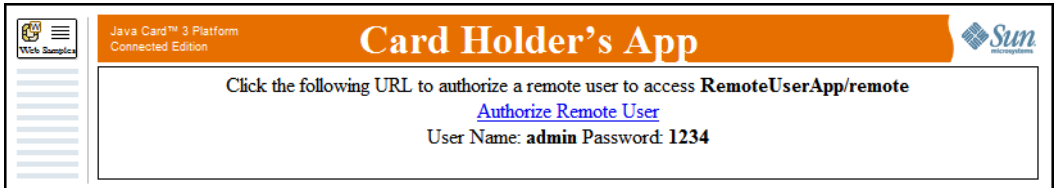
▼ Run `CardHolderAuthorization`

1. **Start the `CardHolderApp` application.**

   a. **Go to the `CardHolderApp` folder.**

   b. **Enter the `ant run` command.**

   The browser displays the Card Holder App page.



2. **Start the `RemoteUserApp` application.**

   a. **Go to the `RemoteUserApp` folder.**

   b. **Enter the `ant run` command.**

   The browser displays the Remote User's App page.

3. **From a remote workstation or PC networked with the platform running the sample, open a browser and enter the following URL:**

   URL: `http://`*IP Address*`:8019/remoteuserapp`

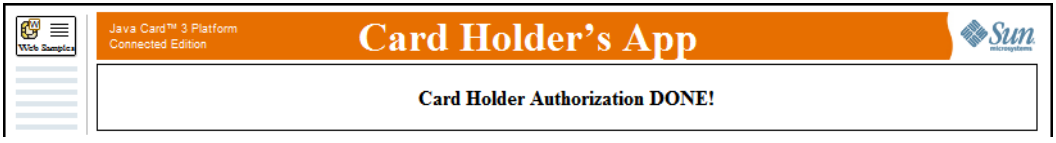
   This action is performed as a remote user who is attempting to access the Remote User's App page. The attempt fails and the browser displays an HTTP error 403 page in the browser stating that card holder authorization is required.

4. **In the Card Holder App page, click the Authorize Remote User hyper link and enter the following login and password:**

   Login: **`admin`**

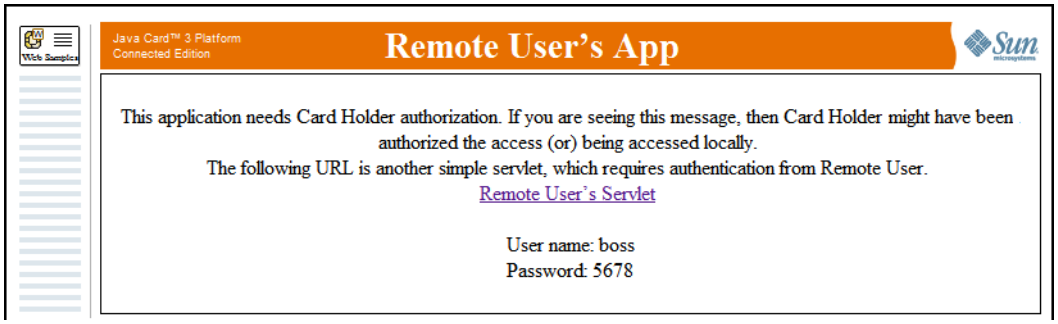
   Password: **`1234`**

   This action is performed as the card holder. The sample displays the authorization page in the browser.



5. **From the remote workstation or PC, either refresh the browser page displayed in step 2 or enter the URL from Step 3 in a new browser page.**

   The sample displays the Remote User page in the browser.



6. **In the Card Holder App page, click the Remote User's Servlet hyper link and enter the following login and password:**

   Login: **`boss`**

   Password: **`5678`**

7. This action is performed as the remote user. The sample displays a Welcome Remote User page in the browser.

Running Classic Applet Samples

The following section describes the classic applet sample contained in the `samples\classic_applets` folder and provides the procedure used to run it.

`ClassicChannels` Sample

The `ClassicChannels` sample demonstrates the behavior of Java Card technology-based logical channels by showing how two applets that interact with each other can each be selected for use at the same time.

The applets may use a contact or contactless interface for communication with the terminal. The `ClassicChannels` sample demonstrates the selection of an applet on both interfaces. The sample also demonstrates use of ExtendedLength APDU.

The `ClassicChannels` sample mimics the behavior of a wireless device connected to a network service. A connection manager tracks whether the device is connected to the service and whether the connection is local or remote.

While it is connected, the user's account is debited on a unit of time basis. The debit rate is based on whether the connection is local or remote, and uses either the contacted or contactless interface.

The sample employs two applets to simulate the behavior of logical channels:

- The `ConnectionManager` applet manages the connection.
- `AccountAccessor` applet manages the account.

When the user turns on the device, the `ConnectionManager` applet is selected. The `ConnectionManager` implements the ExtendedLength interface to handle APDUs with larger data segments such as the ones used for key exchange in the sample. Every unit of time the terminal sends a message containing the area code to the card.

When the user wants to use the service, the `AccountAccessor` applet is selected on another logical channel so that the terminal can query the balance. The `AccountAccessor` can return the balance only if the `ConnectionManager` is active. The `ConnectionManager` applet sets the connection and tracks the connection status. Based on the value of an area code variable, the `ConnectionManager` determines whether the connection is local or remote. It also determines whether the connection is contacted or contactless. `AccountAccessor` uses this information to debit the account at the appropriate rate. The connection is disabled when the user completes the call or when the account is depleted.

▼ Run the `ClassicChannels` Sample

1. **Start the `ClassicChannels` application.**

   a. **Go to the `ClassicChannels` folder.**

   b. **Enter the `ant run` command.**

      The ant script either generates the default output file, `default.out` or the output file name specified in the command line. To specify the name of the output file use the following command:

      `ant -Dredirect.output=`*`outputfile_name`* `run`

      In this command, *outputfile_name* represents the name of the output file. This command redirects the output from the APDUtool execution to the *outputfile_name* file.

2. **Verify that the contents of the output file created by the `ant run` command are the same as the contents of the `ClassicChannels.expected.out` file.**

Running Extended Applet Samples

This release includes two extended applet samples in `samples/extended_applets` that illustrates the use of the Java Card API, scenarios of package masking, and post-manufacture installation.

This section consists of the following sections:

Description of Extended Applet Samples

Version 3.0.1 of the Development Kit includes the `HelloWorld` extended applet sample.

Building the Extended Applet Samples

Ant script files are provided to build the samples, demonstration masks, and `cjcre`.

See *Programming Notes, Java Card 3 Platform, Connected Edition* for additonal information about creating extended applets.

Running the `HelloWorld` Sample

This sample demonstrates the basic structure of a Java Card 3 platform extended applet that developers can use to develop, deploy, create, execute, delete, and unload extended applets. It is a minimal extended applet utilizing the simplest source code and meta-files. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details.

This sample contains one project that demonstrates the function of an extended applet. The project is located in the *JC_CONNECTED_HOME*`\samples\extended_applets` folder and is named `HelloWorld`.

Running the sample consists of using the command-line interface to start the `HelloWorld` extended applet. When running, the project installs the extended applet, processes an incoming APDU, and responds with a text greeting.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

▼ Run the `HelloWorld` Sample

1. **Start the `HelloWorld` application.**

   a. **Go to the `HelloWorld` folder.**

   b. **Enter the `ant run` command.**

      The ant script either generates the default output file, `default.out` or the output file name specified in the command line. To specify the name of the output file use the following command:

      `ant -Dredirect.output=`*outputfile_name* `run`

      In this command, *outputfile_name* represents the name of the output file. This command redirects the output from the APDUtool execution to the *outputfile_name* file.

2. **Verify that the contents of the output file created by the `ant run` command are the same as the contents of the `HelloWorld.expected.out` file.**

Running the `ExtendedChannels` Sample

This sample demonstrates the basic structure of a Java Card 3 platform extended applet that developers can use to develop, deploy, create, execute, delete, and unload extended applets. It is a minimal extended applet utilizing the simplest source code and meta-files. Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for details.

This sample contains one project that demonstrates the function of an extended applet. The project is located in the *JC_CONNECTED_HOME*`\samples\extended_applets` folder and is named `ExtendedChannels`.

Running the sample consists of using the command-line interface to start the `ExtendedChannels` extended applet. When running, the project installs the extended applet, processes an incoming APDU, and responds with a text greeting.

Note – Using the command line to start a sample is described in "Run Samples from the Command Line" on page 22.

▼ Run the `ExtendedChannels` Sample

1. **Start the `ExtendedChannels` application.**

   a. **Go to the `ExtendedChannels` folder.**

   b. **Enter the `ant run` command.**

      The ant script either generates the default output file, `default.out` or the output file name specified in the command line. To specify the name of the output file use the following command:

      `ant -Dredirect.output=`*outputfile_name* `run`

      In this command, *outputfile_name* represents the name of the output file. This command redirects the output from the APDUtool execution to the *outputfile_name* file.

2. **Verify that the contents of the output file created by the `ant run` command are the same as the contents of the `ExtendedChannels.expected.out` file.**

Running Reference Application samples

This release includes a sample reference application in `samples/reference_apps` that illustrate the use of the Java Card API, scenarios of package masking, and post-manufacture installation.

This section consists of the following sections:

- Description of `reference_apps` Samples
- Building a `Transit` Sample Application
- Running the `Transit` Sample

Description of `reference_apps` Samples

Version 3.0.1 of the Development Kit includes the `Transit` reference application sample.

Directories and Files in the `reference_apps` Directory

The `reference_apps` directory is located at *JC_CONNECTED_HOME*`\samples`. It contains the `Transit` sample directory, which consists of the `AdminWeb`,`ClassicWalletApplet`, `POSWeb`, `TransitExtLib`, `TurnstileApplet`, `TurnstileClient`, `TurnstileWeb`, `WalletAssistApplet`, and `WalletClassicLib` source directories and files used to build and run the projects and applications that form the Transit sample.

Building a `Transit` Sample Application

The each application in the `Transit` sample contains a `build.xml` at its root level folder. Developers can use `build.xml` with the `ant` tool to build a sample application without running it.

To build a sample application without running it, use the Command Prompt window to navigate to the appropriate application directory and enter the `ant` command. For example, to build the `ClassicWalletApplet` sample application without running it, navigate to the `ClassicWalletApplet` folder and enter the following command:

```
ant
```

The `ant` tool runs the tools (compiler and Packager) required to build the sample application. It displays a build status message at completion of the task.

See *Programming Notes, Java Card 3 Platform, Connected Edition* for detailed information about the Transit sample.

Running the `Transit` Sample

The `Transit` sample is run by building and running individual `Transit` sample applications in the following sequence:

- `WalletClassicLib`
- `TransitExtLib`
- `ClassicWalletApplet`

- `WalletAssistApplet`
- `POSWeb`
- `AdminWeb`
- `TurnstileApplet`
- `TurnstileClient`
- `TurnstileWeb`

A `RunTransit.bat` file is provided in the `Transit` directory that automatically builds and runs the `Transit` sample applications in their proper sequence. When running the sample with the `RunTransit.bat` file, use the following procedures for performing the actions that are web-page based.

See "General Procedures for Running Samples" on page 21 for a description of the steps performed in running a sample.

Note – The POSWeb application has been internationalized and can be localized for the French language. FIGURE 4-1 illustrates the language setting screen in Firefox. FIGURE 4-2 and FIGURE 4-3 are examples of two POSWeb sample screens as they appear when localized for the French language.

FIGURE 4-1 Browser Language Selection Dialog

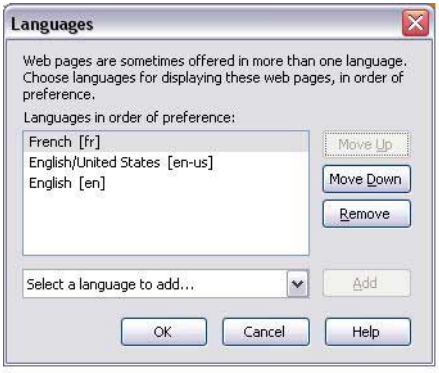


FIGURE 4-2 Example of French Language Version of the Transit Point of Sale Page

FIGURE 4-3 Example of French Language Version of Transit History Page

▼ Run the `Transit` Sample

Note – See "General Procedures for Running Samples" on page 21 for a general description of the steps performed in running a sample.

---

1. **Run the `WalletClassicLib` sample application.**

   a. **Go to the `WalletClassicLib` folder.**

   b. **Enter the `ant run` command.**

   Verify that a second terminal window opens, `cjcre.exe` starts, and the the first window displays, `Build Successful`.

Note – Do not close the `cjcre.exe` window.

2. **Run the `TransitExtLib` sample application.**

   a. **Go to the `TransitExtLib` folder.**

   b. **Enter the `ant run` command.**

   Verify that a second terminal window opens, `cjcre.exe` starts, and the the first window displays, `Build Successful`.

Note – Do not close the `cjcre.exe` window.

3. **Run the `ClassicWalletApplet` sample application.**

   a. **Go to the `ClassicWalletApplet` folder.**

   b. **Enter the `ant run` command.**

   Verify that the applet was successfully created. The SELECT APDU command returns success status word 90 00. Additional APDU commands are used to credit the Wallet additional $100 (0x64). Verify that these commands return success status words 90 00.

4. **Run the `WalletAssistApplet` sample application.**

   a. **Go to the `WalletAssistApplet` folder.**

   b. **Enter the `ant run` command.**

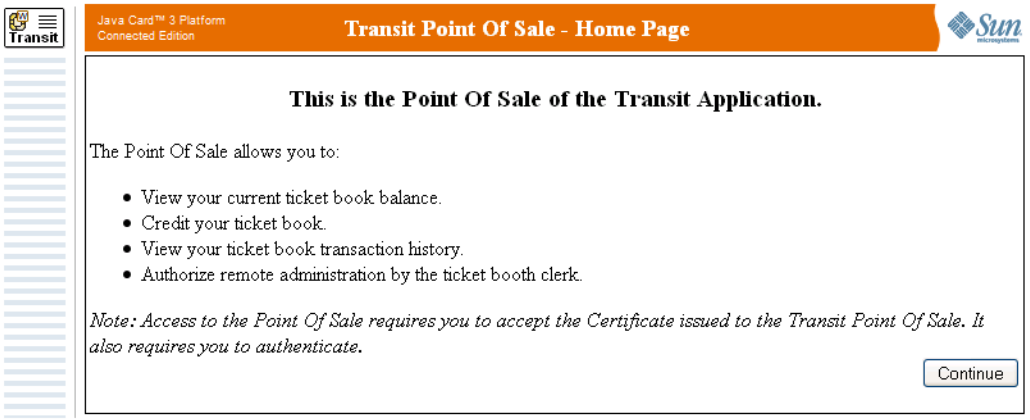
   Verify that the applet was successfully created. The SELECT APDU command returns Status word 69 99.

5. **Run the `POSWeb` sample application.**

   a. **Go to the `POSWeb` folder.**

   b. **Enter the `ant run` command.**

   The browser opens and displays the following page:

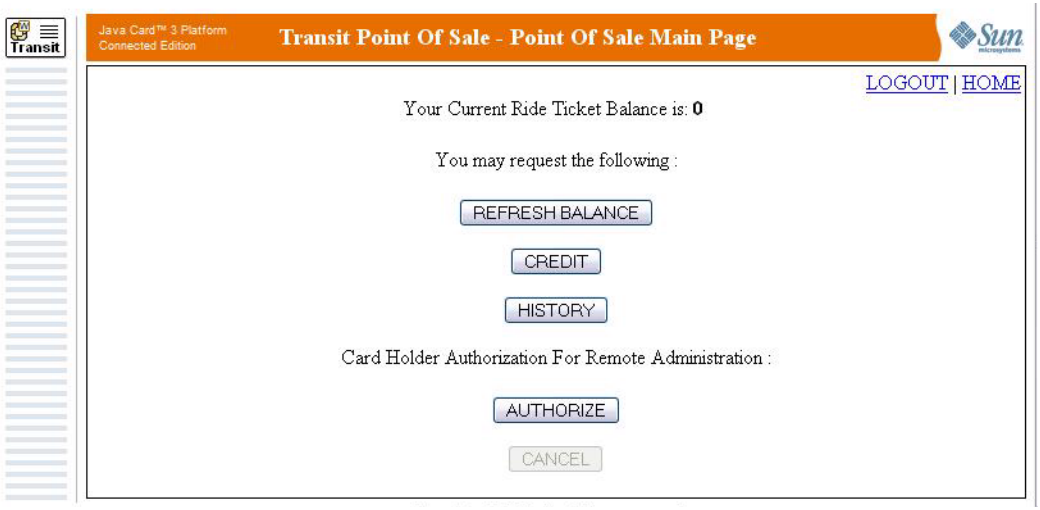


c. Click the Continue button and in the login screen, enter the User Name and PIN as:

- **`owner-pos`**
- **`8888`**

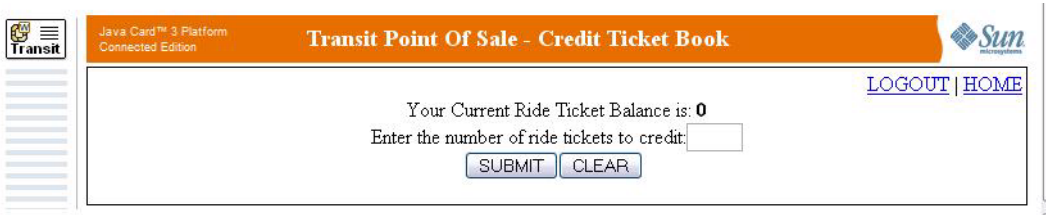
Note – If the browser displays a warning that the security certificate is not issued by a trusted certificate authority, disregard it and choose to continue to the web site. The security certificate for this sample is used for demonstration purposes only and cannot be used for developing deployable samples.If you are unable to continue to the web page, perform the procedure described in "Accepting an Untrusted Certificate" on page 23.

The browser displays the transaction page.



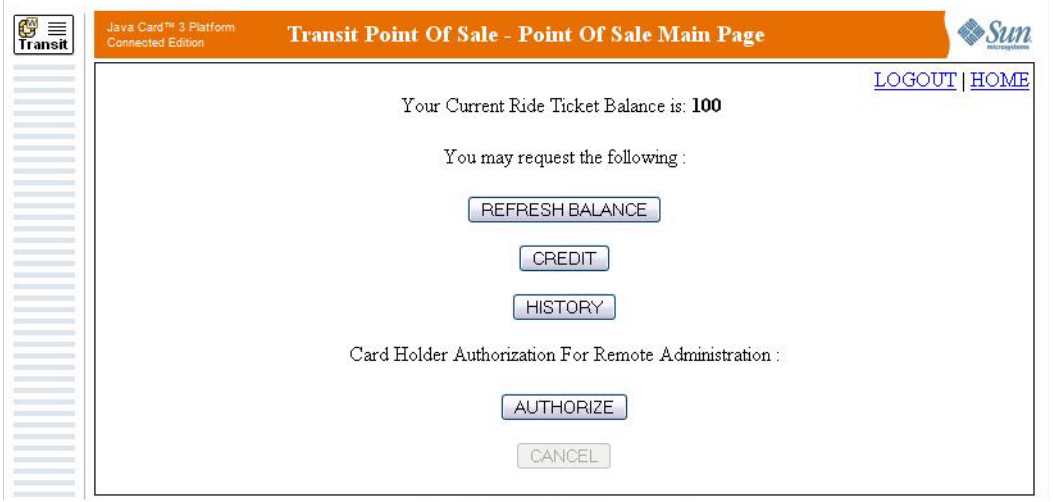
d. Click the CREDIT button.

The browser displays the Credit Request page.



e. Enter a number in the field and click the SUBMIT button.

The page displays the credited amount, in this example 100.



f. Click the AUTHORIZE button.

This authorizes access to the Admin application from a non-trusted client (the Internet Explorer browser in this RI). Clicking the CANCEL button (when enabled) cancels remote authorization.

6. Run the `AdminWeb` sample application.

a. Go to the `AdminWeb` folder.

b. Enter the `ant run` command.

The Administration login page is displayed.

Note – If the browser displays a warning that the security certificate is not issued by a trusted certificate authority, disregard it and choose to continue to the web site. The security certificate for this sample is used for demonstration purposes only and cannot be used for developing deployable samples.If you are unable to continue to the web page, perform the procedure described in "Accepting an Untrusted Certificate" on page 23.

---

c. In the login screen, enter the login User Name and PIN as:

- **`remote-admin`**
- **`8888`**

d. Click the Login button.

The browser displays the Administration page.

7. Run the `TurnstileApplet` sample application.

a. Go to the `TurnstileApplet` folder.

b. Enter the `ant run` command.

The log window displays a 90 00 in select response.

c. Return to the AdminWeb page and click the UNBLOCK button.

8. Clean and build the `TurnstileClient`.

a. Go to the `TurnstileClient` folder.

b. Enter the `ant` command (builds without running `TurnstileClient`).

9. Run the Turnstile Client by executing the `run_tranist_client.bat` file in the `Transit/TurnstileClient` directory.

10. Run the `TurnstileApplet` sample application.

The Output window displays 90 00 completes and SUCCESSFULL.

11. **In the Transit Point of Sale Main Page, click the REFRESH BALANCE button.**

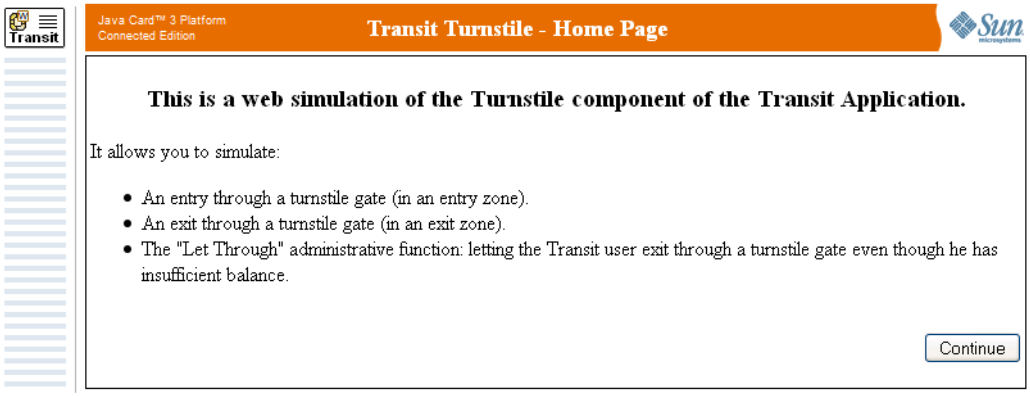
    The balance displays 2 less - 98.

12. **Run the `TurnstileWeb` sample application.**

    a. **Go to the `TurnstileWeb` folder.**

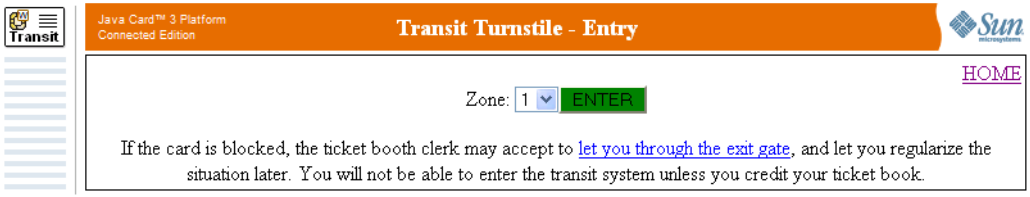
    b. **Enter the `ant run` command.**

    The browser opens and displays the Transit Turnstile Home Page.



    c. **Click the Continue button.**

    The browser displays the Entry page.



    d. **Return to the Transit Administration Main Page and click the UNBLOCK button.**

    e. **Select Zone 1 and click the ENTER button.**

    The browser displays the Transit Turnstile Exit page.

f. Click the Exit button.

g. Click the HISTORY button on the Transit Point of Sale Main Page.

CHAPTER 5

Starting the Java Card Runtime Environment

The Connected Edition reference implementation is written in the Java and C programming languages and is called the C Java Card Runtime Environment (Java Card runtime environment). It is a simulator that can be built with a pre-built ROM mask, much as a Java Card technology-based implementation. It has the ability to simulate persistent memory (EEPROM) as well as to save and restore the contents of EEPROM to and from disk files. The Java Card runtime environment performs I/O via a socket interface, simulating the interaction with a card reader or host machine implementing HTTP(S) communication with the card reader or host machine.

The Java Card runtime environment is supplied by the Development Kit as the pre-built executable, `cjcre.exe`. The executable, `cjcre.exe`, is run from the command line.

This chapter includes the following sections:

Starting `cjcre.exe` from the Command Line

The Java Card runtime environment can be run from the command line by using the following command and options:

JC_CONNECTED_HOME`\bin\cjcre.exe` [*options*]

`cjcre.exe` Command Line Options

The following command line options are listed in order of their expected frequency of use (most frequently used to less frequently used):

- `-version` - Displays version information.
- `-help` [`copyright`] - Prints help and copyright messages.
- `-resume` - Restores the VM state from the previously saved EEPROM image and continues VM execution.

  When `-resume` is specified, other options such as `-ramsize and -e2psize` are ignored and the corresponding values are obtained from the EEPROM image.
- `-e2pfile` *filename* - Supplies the file name in which the EEPROM image is stored.
- `-ramsize` *size* - Configures the amount of RAM used.

  The range of values that the Java Card runtime environment can accept from the command line is `64K` to `32M`. The default value used is `1M`. The value required to run the samples is between `128K` and `32M`.

  size is set as a value in bytes (`2345`), kilobytes (`32K`), or megabytes (`4M`).
- `-e2psize` *size* - Configures the amount of EEPROM used.

  The range of values that the Java Card runtime environment can accept from the command line is `1M` to `32M`. The default value used is `4M`. The value required to run the samples is between `2M` and `32M`.

  size is set as a value in bytes (`2345`), kilobytes (`32K`), or megabytes (`4M`). The specified size is rounded up to a multiple of 4. For example, a size specified at 253, is rounded up to 256.
- `-corsize` *size* - Sets the Clear On Reset (COR) memory size in which a portion of RAM is dedicated to COR memory.

  The range of values that the Java Card runtime environment can accept from the command line is `2K` to `8K`. The default value is `4K`.

  size is set as a value in bytes (`2345`) or kilobytes (`2K`).
- `-httpport` *portnumber* - Sets the HTTP port number on which `cjcre` will be listening for http requests.

  The default value for `-httpport` is 8019.
- `-contactedport` *portnumber* - Sets the port used to simulate the contacted interface for APDU.

  The default value for `-contactedport` is 9025.
- `-contactedprotocol` *protocol* - Sets the APDU protocol on this port, either `T=0` or `T=1`.

  The default value for `-contactedprotocol` is `T=1`.

- `-contactlessport` *port-number* - Sets the port used to simulate the contactless interface.

  The default value for `-contactlessport` is 9026. The protocol (`T=CL`) cannot be changed.

- `-debugger` - Runs `cjcre` in debug mode.
- `-debugport` *portnumber* - Sets the debug port where the Debug proxy communicates.

  The default value for `-debugport` is 7019.

- `-nosuspend` - Valid when `-debugger` is specified. Does not suspend the threads at `cjcre` startup.
- `-enableassertions` - Enables Java code assertions (the assert keyword in Java code).
- `-loggerlevel <none|fatal|error|warn|info|verbose|debug|all>` - Sets the type of log messages output.

  All log messages up to the specified level are displayed.

- `-config` *config file* - Sets a new configuration file.

  The default is `lib/config.properties`.

- `-X`*name=value* - Sets a single configuration property such as `-Xmyproperty=myvalue`.
- `-D`*name=value* - Supplies a system property (such as `-D`*myproperty=myvalue*).

  System properties set in this manner can be retrieved using the API's `System.getProperty("myproperty")` method. A maximum of 50 `-D` properties can be passed in the command line.

Java Card Runtime Environment Configuration Files

If you installed the Development Kit source bundle, the configuration files for the Java Card runtime environment (`config.properties` and `system.config`) files are located in the `lib` folder. These configuration files contain internal configuration information that must not be changed unless specified. Java Card runtime environment execution requires properly configured `config.properties` and `system.config` files. Incorrect changes to these files will prevent execution of the Java Card runtime environment. See Chapter 12 for details on configuring the Java Card runtime environment.

If you installed the Development Kit binary bundle, you cannot change the configuration files for the Java Card runtime environment.

Adding Proprietary Packages

If you installed the Development Kit source bundle, you can add proprietary packages to the ROM mask for the Java Card runtime environment by building a custom `cjcre.exe`. See Chapter 13 for additional information and procedures.

If you installed the Development Kit binary bundle, you cannot add proprietary packages.

CHAPTER 6

Compiling Source Code

This chapter describes the use of the Java Card 3 platform Compiler tool (`javacardc.bat`) in compiling the source code of applications outside of an IDE.

See Chapter 3 to better understand the role and relationship between the Compiler tool and the other Development Kit tools used in developing applications for the Java Card 3 platform.

Running the Compiler Tool from the Command Line

The Compiler tool provides a wrapper for `javac` (the JDK compiler) and includes an annotation processor for the Java Card 3 platform to check for unsupported language features, such as the use of `float` and `double`.

Compiler Tool Options

In addition to Java Card 3 platform specific options, all standard `javac` options for JDK 1.6 can be used:

TABLE 6-1 Compiler Tool Options

Option	Description
`-g`	Generate all debugging info
`-g:none`	Generate no debugging info
`-g:{lines,vars,source}`	Generate only some debugging info

TABLE 6-1 Compiler Tool Options *(Continued)*

Option	Description
`-nowarn`	Generate no warnings
`-verbose`	Output messages about what the compiler is doing
`-deprecation`	Output source locations where deprecated APIs are used
`-classpath` *path*	Specify where to find user class files and annotation processors
`-cp` *path*	Specify where to find user class files and annotation processors.
`-sourcepath` *path*	Specify where to find input source files.
`-bootclasspath` *path*	Override location of bootstrap class files.
`-extdirs` *dirs*	Override location of installed extensions.
`-endorseddirs` *dirs*	Override location of endorsed standards path.
`-proc:{none,only}`	Control whether annotation processing and/or compilation is done.
`-processor` *class1*`[,`*class2*`,`*class3*`...]`	Names of the annotation processors to run; bypasses default discovery process.
`-processorpath` *path*	Specify where to find annotation processors.
`-d` *directory*	Specify where to place generated class files.
`-s` *directory*	Specify where to place generated source files.
`-implicit:{none,class}`	Specify whether or not to generate class files for implicitly referenced files.
`-encoding` *encoding*	Specify character encoding used by source files.
`-source` *release*	Provide source compatibility with specified release.
`-target` *release*	Generate class files for specific VM version.
`-version`	Version information.
`-help`	Print a synopsis of standard options.
`-Akey[=value]`	Options to pass to annotation processors.
`-X`	Print a synopsis of nonstandard options.
`-J`*flag*	Pass *flag* directly to the runtime system.

Format

The following is an example of the Compiler tool command format:

`javacardc.bat [options] [sourcefiles] [@list_files]`

In the format example:

- *options* - standard javac options,
- *sourcefiles* - `.java` files to be compiled
- *@list_files* - plain text file containing a list of all java files that need to be compiled

Examples

A `.java` file named `UsesFloat.java` contains the following source:

```
public class UsesFloat {
    float f = 0;
}
```

It uses `float`, which is not supported by the Java Card 3 platform. Compiling this file with standard javac generates a class file without any errors. However, `javacardc.bat` fails the compilation with an error such as the following:

```
C:\JCDK3.0.1_ConnectedEdition\bin>javacardc.bat UsesFloat.java
Java Card 3.0.1 Compiler
UsesFloat.java:2: float keyword used
    float f = 0;
    ^
1 error
```

The bold text in the example output indicates the error message text.

CHAPTER 7

Creating and Validating Application Modules

This chapter describes creating and validating a Java Card technology-based application module with the Packager tool (Packager). See Chapter 3 to better understand the role and relationship between the Packager and the other Development Kit tools used in developing applications for the Java Card 3 platform.

This chapter contains the following sections:

Packager Operation

When creating an application module, the Packager takes a specified folder containing the files for the application module, validates the input files and creates the application module archive file. If a web application contains JAR files in the `lib` directory, the Packager creates a corresponding library module in the application module.

Each application module can have a descriptor as a part of the `MANIFEST.MF` file that specifies application module declarative items. In cases where an application module has a descriptor, the descriptor information must be validated and preserved.

Options

The following are options of the Packager:

- Modules can be passed to the Packager as paths to directories containing the corresponding structure.
- Manifest files with information contained in an input module folder are preserved without change.

Basic Packaging Sequence

The Packager creates an application module JAR file from input by performing the following actions:

1. Input files are extracted into a `temp` folder under a folder named either with the input file name or a name specified as a command line parameter.
2. Application module file types are checked and the application module type is determined.
3. A type entry is added to the application module.
4. The application module is placed under the `temp` folder.

   If an optional keystore file is specified in the command line parameter, verified information from it is added to the resulting application module.
5. The entire contents are grouped together to create the final application module JAR file.

Use Cases

TABLE 7-1 provides a description of the possible Packager input files and corresponding output conditions.

TABLE 7-1 Packager Tool Input Files and Expected Output

Input	Expected Output
A valid JAR file	A valid application module JAR file
A malformated JAR file	Packager warns the user and exits
Files of the same type	A valid application module JAR file
Files of different types	Packager warns the user and exits
Files of the same type but the type contradicts the passed `--type` argument	Packager warns the user and exits

Signing

The Packager can invoke the appropriate tools automatically to sign the application module JAR file. See "`create` Subcommand" on page 70. For information about creating a custom keystore that can be used to sign the application module JAR file, see Chapter 12.External signing tools can also be used to sign the modules if the user knows about those tools.

The Packager assumes that the keystore has everything needed in it and simply invokes the `jarsigner` to sign the module.

Use Cases

TABLE 7-2 provides a description of the possible Packager signing input and corresponding output conditions

TABLE 7-2 Packager Tool Signing Results

Input	Expected Output
Valid keystore passed	Application module JAR file is signed successfully
Invalid keystore passed or invalid keystore username or password	Packager warns the user and exits

Running the Packager from the Command Line

The command line interface for the Packager has the following syntax:

`packager.bat` *subcommand* `[options]` *module-or-folder*

The following is a list of the available subcommands for the Packager:

- `create` Subcommand
- `validate` Subcommand
- `copyright` Subcommand
- `help` Subcommand

`create` Subcommand

Creates the application module or library from a given module or folder.

`create` Subcommand Options

TABLE 7-3 identifies the `create` subcommand options and provides their descriptions.

TABLE 7-3 `create` Subcommand Options

Options	Description
`-A` *alias* or `--alias` *alias*	Application signing attribute, where alias is the name used to retrieve the key from the keystore.

`-c`
or
`--compress`
	Optional. If specified, the tool compresses the output application module file with DEFLATE algorithm. Otherwise creates an uncompressed application module file.

`-e` *path-of-export-files* or `--exportpath` *path-of-export-files*	Specifies the export files path. System's `api_export` files are implicitely loaded.

`-f`
or
`--force`
	Optional. If specified, descriptors or application module assembly problems are automatically corrected when possible. See "`--force` Option Behavior" on page 71.

`-K` *keystore-file* or `--keystore` *keystore-file*	Required only when the `--sign` option is specified. Application signing attribute, where *keystore-file* is the path and filename where the private keys are stored. A key utility (such as the JDK keytool) must be used to create and maintain this file. See Chapter 12, "Creating a Custom keystore" on page 114.

`-n`
or
`--nowarn`
	Suppresses the warning messages.

`-o` *file-name* or `--out` *file-name*	Specifies the output application module file where *file-name* is the name of the output file.
-P *key-password* or --passkey *key-password*	Application signing attribute, where *key-password* is the password for the private key.

TABLE 7-3 `create` Subcommand Options *(Continued)*

Options	Description
`-p` *package-AID-for-classic-lib* or `--packageaid` *package-AID-for-classic-lib*	Specifies the package AID in //AID/<RID>/<PIX> format for `classic-lib`. Ignored if type is not classic-lib.

`-s`
or
`--sign`
	Optional. Specifies that the Packager sign the application.
	If `--sign` is specified, `--keystore` *keystore-file*, `--storepass` *keystore-password*, `--`passkey *key-password*, and `--alias` *alias* are required.

`-S` *keystore-password* or `--storepass` *keystore-password*	Application signing attribute, where *keystore-password* is the password for the keystore.
`-t` *file-type* or `--type` *file-type*	Specifies the application module file type, where *file-type* can be `web, extended-applet, classic-applet, classic-lib,` or `extension-lib`. The default value is `web`.

`--force` *Option Behavior*

The `--force` option affects the following aspects of the Packager's behavior:

- If the manifest file is missed in a passed application module, the Packager warns the user and tries to gather required information and other available information (such as mandatory attributes defined in the specification, names of passed modules, and web application private libraries).
- If the runtime descriptor of a passed application module(s) and/or an external runtime descriptor contains unsupported information for determined module type, this information is removed (with appropriate warnings).
- If mandatory information is missed in runtime descriptor files, the Packager tries to add it (with appropriate warnings).
- If specified destination directory does not exists, the Packager creates it.
- If the specified output JAR file exists, the Packager overwrites it.

Note – Check the result runtime descriptor created by the Packager in `force` mode. Though the Packager attempts to automatically correct descriptors, the result is not guaranteed. Developers should use this option carefully.

`create` Subcommand Format

The following is an example of the `create` subcommand format:

```
packager.bat create --out file-name [--type file-type] \
          [--exportpath path-of-export-files] \
          [--packageaid package-AID-for-classic-lib] \
          [--sign --storepass keystore-password --passkey key-password \
          --alias alias] [--compress] [--force] [--nowarn]\
           module- file-or-folder
```

`create` Subcommand Examples

Two examples are provided, an example of the output option and an example of the signing option.

Output Option Example

The following is an example of the `create` subcommand with the output option:

```
packager.bat create -o mymodule.jar -t web -c c:\mymodulefolder
```

In this command line example, the Packager performs the following tasks:

1. Extracts the contents of `mymodulefolder` directory to a temporary folder under the subdirectory `mymodulefolder`.
2. Creates corresponding Web Application Module object and performs validation and canonicalization of all xml descriptors.
3. Creates a `META-INF/MANIFEST.MF` file with required information (such as application name).
4. Compresses the contents of the temporary folder to `c:\temp\mymodule.jar`.

Signing Option Example

The following is an example of the `create` subcommand with the output option:

```
packager.bat create -o mymodule.jar -t web --sign \
          --keystore c:\mykeystore\c.keystore --storepass demo \
          --keypass mykey --alias jckey -c c:\mymodulefolder
```

in addition to those tasks described in the previous example, the Packager in this command line example signs the application using the keystore from `c:\mykeystore\c.keystore` by performing the following:

- Provides the password (`demo`) for the `mykeystore` keystore.
- Provides the password (`mykey`) for the private `c.keystore` key.
- Provides the name (`jckey`) required to retrieve the key from the keystore.

`validate` Subcommand

Validates an application module.

`validate` Subcommand Options

The `validate` subcommand has a single option, `-t` or `--type`, used to specify the type of application module or group to be validated. The type can be `web`, `extended-applet`,`classic-applet`, `classic-lib`, or `extension-lib`.

`validate` Subcommand Format

The following is an example of the `validate` subcommand format where *type* can be `web`, `extended-applet`, or `classic-applet`:

`packager.bat validate [--type` *type*`]` *module-file-name (or module-directory-name)*

`validate` Subcommand Example

The following is an example of the `validate` subcommand:

```
packager.bat validate -t web myapp.war
```

In this command line example, the Packager performs the following tasks:

1. Extracts the contents of `myapp.war` application module to a temporary folder.
2. Validates the contents of the descriptors.
3. Validates that the classes specified in the descriptors actually exist in the application module.
4. Cross validates the descriptors.
5. Displays results of validation.

`copyright` Subcommand

Displays the detailed copyright notice.

`copyright` Subcommand Options

There are no options for the `copyright` subcommand.

`copyright` Subcommand Format

The following is an example of the `copyright` subcommand format:

```
packager.bat copyright
```

`copyright` Subcommand Example

The following is an example of the `copyright` subcommand:

```
packager.bat copyright
```

`help` Subcommand

Prints information about using subcommands.

`help` Subcommand Options

While there are no options for the `help` subcommand, it does accept a topic attribute consisting of a specific subcommand name for which detailed information is displayed.

`help` Subcommand Format

The following is an example of the `help` subcommand format:

`packager.bat help` *subcommand*

`help` Subcommand Example

The following is an example of the `help` subcommand:

```
packager.bat help validate
```

Use Cases

TABLE 7-4 provides use cases for the command line arguments and describes the expected output for each.

TABLE 7-4 Use Cases for Command Line Arguments

Input	Expected Output
Valid arguments are passed for all specified types (`web`, `extended-applet`, `classic-applet`,`extension-lib`, or `classic-lib`), `-o` specified.	Valid application module of corresponding type is created.
Valid arguments are passed for all specified types (`web`, `extended-applet`, `classic-applet`,`extension-lib`, or `classic-lib`), `-o` not specified.	Packager performs xml validation. No application module is created.
The same name is specified for several application modules using the *filename* argument.	Error message and modules are renamed automatically.
`-f` is specified, descriptors contain unsupported tags.	Warns developer, cuts out unsupported tags.
`-s` is specified, valid signing related arguments passed.	Signs the resulting JAR file.

CHAPTER 8

Loading and Managing Applications

This chapter describes the use of the card installer in loading, creating, unloading, and deleting applications on a card. The card installer consists of two components, an on-card installer and an off-card Installer tool (Installer tool) provided by the Development Kit, that work in conjunction to provide card application management functions.

See Chapter 3 to better understand the role and relationship between the Installer tool and the other Development Kit tools used in developing and deploying applications for the Java Card 3 platform.

This chapter consists of the following sections:

Description of the On-Card Installer

The on-card installer is a ROMized servlet responsible for handling requests received from the off-card installer, extracting the command and data, forwarding them to the card manager. Upon the return of the card manager, the installer forms the response to send back to the off-card installer.

On-card Installer Operation

The on-card installer provides the interface between the Installer tool and the card manager and provides a request handling function for the card manager to perform card management tasks. The on-card installer assumes the `/cardmanager` context to represent the on-card card manager. All `/cardmanager/`*command* URIs (in which *command* represents load, create, delete, unload, or list) are mapped to one context `/cardmanager` assigned to the on-card installer.

The on-card installer parses and extracts the command, name, and data information in the multi-part POST requests. The information is passed on by calling the appropriate card manager's API. The on-card installer and the filter are registered and started with the web container at card initialization.

On-card Installer Functionality

The on-card installer provides the following functionality:

1. Handles requests received from the off-card installer.

   These requests include the command for card application management and the data (application module JAR file).

2. Extracts data (JAR file) contained in the HTTP request and saves it to an on-card file.

3. Passes the `load`, `create`, `delete`, `unload`, or `list` command, parameters and the location of the saved JAR file to the Card Manager.

4. Handles the return from the Card Manager.

5. Builds the response content and sending the response back to the off-card installer.

6. Can be configured to require PIN authentication of the off-card installer via basic HTTP authentication:
   - `load`, `create`, `delete`, or `unload` are protected with session-scoped authentication.
   - `list` is protected with global card holder authentication.
   - `load`, `create`, `delete`, or `unload` require card holder authorization.

Description of the Installer Tool

The Installer tool works on behalf of the on-card installer to perform various card management tasks such as deploying an application and listing all applications. The communication between the Installer tool and on-card installer is proprietary. For the RI, HTTP POST is used as the communication protocol.

The following is the list of functionalities required by the Installer tool:

- Loads an application module onto the card.
- Creates an instance of an application.
- Deletes (deactivates) an instance of an application.
- Completely removes a module or application from the card.
- Displays information about loaded applications and instances.

Running the Installer Tool

The Installer tool is a command-line tool, implemented using Java SE. The command line interface for the Installer tool has the following syntax:

`installer.bat` *subcommand* `[options] [arguments]`

In the command line, the subcommand must be the first argument after the `installer.bat` command. Options and arguments can be in any order.

In the command line, subcommands and options can be specified in either a short form or a long form. The short form is a single character preceded by a hyphen (-). The long form uses a meaningful name preceded by two hyphens (--). Each subcommand can take one or more options or arguments that must follow the subcommand but can be in any order. For example, `-i` *instance-name* or `--instance` *instance-name*.

Arguments are command line arguments that are not bound to an option. For example, an application or module file name used in the `load` command is an argument.

The following is a list of the available subcommands for the `installer.bat` command:

- `load` Subcommand
- `create` Subcommand
- `delete` Subcommand
- `unload` Subcommand

- `list` Subcommand
- `help` Subcommand

`load` Subcommand

Causes the Installer tool to load a specified application module or library file. The `load` subcommand can have one or more options and arguments.

`load` *Subcommand Options*

TABLE 8-1 lists and describes the available `load` subcommand options.

TABLE 8-1 `load` Options

Option	Description
`-c` *oncardinstaller-url* or `--cardmanager` *oncardinstaller-url*	Specifies the location of the on-card installer where *oncardinstaller-url* represents the complete URL of the on-card installer.
`-n` *module-or-library-name* or `--name` *module-or-library-name*	Specifies the name of the module or library on the card, where *module-or-library-name* represents the module or library name.
`-p` *password* or `--password` *password*	Optional. Used when authentication is required. Specifies the password for the user set by the `--user` or `-u` subcommand, where *password* represents the required user password.

TABLE 8-1 `load` Options

Option	Description
`-s` *signature-file* or `--signature` *signature-file*	Specifies the name of the properties file that contains the BASE64 encoded certificate and signature, where *signature-file* represents the file name. This file is a simple properties file with properties: `signature=`*base64-encoded-signature* `certificate=`*certificate-to-validate-the module-and-digest*
`-t` *file-type* or `--type` *file-type*	Specifies the type of file being loaded, where *file-type* represents one of the following values: • `web` • `classic-applet` • `extended-applet` • `classic-lib` • `extension-lib`
`-u` *user-id* or `--user` *user-id*	Optional. Used when authentication is required to access the card manager. Specifies the user name, where *user-id* represents the user name.

`load` *Subcommand Arguments*

Command line arguments available for the `load` subcommand are the module or application group file name.

`load` *Subcommand Format*

The following is an example of the `load` subcommand format:

```
installer.bat load -c oncardinstaller-url -s signature-file -t file-type \
           -n module-or-library-name [-u user-id -p password] \
           application-module (or library-file)
```

`load` *Subcommand Example*

In the following example, the Installer loads the file `Calculator.war` with the name `calc`.

```
installer.bat load -c http://localhost:8019/cardmanager \
          -s mysig.properties -n calc -t web Calculator.war
```

`create` Subcommand

Causes the Installer to create an instance of an application from a specified group with a specified context. The `create` subcommand can have one or more options but has no arguments.

`create` *Subcommand Options*

TABLE 8-2 lists and describes the available `create` subcommand options.

TABLE 8-2 `create` Options

Option	Description
`-a` *applet-name-or-id* (or) `--applet` *applet-name-or-id*	Specifies the name of the applet loaded by `load` command, where *applet-name-or-id* represents the applet name.
`-c` *oncardinstaller-url* or `--cardmanager` *oncardinstaller-url*	Specifies the location of the on-card installer, where *oncardinstaller-url* represents the complete URL.
`-d` *install-parameters* or `--data` *install-parameters*	Optional. Install parameters (printable hex string) that will be passed to the install method of a classic or extended applet.
`-i` *name* or `--instance` *name*	Specifies the name or ID of the instance, where *name* represents the name or ID. For web applications, a context name used to create the web application. If none is specified, then the default *Web-Context-Path* from JCRD is used.

TABLE 8-2 `create` Options *(Continued)*

Option	Description
`-n` *module-or-library-name* or `--name` *module-or-library-name*	Specifies the name of the module or library loaded by `load` command, where *module-or-library-name* represents the module or library name.
`-p` *password* or `--password` *password*	Optional. Used when authentication is required. Sets the password for the user specified by the `--user` or `-u` subcommand.
`-u` *user-id* or `--user` *user-id*	Optional. If authentication is required to access the card manager, specifies the authorized user, where *user-id* represents the required user name.

`create` *Subcommand Arguments*

There are no command line arguments for the `create` subcommand.

`create` *Subcommand Format*

The following is an example of the `create` subcommand format:

```
installer.bat create -c oncardinstaller-url -n module-or-library-name \
           [-a applet-name-or-id] [-d install-parameters] [-i name] \
           [-u user-id -p password]
```

`create` *Command Example 1*

The following example assumes that a module was previously loaded and named `calc`. See "`load` Subcommand Example" on page 81. The Web-Context-Path in RD is `/Calculator`.

This example of the `create` command registers the web application with a web container using `/Calculator` as the context. Users access this web application by using `http://cardip:cardport/Calculator`.

```
installer.bat create -c http://localhost:8019/cardmanager -n calc
```

`create` *Command Example 2*

Similar to Command Example 1, the following example assumes that a module was previously loaded and named `calc`, with the exception that instead of using the default `/Calculator`, the application is registered with a web-container using the context `/Mycalc`.

```
installer.bat create -c http://localhost:8019/cardmanager -n calc \
          -i /MyCalc
```

`create` *Command Example 3*

Similar to Command Example 2, the following example assumes that a module was previously loaded and named `calc`, with the exception that the application is registered as an applet instead of a web-container and has an instance ID of `/01`.

```
installer.bat create -c http://localhost:8019/cardmanager -n calc \
          -a //aid/A000000062/03010C0201 -d a000f0 -i /01
```

`delete` Subcommand

Causes the installer to delete an instance that was created by the `create` subcommand. The `delete` subcommand can have one or more options but no arguments.

`delete` *Subcommand Options*

TABLE 8-3 lists and describes the available `delete` subcommand options.

TABLE 8-3 `delete` Options

Option	Description
`-c` *oncardinstaller-url* or `--cardmanager` *oncardinstaller-url*	Specifies the location of the on-card installer, where *oncardinstaller-url* represents the complete URL.
`-i` *name* or `--instance` *name* or `-i` *name;name1;name2; ...* or `--instance` *name;name1;name2; ...*	Specifies the instance of the application or multiple instances of applications to be deleted, where *name* represents the instance name of the application.
`-p` *password* or `--password` *password*	Optional. Used when authentication is required. Sets the password for the user specified by the `--user` or `-u` subcommand.
`-u` *user-id* or `--user` *user-id*	Optional. If authentication is required to access the card manager, specifies the authorized user, where *user-id* represents the required user name.

`delete` *Subcommand Arguments*

There are no command line arguments for the `delete` subcommand.

`delete` *Subcommand Format*

The following is an example of the `delete` subcommand format:

```
installer.bat delete -c oncardinstaller-url -i name [-u user-id -p password]
```

`delete` *Command Example*

In the following example, the installer deletes the instance `/MyCalc`.

```
installer.bat delete -c http://localhost:8019/cardmanager -i /MyCalc
```

`unload` Subcommand

Causes the installer to unload (remove) the specified module or application from the card including all instances created by the `create` command. The `delete` subcommand can have one or more options but no arguments.

`unload` *Subcommand Options*

TABLE 8-4 lists and describes the available `unload` subcommand options.

TABLE 8-4 `unload` Options

Option	Description
`-c` *oncardinstaller-url* or `--cardmanager` *oncardinstaller-url*	Specifies the location of the on-card installer, where *oncardinstaller-url* represents the complete URL.
`-n` *module-or-library-name* or `--name` *module-or-library-name*	Specifies the name of the module or library loaded by `load` command, where *module-name* represents the module or library name.

`-f`
or
`--force`
	Optional. Forces an attempt to delete any instances before unloading.

`-p` *password* or `--password` *password*	Optional. Used when authentication is required. Sets the password for the user specified by the `--user` or `-u` subcommand.
`-u` *user-id* or `--user` *user-id*	Optional. If authentication is required to access the card manager, specifies the authorized user, where *user-id* represents the required user name.

`unload` *Subcommand Arguments*

There are no command line arguments for the `unload` subcommand.

`unload` *Subcommand Format*

The following is an example of the `unload` subcommand format:

```
installer.bat unload -c oncardinstaller-url -n module-or-library-name [-f] \
          [-u user-id -p password]
```

`unload` *Command Example*

In the following example, the installer completely removes the `calc` module and all of its instances.

```
installer.bat unload -c http://localhost:8019/cardmanager -n calc
```

`list` Subcommand

Causes the installer to display summary or detailed information about loaded application modules, instances, and libraries.

`list` *Subcommand Options*

TABLE 8-5 lists and describes the available `list` subcommand options.

TABLE 8-5 `list` Options

Option	Description
`-c` *oncardinstaller-url* or `--cardmanager` *oncardinstaller-url*	Specifies the location of the on-card installer, where *oncardinstaller-url* represents the complete URL.

`-d`
or
`--detailed`
	Optional. Displays complete details of the application-modules, instances, and libraries.

`-p` *password* or `--password` *password*	Optional. Used when authentication is required. Sets the password for the user specified by the `--user` or `-u` subcommand.
`-u` *user-id* or `--user` *user-id*	Optional. If authentication is required to access the card manager, specifies the authorized user, where *user-id* represents the required user name.

`list` *Subcommand Arguments*

There are no command line arguments for the `list` subcommand.

`list` *Subcommand Format*

The following is an example of the `list` subcommand format:

```
installer.bat list -c oncardinstaller-url [-d] [-u user-id -p password]
```

`list` *Command Example 1*

In the following example, the installer displays summary information about modules, applications, and libraries.

```
installer.bat list -c http://localhost:8019/cardmanager
```

`list` *Command Example 2*

In the following example, the installer displays detailed information about modules, applications, and libraries.

```
installer.bat list -d -c http://localhost:8019/cardmanager
```

`help` Subcommand

Causes the installer to display summary or detailed information about one or more installer subcommands.

`help` *Subcommand Options*

There are no command line options for the `help` subcommand.

`help` *Subcommand Arguments*

Command line arguments for the `help` command are optional and consist of the name of the subcommand for which detailed help is requested.

`help` *Subcommand Format*

The following is an example of the `list` subcommand format:

```
installer.bat help [subcommand]
```

`help` *Command Example 1*

In the following example, the installer displays summary help about all of its subcommands.

```
installer.bat help
```

`help` *Command Example 2*

In the following example, the installer displays detailed help about the `load` subcommand.

```
installer.bat help load
```

Card Installer Use-Case

The following use case, Load an Application, illustrates a common use of the card installer.

Load an Application

This use case loads an application module to the card.

Pre-Conditions

The following preconditions must be satisfied for this use case:

- A valid module file available (in this use case, `mymodule.war`).
- A signature details file containing Base64 encoded signature and certificate is available (in this use case, `sig.properties`).
- The on-card installer application must be accessible to the off-card installer client via an http connection (in this use case, `http://localhost:8019/cardmanager`).

Post-Conditions

The module is loaded and ready to be created.

Sequence of Events

1. The user executes the following command:

```
installer.bat load -c http://localhost:8019/cardmanager \
          -s sig.properties -n app1 mymodule.war
```

2. The off-card installer connects to the on-card installer servlet and POSTs the required information.
3. A message is displayed on the console with the success information.

CHAPTER 9

Backwards Compatibility for Classic Applets

This chapter describes how to generate application modules for classic applets by using the Normalizer tool (Normalizer). These application modules contain classic CAP files and provide backwards compatibility for the Java Card 3 platform by enabling classic applets to run on Connected Edition and Classic Edition cards.

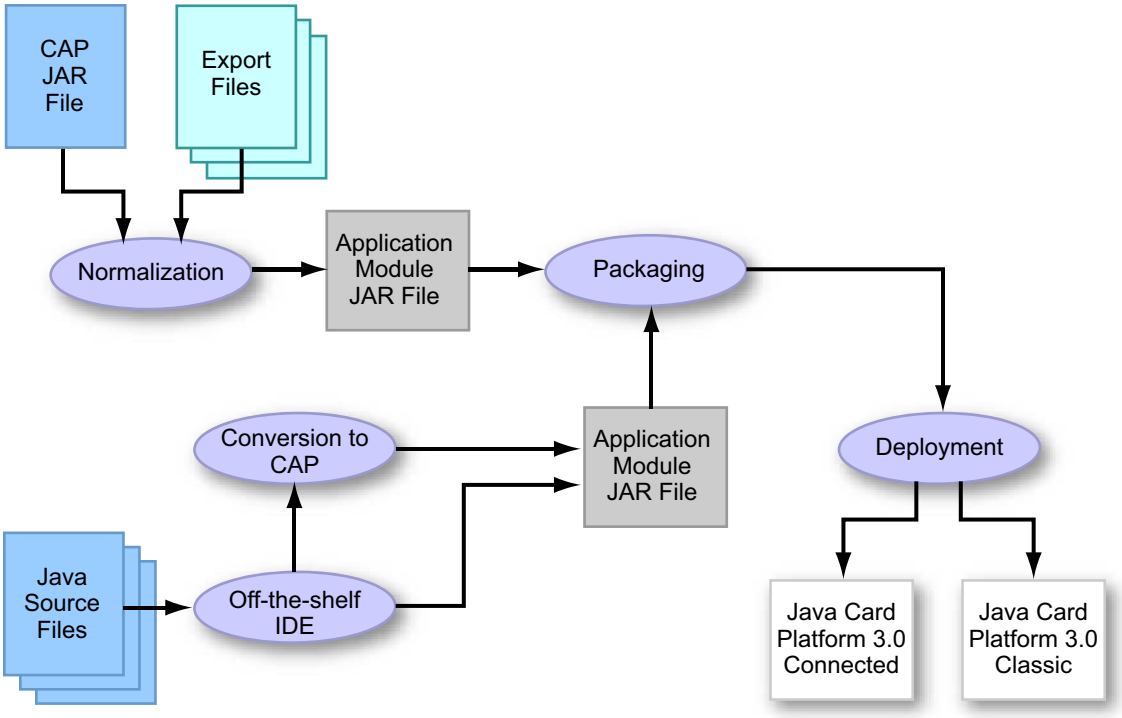
This chapter contains the following sections:

Generating Application Modules From Classic Applets

Developers use the Normalizer to generate application modules for applets created for previous version of the Java Card platform. The Normalizer can generate application module from existing modules when there is no source is available. These application modules contain CAP files and are downloadable on both the Java Card 3 platform Classic Edition and Connected Edition cards.

The output from the tool is a classic module that contains the class files, the CAP components of the CAP file, SIO proxies for classic SIOs (if used), and associated classic application descriptors. The input to the tool must be classic CAP files and associated EXP files. If the input files are not classic CAP files, the normalization will fail. See Appendix A for a description of the application module and library formats supported by the Java Card 3 platform card manager. FIGURE 9-1 illustrates the process of generating application modules from classic applets and deploying them on both the Java Card 3 platform Classic Edition and Connected Edition cards.

FIGURE 9-1 Process of Generating Application Modules From Classic Applets



Running the Normalizer

The command line interface for the Normalizer has the following syntax:

`normalizer.bat` *subcommand* `[options]`

The following is a list of the subcommands for the Normalizer:

- `normalize` - Creates the package class files
- `copyright` - Displays detailed copyright notice
- `help` - Displays information about the Normalizer command

`normalize` Subcommand

Use the `normalize` subcommand and its options to create the package class files. Options are used with the `normalize` subcommand to specify input files, export paths, export file names, and output directories.

`normalize` *Subcommand Options*

TABLE 9-1 identifies the `normalize` subcommand options and provides their descriptions.

TABLE 9-1 `normalize` Subcommand Options

Option	Description
`-i` *file* or `--in` *file*	Specifies the input CAP file name.
`-p` *path* or `--exportpath` *path*	Specifies the path of the export files used by the tool.
`-f` *file* or `--exportfile` *file*	Specifies the name of the export file.
`-o` *directory* or `--out` *directory*	(Optional) This the default setting and does not have to be explicitly set. Specifies the output directory that contains the export file.

`normalize` *Subcommand Format*

The following is the format of the `normalize` subcommand. Options in the subcommand are used in the sequence that are presented in TABLE 9-1. In this format example, an input file and an output directory are specified as options:

```
normalizer.bat normalize  --in file --exportpath path --out directory
```

`normalize` *Subcommand Example*

The following is an example of the `normalize` subcommand in which an input file (`myCAP.cap`) is specified as an option:

```
normalizer.bat normalize -i myCAP.cap
```

`copyright` Subcommand

The `copyright` subcommand displays the detailed copyright notice. There are no options associated with this subcommand.

`help` Subcommand

The `help` subcommand displays information about the Normalizer command. Options are used with the `help` subcommand to specify the information that is displayed about each sub-command.

Normalizer Summary Help

The following command displays summary help about the Normalizer:

```
normalizer.bat help
```

`normalize` *Subcommand Help*

The following command displays help about the `normalize` subcommand:

```
normalizer.bat help normalize
```

Converting Class Files to CAP Files

This section describes using the Converter tool (Converter) provided for the Connected Edition as a stand-alone tool. When run as a stand-alone tool, the Converter can take class files from `javac` and convert them into CAP files that can be loaded by the Connected Edition platform.

Note – If you are developing a classic applet application you want to deploy using the classic development kit, create your CAP file as described in this chapter. Then take your CAP file to the classic development kit to deploy it on the classic Java Card VM.

The Converter is part of the Developer Kit tool chain and is also used by the Normalizer to create application modules for classic applets. The Normalizer can generate application module from existing modules when there is no source is available. See Chapter 9 for a description of using the Normalizer to create application modules from classic applets.

The CAP file is a JAR-format file which contains the executable binary representation of the classes in a Java package. The CAP file also contains a manifest file that provides human-readable information regarding the package that the CAP file represents. For more information on the CAP file and its format, see Chapter 6 of the Virtual Machine Specification, Java Card Platform, Version 3.0.1, Connected Edition.

When running the Converter as a stand-alone tool, developers can use the command line options described in TABLE 9-2 to:

- Specify the root directory where the Converter looks for classes.
- Specify the root directories where the Converter looks for export files.
- Use the token mapping from a pre-defined export file of the package being converted. The Converter will look for the export file in the export path.
- Set the applet AID and the class that defines the install method for the applet.
- Specify the root directories where the Converter outputs files.
- Specify that the Converter output one or more of the following:
  - CAP file
  - JCA file
  - EXP export file
- Identify that the package is used as a mask.

  When a package is used as a mask, restrictions on native methods are relaxed.
- Specify support for the 32-bit integer type.
- Enable generation of debugging information.
- Turn off verification (the default of input and output files. Verification is default.

When the Converter runs, it performs the conversion process in the following sequence:

- **Loads the package** - If the `exportmap` option is set, the converter loads the package from the export path (see "Specifying an Export Map" on page 96). Loads the class files of the Java package and creates a data structure to represent the package.
- **Subset checking** - Checks for unsupported Java features in class files.
- **Conversion** - Checks for consistency between the applet AIDs and the imported package AIDs.

- **Reference Checking** - Checks that all references are valid, internal referenced items are defined in the package, import items are declared in the export files (see "Loading Export Files" on page 97).

  The Converter creates the `JcImportTokenTable` to store tokens for import items (class, methods, and fields). If the Converter only generates an export file, it does not check private APIs and byte code. Also included is a second round of subset checking that operations do not exceed the limitations set by the *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Connected Edition*.

- **Optimization** - Optimizes the bytecode.
- **Generates output** - Builds and outputs the EXP export file and the JCA file, checks the package version in the export file of the current package against the package version specified in the command line. If the `-exportmap` option is used in the command line, the export file specified in the command line must represent the same version as that of the package. The converter does not support upgrading the export file version.

  Before writing the export and JCA files, the Converter determines the output file path. The Converter assumes the output files are written into the director: *root_dir*`\`*package_dir*`\javacard`. By default the *root_dir* is the classroot directory specified by `-classdir` option. Users can specify a different *root_dir* by using `-d` option.

Specifying an Export Map

You can request that the Converter convert a package using the tokens in a pre-defined export file of the package being converted. Use the `-exportmap` command option to do this. The Converter loads the pre-defined export file in the same way that it loads other export files.

There are two distinct cases when using the `-exportmap` flag is desired:

- When the minor version of the package is the same as the version given in the export file (this case is called package reimplementation).

  During package reimplementation, the API of the package (exportable classes, interfaces, fields and methods) must remain exactly the same.

- When the minor version increases (package upgrading).

  During a package upgrade, changes that do not break binary compatibility with preexisting packages are allowed (See "Binary Compatibility" in Section 4.4 of the Virtual Machine Specification, Java Card Platform, Version 3.0.1, Connected Edition).

For example, you must use the `-exportmap` option to preserve binary compatibility with already existing packages that use the package when reimplementing a method (package reimplementation) in an existing package or upgrading an existing package by adding new API elements (new exportable classes or new public or protected methods or fields to already existing exportable classes).

Loading Export Files

A Java Card technology-based export file (export file) contains the public API linking information 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. An applet package is linked with the `java.lang`, the `javacard.framework` and `javacard.security` packages via their 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 platform's directory naming convention.

For example, to load the export file of the package `java.lang`, if you specify `-exportpath` as `c:\myexportfiles`, the Converter searches the directory `JCDK3.0.1_ConnectedEdition\api_export_files\javalang\javacard` for the export file `lang.exp`.

Creating a `debug.msk` Output File

If you select the `-mask` and `-debug` options, the file `debug.msk` is created in the same directory as the other output files. (Refer to "`converter` Command Options" on page 99.)

Verification of Input and Output Files

By default, the converter invokes the off-card verifier for every input EXP file and on the output CAP and EXP files.

- If any of the input EXP files do not pass verification, then no output files are created.
- If the output CAP or EXP files do not pass verification, then the output EXP and CAP files are deleted.

If you want to bypass verification of your input and output files, use the `-noverify` command line option. Note that if the converter finds any errors, output files will not be produced.

File and Directory Naming Conventions

This section describes the naming conventions used for the input and output files of the Converter, and gives the correct location for these files. With some exceptions, the Converter follows the Java programming language naming conventions for default directories for input and output files. These naming conventions are also in accordance with the definitions in Section 4.1 of the Virtual Machine Specification, Java Card Platform, Version 3.0.1, Connected Edition.

Input File Naming Conventions

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 class files for a package must be located in the same directory under the root directory, following the Java programming language 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.

For example, the following command line would be used to convert the package `java.lang`, use the `-classdir` flag to specify the root directory as `C:\mywrk`:

```
converter -classdir C:\mywrk java.lang package_AID package_version
```

In the example, *package_AID* is the application ID of the package and *package_version* is the user-defined version of the package. The Converter will look for all class files in the `java.lang` package in the directory `C:\mywrk\java\lang`.

Output File Naming Conventions

The name of the CAP file, export (EXP) file, and the Java Card Assembly (JCA) file must be the last portion of the package specification followed by the extensions `.cap`, `.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 previous example, the output files are written by default to the directory `C:\mywrk\java\lang\javacard`.

The `-d` flag is used to specify a different root directory for output.

In the previous example, using the `-d` flag to specify the root directory for output to be `C:\myoutput` would cause, the Converter to write the output files to the directory `C:\myoutput\java\lang\javacard`.

When generating a CAP file, the Converter creates a JCA file in the output directory as an intermediate result. If you do not want a JCA file to be produced, do not use the option `-out JCA`. The Converter deletes the JCA file at the end of the conversion when the option `-out JCA` is not used.

Running the Converter

The command line interface for running the Converter takes one of the following forms:

`converter.bat` *options package_name package_aid major_version*`.`*minor_version*

or

`converter.bat -config` *filename*

Use the `-config` subcommand and the associated configuration file to provide the options and parameters to the Converter. See "Using a Command Configuration File" on page 101.

`converter` Command Options

Use the `converter` command options to specify input files, an export path, an export map, names, and output directories.

TABLE 9-2 identifies the `converter` command options and provides their description.

TABLE 9-2 `converter` Command Options

Option	Description
`-classdir` *root- directory-of-class-hierarchy*	Specifies the root directory where the Converter looks for classes.
`-i`	Specifies support the 32-bit integer type.
`-exportpath` *list-of-directories*	Specifies the root directories where the Converter looks for export files.
`-exportmap`	Uses 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*	Sets the applet AID and the class that defines the install method for the applet.
`-d` *root-directory-for-output*	Specifies the root directories where the Converter outputs the files.
`-out [CAP] [EXP] [JCA]`	Specifies that the Converter output the `CAP` file, and/or the `JCA` file, and/or the `EXP` export file.
`-V` or `-version`	Displays the Converter version number.
`-v` or `-verbose`	Enables verbose output.
`-help`	Displays the contents of this table.
`-nowarn`	Instructs the Converter to not report warning messages.
`-mask`	Identifies this package is used for a mask. Restrictions on native methods are relaxed.
`-debug`	Enables generation of debugging information.
`-nobanner`	Suppresses standard output messages.
`-noverify`	Turns off verification. Verification is default.
`-sign`	Signs the output CAP file.
`-keystore` *keystore*	Specifies the keystore to use in signing.

TABLE 9-2 `converter` Command Options *(Continued)*

Option	Description
`-storepass` *storepass*	Specifies the keystore password.
`-alias` *alias*	Specifies the keystore alias to use in signing.
`-passkey` *passkey*	Specifies alias password.

Using a Command Configuration File

Instead of entering all of the command line arguments and options on the command line, you can include them in a text-format configuration file. This is convenient if you frequently use the same set of arguments and options.

The syntax to specify a configuration file is:

```
converter -config filename
```

The *filename* argument contains the file path and file name of the configuration file.

You must use double quote (") delimiters for the command line options that require arguments in the configuration file. For example, if the options from the command line example used in "Using Delimiters with Command Line Options" on page 101 were placed in a configuration file, the result would look like this:

```
-exportpath ".\export files;.;.\JC_CONNECTED_HOME\
api_export_files"
 MyWallet 0xa0:0x00:0x00:0x00:0x62:0x12:0x34 1.0
```

Using Delimiters with Command Line Options

If the command line option argument contains a space symbol, you must use delimiters with this argument. The delimiter is a double quote (" ").

In the following sample command line, the Converter will check for export files in the `.\export` files, `.\`*JC_CONNECTED_HOME*`\api_export_files`, and current directories.

```
converter -exportpath ".\export files;.;.\JC_CONNECTED_HOME\
api_export_files"
 MyWallet 0xa0:0x00:0x00:0x00:0x62:0x12:0x34 1.0
```

CHAPTER 10

Using the APDU Tool

When installing and running applets on a Java Card technology-compliant smart card, the APDU tool reads a script file containing Application Protocol Data Unit (APDU) commands and sends them to the Java Card runtime environment. Each APDU is processed and returned to the APDU tool, which displays both the command and response APDU commands on the console. Optionally, the APDU tool can write this information to a log file.

This chapter includes the following sections:

Running the APDU Tool From the Command Line

The file used to invoke the APDU tool is the `apdutool.bat` batch file.

Unless otherwise specified, the APDU tool starts listening to APDU commands in the default format of T=1 on the TCP/IP port specified by either the `-p` *`portNumber`* parameter (for contacted) or the `-p` *`portNumber`*`+1` parameter (for contactless). The default port is 9025.

The command line usage for the APDU tool is:

```
apdutool.bat [-h hostname] [-nobanner] [-noatr] [-o outputFile]
 [-p portNumber] [-s serialPort] [-t0]
 [-version] [inputFile ...] [-verbose]
```

The option values and their actions are shown in TABLE 10-1.

TABLE 10-1 `apdutool` Command Line Options

Option	Description
`-h` *hostname*	Specifies the host name on which the TCP/IP socket port is found. (See the `-p` option.)
`-help`	Displays online documentation for this command. To get help for the APDU tool, run `bin/apdutool.bat -help` on the command line.
`-noatr`	Suppresses outputting an ATR (answer to reset).
`-nobanner`	Suppresses all banner messages.
`-o` *outputFile*	Specifies an output file. If an output file is not specified with the `-o` option, output defaults to standard output.
`-p` *portNumber*	Specifies a TCP/IP socket port other than the default port of 9025.
`-s` *serialPort*	Specifies the serial port to use for communication, rather than a TCP/IP socket port. For example, *serialPort* can be `COM1`. To use this option, the `javax.comm` package must be installed on your system. If you enter the name of a serial port that does not exist on your system, the APDU tool will respond by printing the names of available ports.
`-t0`	Runs T=0 single interface.
`-version`	Outputs the version information.
inputFile	Allows you to specify the input script (or scripts).
`-verbose`	Displays descriptive text during operation.

Examples of Using the APDU Tool

The following examples show how to use the APDU tool to direct output to the console or to a file.

Directing Output to the Console

The following command runs the APDU tool with the file `example.scr` as input. Output is sent to the console. The default TCP port (9025) is used.

```
apdutool.bat example.scr
```

Directing Output to a File

The following command runs the APDU tool with the file `example.scr` as input. Output is written to the file `example.scr.out`.

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

Using APDU Script Files

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

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

C-APDU syntax for the APDU tool 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. 1 byte in non-extended mode, 2 bytes in extended mode.
- *byte 0 ... byte LC-1* - Input data bytes.
- *LE* - ISO 7816- 4 expected output length. 1 byte in non-extended mode, 2 bytes in extended mode.

The script file commands shown in TABLE 10-2 are supported:

TABLE 10-2 Supported APDU Script File Commands

Command	Description
`contacted;`	Redirects APDU activity to the contacted or primary interface.
`contactless;`	Redirects output to the contactless or secondary interface.
`delay` *`Integer`*`;`	Pauses execution of the script for the number of milliseconds specified by *`Integer`*.
`echo` *`"string"`*`;`	Echoes the quoted string to the output file. The leading and trailing quote characters are removed.
`extended on;`	Turns extended APDU input mode on.
`extended off;`	Turns extended APDU input mode off.
`output off;`	Suppresses printing of the output.
`output on;`	Restores printing of the output.
`powerdown;`	Sends a `powerdown` command to the reader in the active interface.
`powerup;`	Sends a `powerup` command to the reader in the active interface. A `powerup` command must be sent to the reader prior to executing any APDU on the selected interface.

These packages provide a convenient API for writing client-side applications that communicate with Java Card technology enabled smart cards.

CHAPTER 11

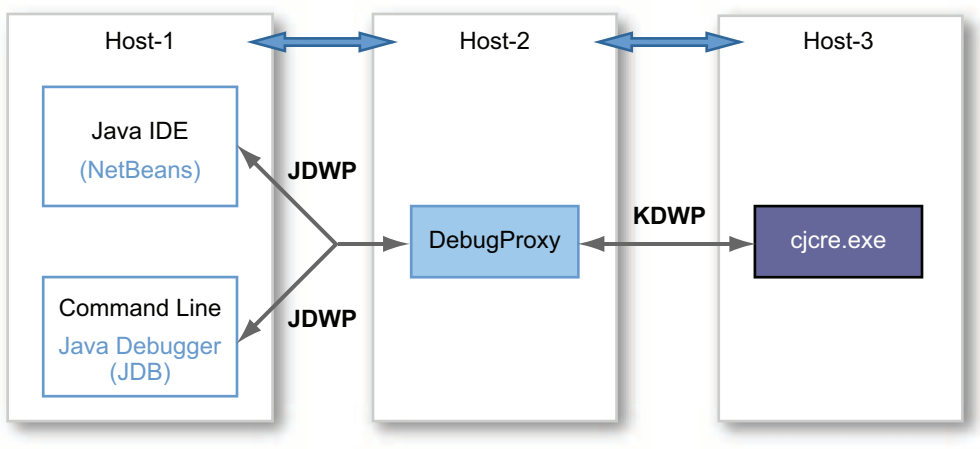
Debugging Applications

This chapter describes the Debugger tool (Debugger) for Java Card 3 platform application developers and how to use it as a separate tool with any Java enabled IDE. Using this tool, developers can debug their applications in any Java enabled IDE.

Debugger Architecture

The following diagram illustrates the debugger architecture for `cjcre`.

FIGURE 11-1 Debugger Architecture



The Java Debug Wire Protocol (JDWP) used by the IDE is heavy for a small VM such as that provided by `cjcre`. Consequently, `cjcre` uses KVM Debug Wire Protocol (KDWP) to provide a minimum set of debugging capabilities. The `debugproxy` translates and sends the translated JDWP commands from the IDE to `cjcre` in KDWP format. Responses from `cjcre` are converted into JDWP format by `debugproxy` before it sends them to the IDE.

The communication between `cjcre`, `debugproxy`, and the IDE happens through sockets. Socket based communication enables developers to debug `cjcre` from remote hosts. For example, `cjcre` could run on *machine1*, `debugproxy` could run on *machine2*, and the IDE could run on *machine3*. Developers can also run `cjcre`, `debugproxy`, and the IDE on same host.

Ports used by IDE communication to and from `debugproxy` and `debugproxy` communication to and from `cjcre` are distinguished by the names "**listen** port" and "**remote** port" respectively.

Using the Debugger

To fully utilize the capabilities of the Debugger, the application's class files must be compiled with debug info. This is done by specifying the `-g` flag for `javac` when compiling the source files. These class files must be available to the `debugproxy`, so that the line number information can be retrieved while stepping through the code.

▼ Debug a Java Card 3 Platform Application

1. Compile the source code with `-g` option.

All source files must be compiled using `-g` option to generate the debug information in the class files. If the `-g` option is not used, it is not possible to set breakpoints in the source code.

2. Start `debugproxy` (`debugproxy.bat`) from a command line window.

The `debugproxy` needs to know the location of class files being debugged. When starting `debugproxy`, include `-c` (or `--classpath`) option in the command to specify the path of the class files to be debugged.

The following is an example of a command that starts `debugproxy` and specifies `myapp.war` as the location of the class files to be debugged:

```
debugproxy.bat -c myapp.war
```

See Appendix C for additional details about debugger command line options.

3. **Attach to the debugger from the IDE.**

This procedure depends on the IDE used. For procedures required to attach the debugger to an IDE, refer to the documentation provided with the IDE.

4. **Start the `cjcre.exe` with `-debugger` option.**

   This option enables the debugger functionality in `cjcre`. Without this option, debugger functionality is disabled in `cjcre`.

5. **Set break points in the the application source code.**

   This procedure depends on the IDE used. The following steps are typical for all IDEs. Refer to the documentation provided with the IDE specific instructions.

   a. **Display the source code of the application in the IDE.**

   b. **With the source code displayed in the IDE, open any file and set break points where required.**

      Break points can be set at any time, even before attaching the debugger.

6. **Step through the code by executing the application from within the IDE.**

   When a break point is hit, the IDE stops execution and highlights the current line. Depending on the IDE being used, there are various options available to developers for stepping over or stepping into the code.

Note – Various IDE windows are available to monitor items such as local variables and threads. Refer to the documentation provided by the IDE for additional information about the windows used in monitoring debugger and application execution.

Configuring the Debugger

Various command line options are available to configure the `debugproxy` and the `cjcre`. See Appendix C for additional details about debugger and `cjcre` command line options.

PART II Programming With the Development Kit

This part of the user's guide provides solutions for various programming issues.

CHAPTER 12

Configuring the RI

This chapter describes the options used to configure a custom RI. This chapter is useful only if you have a source release of the develoment kit. For real cards, there are a few items such as Protection Domains and Certificates that must be setup at manufacturing time. The RI provides a means of configuring some factory settings by using the `config.properties` file under the `lib` folder.

This chapter contains the following sections:

Configuring Authenticators

In the `lib/config.properties` file, the following properties must be added to add an authenticator:

- `authenticator.`*`index`*`.uri`
- `authenticator.`*`index`*`.factory`
- `authenticator.`*`index`*`.pin`
- `authenticator.`*`index`*`.digest`

The following items describe the contents of the preceeding list of properties:

- *index* is a zero based number. At startup, the RI starts reading these properties beginning with index zero and creates authenticators until the sequence is broken.
- The `uri` property provides the SIO uri used for this authenticator.
- The `factory` property provides the factory class. For example, `com.sun.javacard.security.PINSessionAuthenticatorFactory`.

- The `pin` property provides the 4 digit pin number.
- The `digest` property provides `true` or `false` depending on if it should use `digest`.

Creating Custom Protection Domains

The Java Card 3 platform RI assigns a protection domain to an application based on the certificate used to sign the application bundle with the Packager tool. In the `lib/config.properties` file the following properties must be added to add a new protection domain:

- `pd.`*`pd-index`*`.certificate`
- `pd.`*`pd-index`*`.include.`*`include-index`*
- `pd.`*`pd-index`*`.exclude.`*`exclude-index`*

The following items describe the contents of the preceeding list of properties:

- All the indexes (*pd-index*, *include-index*, and *exclude-index*) are zero based numbers.
- The `certificate` property provides the BASE-64 encoded certificate.
- The include.*include-index* property provides a list of permissions that should be included for this protection domain.
- The exclude.*exclude-index* property provides a list of permissions that should be excluded for this protection domain.

Creating a Custom keystore

A custom ketstore can be crested by using the keytool to generate the certificates and private keys.

Using `keytool` *to Generate Certificates and Private Keys*

Enter the following `keytool` command and options on the command line:

```
keytool -genkey -alias alias -keyalg RSA
keytool -selfcert -alias alias
keytool -list -rfc
java DumpPrivateKey
```

This is how the `PolicyManager.java` certificate and key were generated. For scripting use the following `keytool` command:

```
keytool -keystore keystore -storepass keystore-password \
          -alias alias -keypass alias-password -genkey \
          -keyalg RSA -dname "cn=X, ou=U, o=O, c=US"
```

This `keytool` command runs in batch mode without prompting for input values.

Configuring SSL Support

In the `lib/config.properties` file, the following properties must be added to add an ssl support:

- `ssl.serverIdentity`
- `ssl.selfIdentityAsClient`
- `ssl.selfIdentityAsServer`
- `ssl.selfIdentitySSLPrivateKeyExp`
- `ssl.selfIdentitySSLPrivateKeyMod`
- `PSKIdentityHint=X509`

CHAPTER 13

Building the RI From Sources

This chapter describes how to build a customized Java Card 3 platform RI. This chapter is useful only if you have a source release of the develoment kit. The src folder under *JC_CONNECTED_HOME* contains all of the source files for the RI including VM code, and all tools (such as the packager and installer). You can modify or add to these files and build a cutomized Java Card 3 platform RI according to their specific requirements. The following actions are possible reasons a developer might have for building a custom RI:

- Add additional classes or packages if a proprietary API or other implementation classes are used.
- Fine tune the existing sources.
- Update tools to work with target platform.
- ROMize the applications. ROMizing masks the applications into the `cjcre.exe`.

This chapter contains the following sections:

Prerequisites to Building the RI

Before building the RI, the following software must be installed on the system:

- MinGW
- JDK 6
- ANT

See Chapter 2 for more details on the pre-requisites.

Contents of *JC_CONNECTED_HOME*\ `src` Folder

The following describes the contents of the `src` folder.

- **api** - Contains all of the `.java` files required to build a custom RI. If a new package must be added, it is added under this folder.
- **tools** - Contains the source code of all shipped tools organized in separate folders. To make a tool to work with a target platform, edit the code of the corresponding tool.
- **romized_apps** - Contains the source files for the CardManager.
- **vm/c** - Contains the source files of core VM.
- **vm/h** - Contains the header files of core VM.
- **vm/lib** - Contains configuration files `config.properties` and `system`. See Chapter 12 for additional details.
- **`vm/ignore.list`** - If a class must be excluded from ROMization, add its name in this file.
- **build.xml** - The main file used to build the tools and `cjcre.exe` in a single step.

Running the ROMizer Tool

When building a custom RI, the ROMizer tool takes system class files and application modules as input and creates a ROM image of these in an output ROM image file. The ROMizer tool converts the class files into C code, which is often called a ROM mask or simply a mask. For applications, the ROMizer tool stores non-class files in appropriate directories in the internal Java Card 3 platform filesystem, so that these files are available during the execution of the application. See "Building a Custom `cjcre.exe`" on page 120 for detailed description of using the ROMizer tool.

Either of the following commands will run the Romizer tool:

`romizer.bat -o` *ROM-output- filename* `-e` *EEPROM- filename* `-a` *apps-filename*

or

```
romizer.bat --out ROM-output- filename --e2pfile EEPROM -filename \
          --apps apps-filename
```

In the previous format examples, the following parameters are used:

- `-o` (or `--out`) - Must be followed by the path to output file where the mask is written. For example:

```
--out MyROMJava.c
```

  See "Romizer Tool Output" on page 120 for a description of the ROM output file.

- `-e` (or `--e2pfile`) - Must be followed by the path to the initial `eeprom` file. For example:

```
--e2pfile myeeprom.eeprom
```

- `-a` (or `--apps`) - Must be followed by the path to the applications list file which contains the list of applications to be masked. For example:

```
--apps myapps.list.
```

  See "Example Contents of Apps List File" on page 120 for a description of the configuration file.

Apps list File Contents

The apps list file contains information about applications that needs to be romized. All system classes and applications must be provided as input to the `romizer` in compressed files (`.jar`, `.war`, or `.zip` files).

Each application file must be specified in the apps list file on a new line. Each application module entry in the configuration file must provide additional information as noted in the following format example:

```
application-module -t <web|classic-applet|extended-applet|classic-lib| \
          extension-lib> -s signature-file -n module-name
```

In the previous example, the following parameters are used:

- *application-module* is the `.jar`, `.war`, or `.zip` application module file.
- `-t` followed by `web`, `classic-applet`, `extended-applet`, `classic-lib`, or `extension-lib` to identify the type of application being romized.
- `-s` followed by the name of the properties file that contains the BASE64 encoded certificate and signature, where *signature-file* represents the file name.

  This file is a simple properties file containing the following properties as name-value pairs:

  - `signature=`*base64 encoded signature*
  - `certificate=`*certificate to validate this group and digest*

- `-n` followed by the module name that will be referenced by `cjcre.exe` for this application module.

The following is an example of an entry in the configuration file:

```
HelloWorld.war -t web -s mykey1.txt -n helloapp
```

Example Contents of Apps List File

The following is an example of the contents of an apps list file:

```
HelloWorld.war -t web -s key1.txt -n helloapp
GCFClient.war -t web -s key2.txt -n gcfapp
```

Romizer Tool Output

The output created by running the Romizer tool is a preliminary EEPROM file and a C language source file that contains the ROM image of the input file including the following:

- Java class files that contain the API implementation
- Implementation of containers
- Applications selected by the user for romization

Building a Custom `cjcre.exe`

The `build.xml` provided in the src folder build everything including tools and `cjcre.exe`. This section gives details on how the `cjcre.exe` is generated.

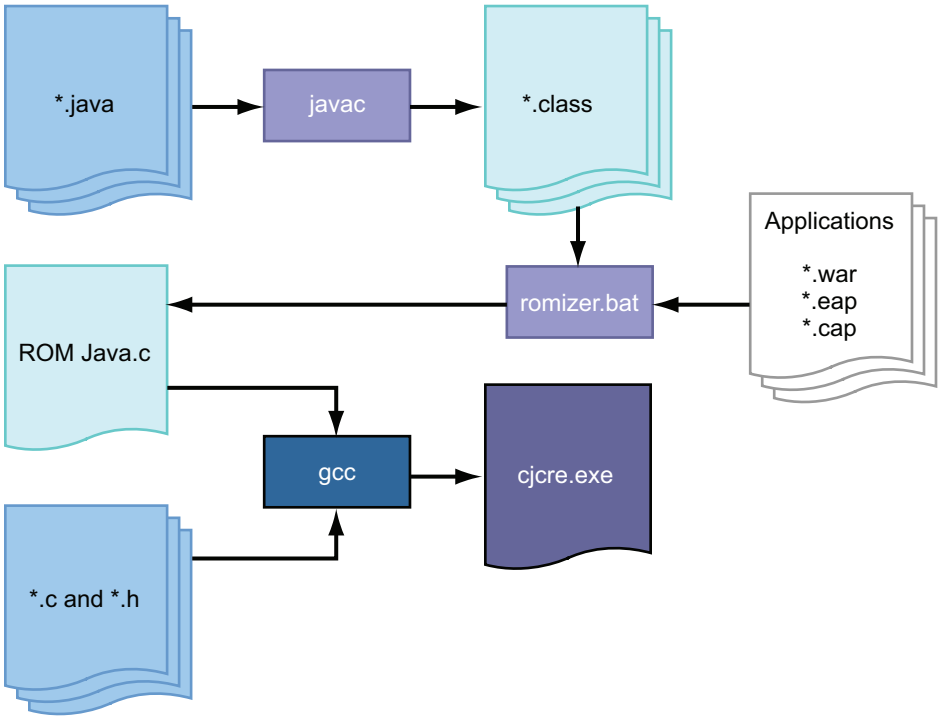
Developers can modify the RI by adding or modifying the reference implementation code and using the ROMizer tool. RI consists of `.java` and C source files. The core VM is written in C programming language and rest of the API and supported implementation is written in the Java programming language. The ROMizer tool converts the class files into C code, which is often called the ROM mask or simply the mask. Then all the C source code is compiled to an executable to generate `cjcre.exe`.

The ROM image can include any supported application files (web, extended-applet, classic-applet, extension library, and classic library). ROMized applications can be instantiated without requiring download after the runtime environment starts up. The ROMizer tool takes system class files and application module as input and creates a ROM image of these in an output ROM image file.

For applications, the ROMizer tool stores the non-class files in appropriate directories in the internal Java Card 3 platform filesystem, so that these files are available during the execution of the application.

The following diagram illustrates the procedure of building the `cjcre.exe` from sources.

EXAMPLE 0-1 Building `cjcre.exe` From Sources



Java files are compiled into class files using the `javac` compiler. Details of applications to be ROMized are listed in a text file. The class files and the list file are given as input to the `romizer.bat` tool (see "Preprocessor Symbols to Customize the VM" on page 122). By default `romizer.bat` generates `ROMJava.c`, a `C` file that contains the information about all classes and applications.

The GNU C compiler (`gcc`) is used to build the final executable. The generated `ROMJava.c` and the rest of the `C` files are compiled using `gcc`, which generates `cjcre.exe`. Use the provided ANT build file to build custom `cjcre.exe`. See "Build a Custom RI" on page 122.

Preprocessor Symbols to Customize the VM

The following preprocessor symbols can be used to customize the Java Card VM:

- `INCLUDEDEBUGCODE=0`
- `TRACE_EXCEPTIONS_NATIVE=1`
- `INCLUDE_FIREWALL_DEBUG_CODE=0`
- `ENABLE_JAVA_DEBUGGER=1`

  Enables the kdwp code. Default is 1. If set to 0, then cjcre can not be used to debug the applications.

- `APDU_PROTOCOL_T=0`

  Controls the protocol that will be supported by cjcre. default is t=0. Valid values are 0, 1 for T=0, T=1 respectively.

- `APDU_INTERFACE=0`

  Contacted or contactless or both. Valid values are 0, 1, 2 for contacted, contactless and Dual respectively

▼ Build a Custom RI

1. **Edit the files or add more files.**
2. **Update the tools source code if required.**
3. **From command line navigate to the `src` folder and run the `ant` command.**

   If there is a apps list file that contains the list of applications for ROMization, set the property apps_file_for_romizer while running the ant as shown below.

   ```
   ant -Dapps_file_for_romizer=path-to-apps-file
   ```

   The ant command creates the *JC_CONNECTED_HOME*`\custom_build` folder with a `bin` and `lib` folder under it.

   - The `bin` directory contains the new `cjcre.exe` and all of the other tool's `.bat` files.
   - The `lib` folder contains the `.jar` files and config files.

JC_CONNECTED_HOME`\custom_build\bin` and *JC_CONNECTED_HOME*`\custom_build\lib` are similar to *JC_CONNECTED_HOME*`\bin` and *JC_CONNECTED_HOME*`\lib`, except that `custom_build` contains the binaries from the updated source code.

▼ Test the Custom RI

- **Use the following command to run the new `cjcre.exe` file stored in *JC_CONNECTED_HOME*`\custom_build\bin`.**

JC_CONNECTED_HOME`\custom_build\bin\cjcre.exe [`*options*`]`

See Chapter 5 for a description of the available options.

Files created as a result of running or building the custom RI are stored in the *JC_CONNECTED_HOME*`\custom_build\bin` and *JC_CONNECTED_HOME*`\custom_build\lib` directories. These directories are created the first time the RI is built and will be over written every time the RI is built.

CHAPTER 14

Programming to the Java Card RMI Client-Side API

This chapter describes how to use the Java Card RMI client-side API. Support for the Java Card RMI client-side API is optional according to the Java Card specifcations, but it has been implemented in the Classic RI available in the Classic development kit. Note that it has not been implemented in the Connected RI available in the Connected development kit.

A Java Card RMI client application runs on a Card Acceptance Device (CAD) terminal that supports a Java SE or Java ME platform. The client application requires a portable and platform-independent mechanism to access the Java Card RMI server applet executing on the smart card.

The basic client-side framework is implemented in the package `com.sun.javacard.javacard.rmiclientlib and com.sun.javacard.javacard.clientlib`.

The library is located in the file `lib/tools.jar`.

The reference implementation of the Java Card RMI client-side API is based on APDU I/O for its card access mechanisms. For more information on APDU I/O, see Chapter 15.

Remote Stub Object

The Java Card RMI API supports two formats for passing remote references. The format for remote references containing the class name requires stubs for remote objects available to the client application.

The standard Java RMIC compiler tool can be used as the stub compilation tool to produce stub classes required for the client application. To produce these stub classes, the RMIC compiler tool must have access to all the non-abstract classes defined in the applet package which directly or indirectly implement remote interfaces. In addition, it needs to access the `.class` files of all the remote interfaces implemented by them.

If you want the stub class to be Java Card RMI-specific when it is instantiated on the client, it must be customized with a Java Card platform-specific implementation of the `CardObjectFactory` interface.

The standard Java RMIC compiler is used to generate the remote stub objects. JCRemoteRefImpl, a Java Card platform-specific implementation of the `java.rmi.server.RemoteRef` interface, allows these stub objects to work with the Java Card RMI API. The stub object delegates all method invocations to its configured RemoteRef instance.

The `com.sun.javacard.rmiclientlib.JCRemoteRefImpl` class is an example of a RemoteRef object customized for the Java Card platform.

For examples of how to use these interfaces and classes, see *Application Programming Notes, Java Card Platform, Version 3.0.1, Classic Edition*.

Note – Since the remote object is configured as a Java Card platform-specific object with a local connection to the smart card via the CardAccessor object, the object is inherently not portable. A bridge class must be used if it is to be accessed from outside of this client application.

Note – Some versions of the RMIC do not treat `Throwable` as a superclass of `RemoteException`. The workaround is to declare remote methods to throw `Exception` instead.

Java Card RMI Client-Side API

The two packages in the Java Card RMI client-side reference implementation demonstrate remote stub customization using the RMIC compiler generated stubs and card access for Java Card applets.

The package `com.sun.javacard.javacard.rmiclientlib` implements Java Card RMI-specific functionality.

The package `com.sun.javacard.javacard.clientlib` implements basic functionality to exchange APDUs with a smart card or a smart card simulator. This implementation of `clientlib` requires that the ApduIO library is included in the CLASSPATH.

In the release bundles, the Javadoc tool files for this API are located at:

JC_CLASSIC_HOME`/docs/rmijavadocs`

Package `rmiclientlib`

This package includes several classes.

- **class JCRMIConnect**–The main class of the RMI framework that provides methods to select a card applet and to get an initial reference.
- **class JCCardObjectFactory**–An implementation of the `CardObjectFactory` that processes the data returned from the card in the format defined in the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Classic Edition*. Any object references must contain class names.
- **class JCCardProxyFactory**–The `JCCardProxyFactory` class is similar to `JCCardObjectFactory`, but processes references containing lists of names. `JCCardProxyFactory` uses the JDK 1.4.+ proxy mechanism to generate proxies dynamically.
- **class JCRemoteRefImpl**–An implementation of interface `java.rmi.server.RemoteRef`. These remote references can work with stubs generated by the RMIC compiler with the `-v1.2` option.

  The main method is:
  `public Object invoke(Remote remote, Method method, Object[] params, long unused) throws IOException, RemoteException, Exception`

  This method prepares the outgoing APDU, passes it to `CardAccessor`, and then uses `CardObjectFactory` to parse the returned APDU and instantiate the returned object or throw an exception.

Package `clientlib`

This package includes an interface and a class.

- **interface CardAccessor**–An interface defining methods to exchange APDUs with a card and to close connection to a card.

- **class ApduIOCardAccessor**–A simple implementation of the CardAccessor interface that passes the APDUs to a card or a card simulator using the ApduIO library. This class takes parameters to start the ApduIO from the file `jcclient.properties`, which must be included in CLASSPATH.

CHAPTER 15

Working with APDU I/O

This chapter describes the APDU I/O API, which is a library used by many Java Card development kit components, such as `apdutool`, and the RMI client framework, see Chapter 14.

The APDU I/O library can also be used by developers to develop Java Card client applications and Java Card platform simulators. It provides the means to exchange APDUs by using the T=0 protocol over TLP224, by using T=1, and by using the PC/SC API. (However, note that PC/SC is unsupported and may not work on all platforms with all card readers).

The library is located in the file `lib/tools.jar`.

The APDU I/O API

The following sections describe the APDU I/O API. All publicly available APDU I/O client classes are located in the package `com.sun.javacard.apduio`.

Javadoc tool files for the APDU I/O APIs are also in a PDF file located in this bundle at *`JC_CONNECTED_HOME`*`\docs\apduiojavadocs.pdf`.

APDU I/O Classes and Interfaces

The APDU I/O classes and interfaces are described in this section.

- `class Apdu`

  Represents a pair of APDUs (both C-APDU and R-APDU). Contains various helper methods to access APDU contents and constants providing standard offsets within the APDU.

- `interface CadClientInterface`

Represents an interface from the client to the card reader or a simulator. Includes methods for powering up, powering down and exchanging APDUs.

- `void exchangeApdu(Apdu apdu)`

  Exchanges a single APDU with the card. Note that the APDU object contains both incoming and outgoing APDUs.

- `public byte[] powerUp()`

  Powers up the card and returns ATR (Answer-To-Reset) bytes.

- `void powerDown(boolean disconnect)`

  Powers down the card. The parameter, applicable only to communications with a simulator, means "close the socket". Normally, it is `true` for contacted connection, `false` for contactless. See "Two-interface Card Simulation" on page 131 for more details.

- `void powerDown()`

  Equivalent to `powerDown(true)`.

- `abstract class CadDevice`

  Factory and a base class for all `CadClientInterface` implementations included with the APDU I/O library. Includes constants for the T=0, T=1 and PC/SC (unsupported) clients.

  The factory method `static CadClientInterface getCadClientInstance(byte protocolType, InputStream in, OutputStream out)` returns a new instance of `CadClientInterface`. The in and out streams correspond to a socket connection to a simulator. Protocol type can be one of:

  - `CadDevice.PROTOCOL_T0`
  - `CadDevice.PROTOCOL_T1`
  - `CadDevice.PROTOCOL_PCSC`

  The parameters, `InputStream` and `OutputStream`, are not used for PC/SC (unsupported).

Exceptions

Various exceptions may be thrown in case of system malfunction or protocol violations. In all cases, their `toString()` method returns the cause of failure. In addition, `java.io.IOException` may be thrown at any time if the underlying socket connection is terminated or could not be established.

- `CadTransportException` extends `Exception`
- `T1Exception` extends `CadTransportException`

- `TLP224Exception` extends `CadTransportException`

Two-interface Card Simulation

To simulate dual-interface cards with the RI the following model is used:

- The simulator (`cjcre.exe`) listens for communication on two TCP sockets: (*n*) and (*n*+1), where *n* is the default (9025) or the socket number given in the command line.
- The client creates two instances of the `CadClientInterface`, with protocols T= 1 on both. One of these instances communicates on the port (*n*), while the other communicates on the port (*n*+1).
- Each of these client interfaces needs to issue the `powerUp` command before being able to exchange APDUs.
- Issuing the `powerDown` command on the contactless interface closes all contactless logical channels. After this, the contacted interface is still available to exchange APDUs. The client also may issue `powerUp` on a contactless interface again and continue exchanging APDUs on the contactless interface too.
- Issuing the `powerDown` command on the contacted interface closes all channels and causes the simulator (`cjcre.exe`) to exit. That is, any activity after powering down the contacted interface requires restarting the simulator and reestablishing connections between the client and the simulator.
- At most, one socket can be processing an APDU at any time. The client may send the next APDU only after the response of the previous APDU is received. This means, behavior of the client+simulator still remains deterministic and reproducible.
- If you have a source release of the Java Card development kit, you can see a sample implementation of such a dual-interface client in the file `ReaderWriter.java` inside the `apdutool` source tree.

Examples of Use

The following sections give examples of how to use the APDU I/O API.

To Connect To a Simulator

To establish a connection to a simulator such as `cjcre.exe` , use the following code.

```
CadClientInterface cad;
Socket sock;
sock = new Socket("localhost", 9025);
InputStream is = sock.getInputStream();
OutputStream os = sock.getOutputStream();
cad=CadDevice.getCadClientInstance(CadDevice.PROTOCOL_T0, is, os);
```

This code establishes a T=0 connection to a simulator listening to port `9025` on `localhost`. To open a T=1 connection instead, in the last line replace `PROTOCOL_T0` with `PROTOCOL_T1`.

Note – For dual-interface simulation, open two T=1 connections on ports (*n*) and (*n*+1), as described in "Two-interface Card Simulation" on page 131.

To Establish a T=0 Connection To a Card

To establish a T=0 connection to a card inserted in a TLP224 card reader, which is connected to a serial port, use the following code.

```
String port = "com1";  // serial port's name
CommPortIdentifier portId = CommPortIdentifier.getPortIdentifier(port);
String appname = "Name of your application";
int timeout =  30000;
CommPort commPort = portId.open(appname, timeout);
InputStream is = commPort.getInputStream();
OutputStream os = commPort.getOutputStream();
cad=CadDevice.getCadClientInstance(CadDevice.PROTOCOL_T0, is, os);
```

Note – For this code to work, you need a TLP224-compatible card reader, which is not widely available. You also need the `javax.comm` library installed on your machine. See "`javax.comm` Package" on page 133 for details on how to obtain this library.

`javax.comm` Package

Install `javax.comm` **package** if you are planning to use the development kit to communicate with a TLP224-compatible card reader.

Use the `javax.comm` package included in the latest version of the Java Communications API, available on Sun's web site at:
`http://java.sun.com/products/javacomm`

Follow the instructions provided in the file `Readme.html` to install the package, and make sure that the `comm.jar` file is added to the CLASSPATH.

To Establish a Connection To a PC/SC-Compatible Card Reader

To establish a connection to the default PC/SC-compatible card reader (unsupported) installed on the machine, use the following code.

```
cad=CadDevice.getCadClientInstance(CadDevice.PROTOCOL_PCSC, null,
null);
```

To Power Up And Power Down the Card

To power up the card, use the following code.

```
cad.powerUp();
```

To power down the card and close the socket connection (for simulators only), use either of the following code lines.

```
cad.powerDown(true);
```

or

```
cad.powerDown();
```

To power down, but leave the socket open, use the following code. If the simulator continues to run (which is true if this is contactless interface of the RI) you can issue `powerUp()` on this card again and continue exchanging APDUs.

```
cad.powerDown(false);
```

The dual-interface RI is implemented in such a way that once the client establishes connection to a port, the next command must be `powerUp` on that port.

For example, the following sequence is valid:

1. **Connect on "contacted" port.**
2. **Send** `powerUp` **to it.**
3. **Exchange some APDUs.**
4. **Connect on "contactless" port.**
5. **Send** `powerUp` **to it.**
6. **Exchange more APDUs.**

However, the following sequence is not valid:

1. **Connect on "contacted" port.**
2. **Connect on "contactless" port.**
3. **Send** `powerUp` **to any port.**

To Exchange APDUs

To exchange APDUs, first create a new APDU object using the following code:

```
Apdu apdu = new Apdu();
```

Copy the header `(CLA, INS, P1, P2)` of the APDU to be sent into the `apdu.command` field.

Set the data to be sent and the `Lc` using the following code:

```
apdu.setDataIn(dataIn, Lc);
```

where the array `dataIn` contains the C-APDU data, and the `Lc` contains the data length.

Set the number of bytes expected into the `apdu.Le` field.

Exchange the APDU with a card or simulator using the following code:

```
cad.exchangeApdu(apdu);
```

After the exchange, `apdu.Le` contains the number of bytes received from the card or simulator, `apdu.dataOut` contains the data received, and `apdu.sw1sw2` contains the SW1 and SW2 status bytes.

These fields can be accessed through the corresponding `get` methods.

To Print the APDU

The following code prints both C-APDU and R-APDU in the `apdutool` output format.

```
System.out.println(apdu)
```

CHAPTER 16

Generating SSL Keys and Certificates

This chapter describes how to generate a certificate that can be used in SSL and HTTPS transactions.

SSL and HTTPS Certificates and Keys

An SSL implementation needs four algorithms, digital signature., key establishment, and bulk encryption, and message digest. The Java Card 3 platform implements the SSL key establishment algorithm through the use of the following set of certificates and keys as *key=value* pairs in `system.config`:

- `ssl.serverIdentity` - The CA certificate.
- `ssl.selfIdentityAsServer` - The server public certificate.
- `ssl.selfIdentitySSLPrivateKeyMod` - The server private key (mod).
- `ssl.selfIdentitySSLPrivateKeyExp` - The server private key (exp).
- `ssl.selfIdentityAsClient` -The client public certificate.

Custom implementations require that the developer generate corresponding custom certificates and keys. The certificates and keys are used by the Card Manager to verify the digital signature of WAR file.

▼ Generating an SSL Certificate

1. **Generate a server key and certificate signing request (csr):**

```
openssl genrsa -out s.key 1024
openssl req -new -key s.key -out server.csr
```

2. **Generate a CA key and self-signed certificate:**

```
openssl genrsa -out ca.key 1024
openssl -req new -x509 -days 365 -key ca.key -out ca.crt
```

3. **Sign the csr and create the certificate:**

```
sign.sh server.csr
```

APPENDIX **A**

Application Module and Library Formats

This appendix describes the application module and library formats supported by the Java Card 3 platform card manager. Applications are distributed and deployed as application module JAR files. The application module distribution format JAR file contains one application. Libraries are distributed and deployed as standard library JAR files containing the library classes.

There are two types of library formats:

- The extension library JAR file is a standard library JAR format containing Java class files. Extension library classes are accessible to all applications on the card. Instances of classes instantiated from the extension library are placed in the owner context of the application which creates the instance.
- The classic library JAR file is a standard JAR library format containing Java class files. Classic library classes are only accessible to the classic applications on the card. Instances of classes instantiated from the classic library are placed in the owner context of the classic application which creates the instance.

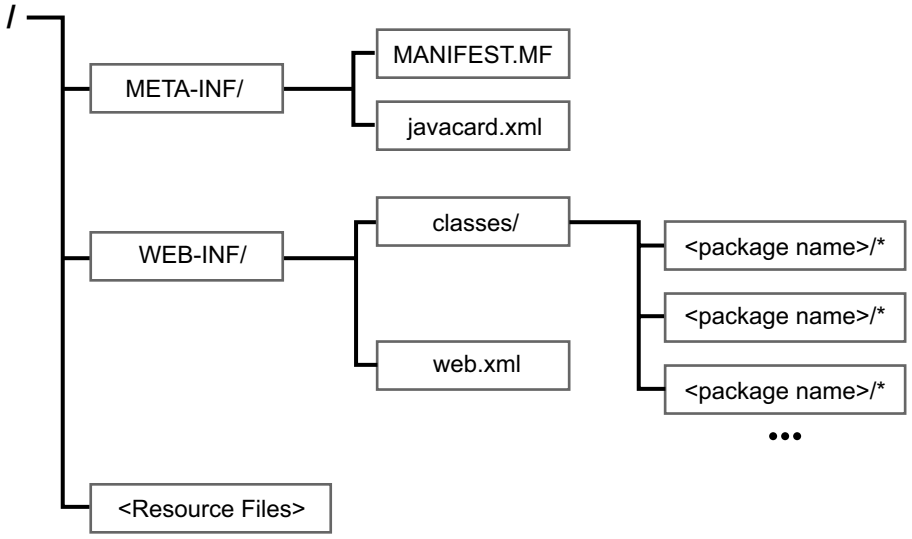
This appendix contains the following sections:

Web Application Module Format

FIGURE A-1 shows the directory structure of the web application module distribution format. The structure must be that of the web archive (`.war`) file with the following differences:

- No support for application private library directory `WEB-INF/lib`
- An additional Java Card 3 platform-specific application descriptor file `javacard.xml` is supported. The format of this descriptor is specified in *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition.*

FIGURE A-1 Web Application Module Format

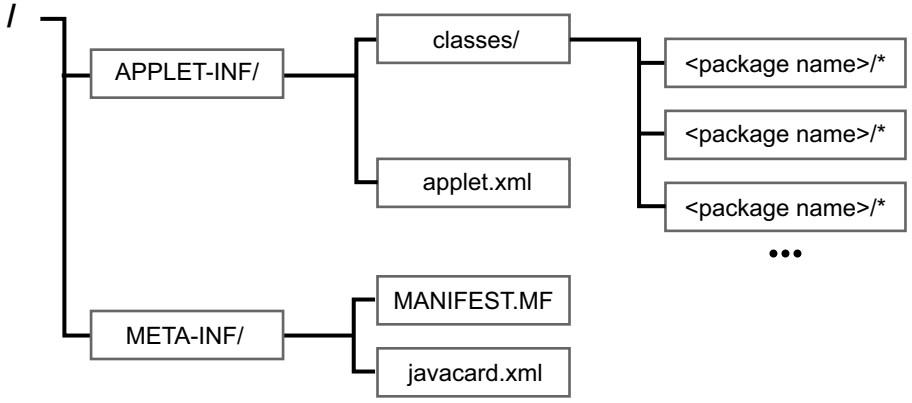


See the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for specific details about the web application module format.

Extended Applet Application Module Distribution Format

FIGURE A-2 shows the directory structure of the extended applet application module format. See the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for specific details.

FIGURE A-2 Extended Applet Application Module Format



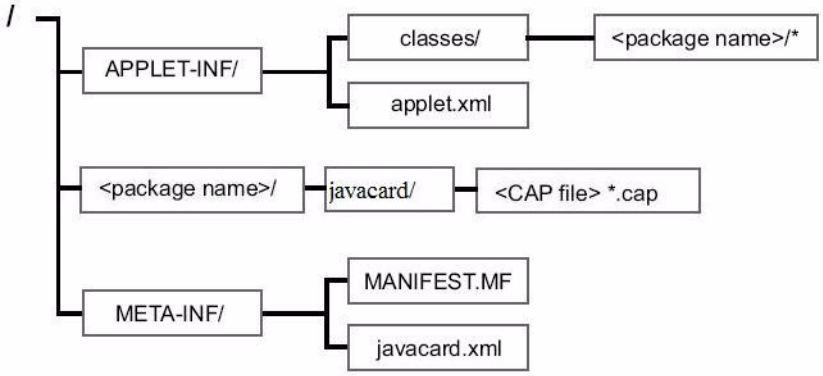
See the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for specific details about the extended applet application module format.

Classic Applet Application Module Format

FIGURE A-3 shows the directory structure of the classic applet application module distribution format. The structure is similar to that of the extended applet application module with the following differences:

- The `classes` directory contains only one package and optionally a subpackage named `proxy` containing SIO proxy classes.
- The Classic Edition's CAP file components, `*.cap`, are included in the JAR file.

FIGURE A-3 Classic Applet Application Module Format

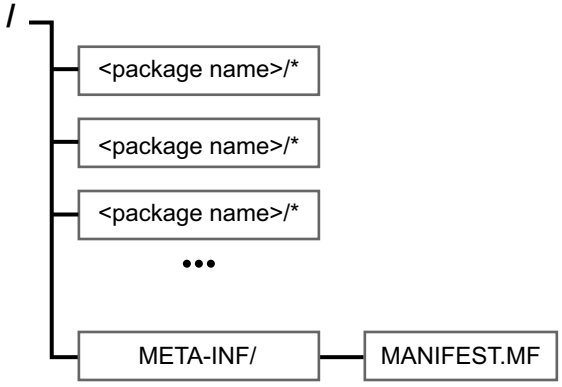


See the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for specific details about the requirements of the classic applet application module format.

Extension Library Format

The extension library distribution format uses the Java Platform Standard Edition library JAR file structure. FIGURE A-4 shows the format of a Java Platform Standard Edition library JAR file format.

FIGURE A-4 Java Platform Standard Edition Library JAR Format



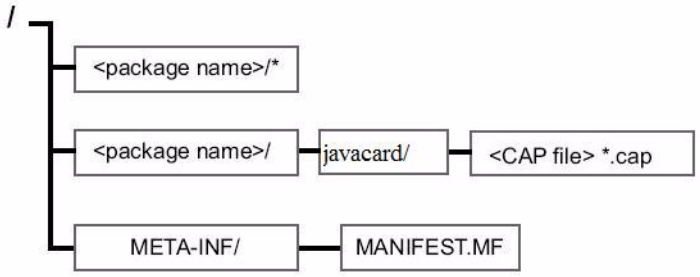
See the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for specific details about the extension library format.

Classic Library Format

FIGURE A-5 shows the format of a classic library distribution format. The classic library distribution format uses the Java Platform Standard Edition library JAR file format (see FIGURE A-4) with the following restrictions and additions:

- It contains only one package and, optionally, a subpackage `proxy` containing SIO proxy classes.
- It includes the classic CAP file components, `*.cap`, in a directory named `javacard` that is in a subdirectory representing the library package directory as described in *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*. The format of the CAP file components are described in *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition.*

FIGURE A-5 Classic Library Format



See the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for specific details about the classic library format.

APPENDIX **B**

Installed Directories and Files

The Development Kit source bundles install files and directories containing the binary files and source code described in TABLE B-1 and TABLE B-2. Except for the src directory and its contents, the binary bundles install the files described in TABLE B-1. The files and directories are installed under the root installation directory, `c:\JCDK3.0.1_ConnectedEdition` or in the directory that you specified during installation. The root installation directory is referred to as *JC_CONNECTED_HOME* in this guide.

TABLE B-1 Installed Directories and Files

Directory or File	Description
`COPYRIGHT-software.html`	The copyright file for the Java Card 3 platform.
`COPYRIGHT-docs.html`	The copyright file for the documentation of the Java Card 3 platform.
`RELEASENOTES.html`	The release notes for this Java Card 3 platform Development Kit.
`document.css`	The style sheet for the HTML documentation.
`platform.properties`	Specfies properties of the Java Card 3 platform RI that are used by the tools.
`api_export_files/`	Contains `java`, `javacard`, and `javacardx` directories of API export files.
`bin/`	Contains all shell scripts and batch files (including the `cjcre.exe` binary executable) used in running the tools .

TABLE B-1 Installed Directories and Files *(Continued)*

Directory or File	Description
`docs/`	Contains the following: • `apduiojavadocs.pdf` - A compilation of the Javadoc tool files for the publicly available APDU I/O client classes in PDF format. • `rmijavadocs.pdf` - A compilation of the Javadoc tool files for the Java Card RMI client-side reference implementation in PDF format. Java Card RMI client-side reference implementation demonstrates remote stub customization using the RMIC compiler generated stubs and card access for Java Card applets. **Note -** The RI for the Classic Edition supports RMI but the RI for the Connected Edition does not. • `JCDevKitUG-Connected-3_0_1.pdf` - this user's guide. • `api javadoc` folder - contains the Javadoc tool files for the API in HTML format. • `UserGuide_html` folder - contains the HTML version of this user's guide.
`legal/`	Contains three files: • `TechnologyEvaluationLicense.txt` - License for the Java Card 3 platform. • `THIRDPARTYREADME.txt` - License for the Jetty HTTP Server. • `Distribution_ReadME.txt` - Describes the terms and conditions for redistribution of the Java Card Development Kit.

TABLE B-1 Installed Directories and Files *(Continued)*

Directory or File	Description
`lib/`	Contains all Java programming language JAR files and `config` files required for the tools: • `ant-contrib-1.0b3.jar` • `api_classic.jar` • `api_connected.jar` • `bcel-5.2.jar` • `commons-cli-1.0jar` • `commons-codec-1.3.jar` • `commons-httpclient-3.0.jar` • `commons-logging-1.1.jar` • `config.properties` • `jcapt.jar` • `jctasks.jar` • `nbtasks.jar` • `nbutils.jar` • `nhelper.dll` • `romizer.jar` • `system.properties` • `tools.jar`
`samples/classic_applets`	Contains source files and directories for classic applet sample applications adapted to run on the Connected Edition.
`samples/extended_applets`	Contains source files and directories for extended applet sample applications.
`samples/keystore`	Contains keystore and other certificate files for use by the samples provided in this release. These keystore and other certificate files are for demonstration purposes only and cannot be used for developing deployable applications.

TABLE B-1 Installed Directories and Files *(Continued)*

Directory or File	Description
`samples/reference_apps`	Contains source files and directories for sample refernce applications.
`samples/web`	Contains source files and directories for sample web applications. .
`src/`	**Note –** This directory is only installed by the source bundle. Contains the source code for the Java Card API, the ROMized applications, the Development Kit tools, and the Java Card virtual machine. For more information on the contents of the directory, see "Directories and Files Installed in the `src` Directory" on page 148.

Directories and Files Installed in the `src` Directory

TABLE B-2 describes the `src` directory and files installed under `c:\JCDK3.0.1_ConnectedEdition` (or the alternate directory you specified during installation).

TABLE B-2 Contents of the `src` Directory

Directory or File	Description
`build.xml`	Resource file for rebuilding the Development Kit source bundle.
`apiImpl.jar`	
`bat.template`	
`crypto.jar`	
`api/`	Sources for the Java Card API version 3.0.1 in the following subdirectories: • `com\sun` • `java` • `javacard` • `javacardx` • `javax` • `org\mortbay`

TABLE B-2 Contents of the `src` Directory *(Continued)*

Directory or File	Description
`romized_apps`	Sources for the CardManager servlet.
`tools/`	Sources for Development Kit tools.
`vm/`	Sources for the Java Card virtual machine in the following subdirectories and file: `c` - C programming language sources. `h` - Header files for the C programming language sources. `lib` - System and internal web configuration files `ignore.list` - List of classes ignored by the ROMizer

APPENDIX C

Development Kit Tool Commands

This appendix provides a quick reference to the following commands used to run the Development Kit tools:

`apdutool.bat` Command

Use the `apdutool.bat` command from the *JC_CONNECTED_HOME* directory to send APDU commands to `cjcre.exe` where they are processed and returned to `the` APDU tool. The APDU tool displays both the command and response APDU commands on the console. See Chapter 10 for detailed information about the APDU tool.

The APDU tool is run from the command line using the following syntax:

```
apdutool.bat [-t0] [-verbose] [-nobanner] [-noatr] [-o outputFile]
          [-h hostname] [-p port] [-s serialPort ] [-version]
          [inputFile ...]
```

Options for the `apdutool.bat` command include the following:

- `-t0`

  Runs T=0 single interface

- `-verbose`

  Displays descriptive text during operation.

- `-nobanner`

  Suppresses all banner messages.

- `-noatr`

  Suppresses outputting an ATR (answer to reset).

- `-o` *outputFile*

  Specifies an output file. If an output file is not specified with the `-o` option, output defaults to standard output.

- `-h` *hostname*

  Specifies the host name on which the TCP/IP socket port is found. See the `-p` *portNumber* option.

- `-p` *port*

  Specifies a TCP/IP socket port other than the default port (9025)

- `-s` *serialPort*

  Specifies that the serial port is used for communication, rather than a TCP/IP socket port. For example, *serialPort* can be `COM1`.

  To use this option, the `javax.comm` package must be installed on your system.

  If you enter the name of a serial port that does not exist on your system, the APDU tool will respond by printing the names of available ports.

- `-version`

  Outputs the version information.

- *inputFile*

  Allows you to specify the input script or scripts.

- `-help`

  Displays online documentation for the `apdutool.bat` command. To get help for `the` APDU tool, run `bin/apdutool.bat -help` on the command line.

cjcre.exe Command

Use the cjcre.exe command to run the reference implementation (RI). The RI is a pre-built executable located in *JC_CONNECTED_HOME*\bin. See Chapter 5 for detailed information about the RI.

Assuming that *JC_CONNECTED_HOME*\bin is on your path, the following syntax is used to run the RI:

```
cjcre.exe [options]
```

cjcre.exe Options

Options must follow the cjcre.exe command on the command line. Valid cjcre.exe command options used in the command line consist of one or more of the following:

- -config *config file*

  Sets a new configuration file. The default is lib/config.properties.

- -contactedport *portnumber*

  Sets the port used to simulate the contacted interface for APDU. The default value for -contactedport is 9025.

- -contactedprotocol *protocol*

  Sets the APDU protocol on this port, either T=0 or T=1. The default value for the -contactedprotocol is T=1.

- -contactlessport *port-number*

  Port number used to simulate contactless interface. Default is 9026. The protocol, T=CL, cannot be changed.

- -corsize *size*

  Sets the Clear On Reset (COR) memory size in which a portion of RAM is dedicated to COR memory. The range of values that the Java Card runtime environment can accept from the command line is 2K to 8K. The default value is 4K. *size* is set as a value in bytes (2345) or kilobytes (2K).

- -D*name=value*

  Supplies a system property (such as -D*myproperty=myvalue*). System properties set in this manner can be retrieved using the API's System.getProperty("myproperty") method. A maximum of 50 -D properties can be passed in the command line.

- `-debugger`

  Runs `cjcre` in debug mode.

- `-debugport` *portnumber*

  Sets the debug port where the Debug proxy communicates. The default value for `-debugport` is 7019.

- `-e2pfile` *filename*

  Supplies the file name in which the EEPROM image is stored.

- `-e2psize` *size*

  Configures the amount of EEPROM used. *size* is set as a value in bytes (`2345`), kilobytes (`32K`), or megabytes (`4M`). The specified size is rounded up to a multiple of 4. For example, a size specified at 253, is rounded up to 256.

  The range of values that the Java Card runtime environment can accept from the command line is `1M` to `32M`. The default value used is `4M`. The value required to run the samples is between `2M` and `32M`.

- `-enableassertions`

  Enables Java code assertions (the assert keywork in Java code).

- `-help` [`copyright`]

  Prints help and copyright messages.

- `-httpport` *portnumber*

  Sets the HTTP port number on which `cjcre` will be listening. The default value for `-httpport` is 8019.

- `-loggerlevel <none|fatal|error|warn|info|verbose|debug|all>`

  Sets the type of log messages output. All log messages up to the specified level are displayed.

- `-nosuspend`

  Valid when `-debugger` is specified. Will not suspend threads at `cjcre` startup.

- `-ramsize` *size*

  Configures the amount of RAM used. *size* is set as a value in bytes (`2345`), kilobytes (`32K`), or megabytes (`4M`).

  The range of values that the Java Card runtime environment can accept from the command line is `64K` to `32M`. The default value used is `1M`. The value required to run the samples is between `128K` and `32M`.

- `-resume`

  Restores the VM state from the previously saved EEPROM image and continues VM execution. When `-resume` is specified, other options such as `-ramsize and -e2psize` are ignored and the corresponding values are obtained from the EEPROM image. The range is 256 bytes to 8K.

- `-version`

  Displays version information.

- `-X`*name=value*

  Sets a single configuration property such as:
  `-Xmyproperty=myvalue`

- `-exactlogger`

  Displays only the log messages that match the level set by the `-loggerlevel` option.

converter.bat Command

The Converter generates SIO proxy for the CAP file and adds it to the JAR created by the Normalizer. The CAP files are not changed by the Converter.

When the Normalizer is run, it automatically calls the Converter. A developer only needs to run the Converter as a separate operation when the Normalizer is not run. The command line interface for running the Converter takes one of the following forms:

`converter.bat` *options package_name package_aid major_version*`.`*minor_version*

or

`converter.bat -config` *filename*

Use the `-config` subcommand and the associated file to provide all options and parameters to the Converter.

converter Command Options

Use the following `converter` command options to specify input files, an export path, an export map, names, and output directories:

- `-classdir` *root- directory-of-class-hierarch*y

  Specifies the root directory where the Converter looks for classes.

- `-i`

  Specifies support the 32-bit integer type.

- `-exportpath` *list-of-directories*

  Specifies the root directories where the Converter looks for export files.

- `-exportmap`

  Uses 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*

  Sets the applet AID and the class that defines the install method for the applet.

- `-d` *root-directory-for-output*

  Specifies the root directories where the Converter outputs the files.

- `-out` [`CAP`] [`EXP`] [`JCA`]

  Specifies that the Converter output the `CAP` file, and/or the `JCA` file, and/or the `EXP` export file.

- `-V` (or `-version`)

  Displays the Converter version number.

- `-v` (or `-verbose`)

  Enables verbose output.

- `-help`

  Displays the contents of this table.

- `-nowarn`

  Instructs the Converter to not report warning messages.

- `-mask`

  Identifies this package is used for a mask. Restrictions on native methods are relaxed.

- `-debug`

  Enables generation of debugging information.

- `-nobanner`

  Suppresses standard output messages.

- `-noverify`

  Turns off verification. Verification is default.

- `-sign`

  Signs the output CAP file.

- `-keystore` *keystore*

  Keystore to use in signing.

- `-storepass` <storepass>

  Keystore password.

- `-alias` *alias*

Keystore alias to use in signing.

- `-passkey` *passkey*

  Alias password.

`debugproxy.bat` Command

Use the Debugger tool's functionality, by starting the `debugproxy` (`debugproxy.bat`), attaching a Java technology enabled IDE to it, and then starting `cjcre` with the `-debugger` option. This tool is located in the *JC_CONNECTED_HOME*`\bin` directory. See Chapter 11 for detailed information about the Debugger tool.

The Debugger is run from the command line using the following syntax:

`debugproxy.bat` *options filelocation*

When starting debugproxy, include the `-c` (or `--classpath`) option in the command to specify the path of the class files to be debugged. The debugproxy needs to know the location of class files being debugged.

The following is an example of a command that starts `debugproxy` and specifies `myapp.war` as the location of the class files to be debugged:

```
debugproxy.bat -c myapp.war
```

`installer.bat` Command

Use the `installer.bat` command to run the Off-Card installer. The Installer is a command-line tool that performs various card-management tasks such as loading or unloading an application, creating instances, and listing available applications. This tool is located in the *JC_CONNECTED_HOME*`\bin` directory. See Chapter 8 for detailed information about the Installer tool.

The Off-Card installer is run from the command line using the following syntax:

`installer.bat` *subcommand* [*options*] [*arguments*]

Each subcommand can take one or more options or arguments that must follow the subcommand but can be in any order. Some options might require a value. Command-line arguments are not bound to any particular option.

Subcommands for the Off-Card installer include the following:

- `load` - Loads a specified application-module or library file.
- `create` - Creates an instance of a web application from a specified module with specified context.
- `delete` - Deletes an instance that was created by the `create` subcommand.
- `unload` - Unloads (removes) the given application-module (or) library from the card.
- `list` - Displays summary (or) detailed information about application-modules, instances, and libraries.
- `help` - Displays usage information details for individual installer sub-commands.

`load` Subcommand

The `load` subcommand can have one or more options and arguments.

`load` Options

Options must follow the subcommand on the command line. Valid `load` options include the following:

- `-c` (or `--cardmanager`) *oncard installer-url*

  Specifies the location of the on-card installer where *oncard installer-url* represents the complete URL of the on-card installer.

- `-n` (or `--name`) *module-or-library-name*

  Specifies the module name on the card, where *module-or-library-name* represents the name of the module or library.

- `-p` (or `--password`) *password*

  Optional. Used when authentication is required. Specifies the password for the user set by the `--user` or `-u` subcommand, where *password* represents the required user password.

- `-s` (or `--signature`) *signature-file*

  Specifies the name of the properties file that contains the BASE64 encoded certificate and signature, where *signature-file* represents the file name. This file is a simple properties file setting the following properties:

  - `signature=`*base64-encoded-signature*
  - `certificate=`*certificate-to-validate-the-module-and-digest*

- `-t` (or `--type`) *file-type*

Specifies the type of file being loaded, where *file-type* represents one of the following values:

- `web`
- `classic-lib`
- `classic-applet`
- `extension-lib`
- `extended-applet`

- `-u` (or `--user`) *user-id*

  Optional. Used if authentication is required to access the card manager, where *user-id* represents the user name.

`load` Arguments

The `load` subcommand must include the application-module or library file name as an argument following the appropriate options.

`load` Subcommand Format

```
installer.bat load -c oncard installer-url [-s signature-file] \
          -t file-type -n module-or-library-name\
          [-v] [-u user-id -p password] \
          application-module (or library-file)
```

`load` Subcommand Example

The following is an example of the installer `load` subcommand:

```
installer.bat load -c http://localhost:8019/cardmanager \
          -s mysig.properties -n calc -t web Calculator.war
```

`create` Subcommand

The `create` subcommand can have one or more options but has no arguments.

`create` Options

Options must follow the subcommand on the command line. Valid `create` options include the following:

- `-a` (or `--applet`) *applet-name-or-id*

  Specifies the name of the applet loaded by load command, where *applet-name-or-id* represents the applet name.

- `-c` (or `--cardmanager`) *oncard installer-url*

  Specifies the location of the on-card installer, where *on-card installer url* represents the complete URL.

- `-d` (or`--data`) *install-parameters*

  Optional. Specifies the parameters (printable hex string) that will be passed to the install method of a classic or extended applet.

- -i (or --instance) *instance-name*

  For web applications, the context name is used to create the web application. If none is specified, then the default Web-Context-Path from JCRD is used.

  For Applet applications, this is the instance id for applet. If none specified, then a new instance id is created based on the aid of that applet.

- `-n` (or `--name`) *module-or-library-name*

  The option must include the *module-or-library-name*. *module-or-library-name* represents the name of the module or library loaded by the `load` command.

- `-p` (or `--password`) *password*

  Optional. Used when authentication is required. Sets the *password* for the user specified by the `--user` or `-u` subcommand.

- `-u` (or `--user`) *user-id*

  Optional. If authentication is required to access the card manager, the *user-id* represents the user name.

`create` Arguments

None

`create` Subcommand Format

```
installer.bat create -c oncard installer-url -n module-or-library-name \
          [-a applet-name-or-id] [-d install-parameters][-i instance-name] \
          [-v] [-u user-id -p password]
```

`create` Subcommand Example

The following is an example of the installer `create` subcommand:

```
installer.bat create -c http://localhost:8019/cardmanager -n calc \
          -i /MyCalc
```

`delete` Subcommand

The `delete` subcommand can have one or more options but no arguments.

`delete` Options

Options must follow the subcommand on the command line. Valid `delete` options include the following:

- `-c` (or `--cardmanager`) *oncardinstaller-url*

  Specifies the location of the on-card installer, where *oncardinstaller-url* represents the complete URL.

- `-i` *name* (or `--instance` *name*) or `-i` *name;name1;name2;...* (or `--instance` *name;name1;name2;...*)

  Instance name of the application or multiple instances of applications that need to be deleted, where *name* represents the instance name of the application.

- `-p` (or `--password`) *password*

  Optional. Used when authentication is required. *password* is for the user specified by either the `--user` or `-u` subcommand.

- `-u` (or `--user`) *user-id*

  Optional. Used if authentication is required to access the card manager. *user-id* represents the user name.

`delete` Arguments

None

`delete` Subcommand Format

```
installer.bat delete -c oncardinstaller-url -i instance-name \
          [-u user-id -p password]
```

`delete` Subcommand Example

The following is an example of the installer `create` subcommand:

```
installer.bat delete -c http://localhost:8019/cardmanager -i /MyCalc
```

`unload` Subcommand

The `unload` subcommand can have one or more options but no arguments.

`unload` Options

Options must follow the subcommand on the command line. Valid `unload` options include the following:

- `-c` (or `--cardmanager`) *oncardinstaller-url*

  Specifies the location of the on-card installer, where *oncardinstaller-url* represents the complete URL.

- `-n` (or `--name`) *module-or-library-name*

  The option must include the *module-or-library-name*. *module-or-library-name* represents the name of the module or library loaded by the `load` command.

- `-f` (or `--force`)

  (Optional) Forces an attempt to delete any instances before unloading.

- `-p` (or `--password`) *password*

  Optional. Used when authentication is required. *password* is for the user specified by either the `--user` or `-u` subcommand.

- `-u` (or `--user`) *user-id*

  Optional. Used if authentication is required to access the card manager. *user-id* represents the user name.

`unload` Arguments

None

`unload` Subcommand Format

```
installer.bat unload -c oncardinstaller-url -n module-or-library-name \
          [-f] [-u user-id -p password]
```

`unload` Subcommand Example

The following is an example of the installer `unload` subcommand:

```
installer.bat unload -c http://localhost:8019/cardmanager -n calc
```

`list` Subcommand

The `list` subcommand can have one or more options but no arguments.

`list` Options

Options must follow the subcommand on the command line. Valid `list` options include the following:

- `-c` (or `--cardmanager`) *oncardinstaller-url*

  Specifies the location of the on-card installer, where *oncardinstaller-url* represents the complete URL.

- -d (or --detailed)

  Optional. Displays complete details of the application-modules, instances, and libraries.

- `-p` (or `--password`) *password*

  Optional. Used when authentication is required. *password* is for the user specified by either the `--user` or `-u` subcommand.

- `-u` (or `--user`) *user-id*

  Optional. Used if authentication is required to access the card manager. *user-id* represents the user name.

`list` Arguments

None

`list` Subcommand Format

`installer.bat list -c` *oncardinstaller-url* `[-d] [-v] [-u` *user-id* `-p` *password*`]`

`list` Subcommand Example

The following is an example of the installer `list` subcommand that displays summary information about application-modules, instances, and, libraries:

```
installer.bat list -c http://localhost:8019/cardmanager
```

The following is an example of the installer `list` subcommand that displays detailed information about application-modules, instances, and, libraries.

```
installer.bat list -d -c http://localhost:8019/cardmanager
```

`help` Subcommand

Causes the installer to display summary or detailed information about one or more installer subcommands.

`help` Subcommand Options

There are no options, but the `help` subcommand accepts a topic attribute of a specific subcommand name for which detailed information is displayed.

`help` Subcommand Format

The following is an example of the `help` subcommand format:

`installer.bat help` *subcommand*

`help` Subcommand Example

The following is an example of the `help` subcommand:

```
installer.bat help list
```

`javacardc.bat` Command

The Java Card 3 platform Compiler tool (`javacardc.bat`) is used by developers to compile the source code of Java Card 3 platform applications outside of an IDE. See Chapter 6 for detailed information about the Compiler tool.

The following is an example of the Compiler tool command format:

`javacardc.bat [`*options*`] [`*sourcefiles*`] [@`*list_files*`]`

In the format example:

- *options* - standard javac options,
- *sourcefiles* - `.java` files to be compiled
- *@list_files* - plain text file containing a list of all java files that need to be compiled

Compiler Tool Options

In addition to the following Java Card 3 platform specific options, all standard `javac` options for JDK 1.6 can be used:

- `-g`

  Generate all debugging info.

- `-g:none`

  Generate no debugging info.

- `-g:{lines,vars,source}`

  Generate only some debugging info.

- `-nowarn`

  Generate no warnings.

- `-verbose`

  Output messages about what the compiler is doing.

- `-deprecation`

  Output source locations where deprecated APIs are used.

- `-classpath` *path*

  Specify where to find user class files and annotation processors.

- `-cp` *path*

  Specify where to find user class files and annotation processors.

- `-sourcepath` *path*

  Specify where to find input source files.

- `-bootclasspath` *path*

  Override location of bootstrap class files.

- `-extdirs` *dirs*

  Override location of installed extensions.

- `-endorseddirs` *dirs*

  Override location of endorsed standards path.
- `-proc:{none,only}`

  Control whether annotation processing and/or compilation is done.
- `-processor` *class1*`[,`*class2*`,`*class3*`...]`

  Names of the annotation processors to run; bypasses default discovery process.
- `-processorpath` *path*

  Specify where to find annotation processors.
- `-d` *directory*

  Specify where to place generated class files.
- `-s` *directory*

  Specify where to place generated source files.
- `-implicit:{none,class}`

  Specify whether or not to generate class files for implicitly referenced files.
- `-encoding` *encoding*

  Specify character encoding used by source files.
- `-source` *release*

  Provide source compatibility with specified release.
- `-target` *release*

  Generate class files for specific VM version.
- `-version`

  Version information.
- `-help`

  Print a synopsis of standard options.
- `-Akey[=value]`

  Options to pass to annotation processors.
- `-X`

  Print a synopsis of nonstandard options.
- `-J`*flag*

  Pass *flag* directly to the runtime system.

normalizer.bat Command

Use the normalizer.bat command to run the Normalizer tool to generate application modules from applets created for previous version of the Java Card platform. The output from the tool is a classic module that contains the class files, the CAP components of the CAP file, SIO proxies for classic SIOs (if required), and associated classic application descriptors. The input to the tool must be classic CAP files and associated EXP files. If the input files are not classic CAP files, the normalization will fail.

Assuming that *JC_CONNECTED_HOME*\bin is on your path, use the following syntax to run the Normalizer:

```
normalizer.bat subcommand [options]
```

A subcommand must be the first argument on the command line after the normalizer.bat command. Valid subcommands for the normalizer.bat command can be any one of the following:

- normalize - Creates package class files.
- copyright - Displays detailed copyright notice.

  There are no options for the copyright subcommand.
- help - Displays information about the Normalizer command.

normalize Subcommand and Options

The normalize subcommand creates the package class files. Options are used to specify input files, export paths, export file names, and output directories.

- -i (or --in) *filename*

  Specifies the input CAP file name.
- -p (or --exportpath) *path*

  Specifies the path of the export files used by the tool.
- -o (or --out) *directory*

  (Optional) This the default setting and does not have to be explicitly set. Specifies the output directory that contains the export file.

normalize Subcommand Format

```
normalizer.bat normalize  --in file --out directory
```

`normalize` Subcommand Example

The following is an example of the normalize normalize subcommand:

```
normalizer.bat normalize -i myCAP.cap
```

`help` Subcommand and Options

Options are used to display summary information about the Normalizer subcommands and specific information about individual Normalizer sub-commands.

Summary Help Option

The following command displays summary help about the Normalizer tool:

```
normalizer.bat help
```

`normalize` Help Option

The following command displays help about the `normalize` subcommand:

```
normalizer.bat help normalize
```

`packager.bat` Command

Use the `packager.bat` command to run the Packager tool for creating signature information from application modules and for validating application modules. The Packager tool is located at *JC_CONNECTED_HOME*`\bin`. See Chapter 7 for detailed information about the Packager tool.

Assuming that *JC_CONNECTED_HOME*`\bin` is on your path, use the following syntax to run the packager:

`packager.bat` *subcommand* `[options]` *module-or-folder*

A subcommand must be the first argument on the command line after the `packager.bat` command. Valid subcommands for the `packager.bat` command can be any one of the following:

- `create` - Creates the application module from given module folder or file.

Can have one or more options. See "`create` Subcommand and Options" on page 169 for the list of valid options.

- `validate` - Validates a specified application module.

  Can have one option. See "`validate` Subcommand" on page 171 for the list of valid options.

- copyright - Displays detailed copyright notice.
- `help` - Displays usage information.

  Use help *sub-command* to get more details on each sub-command

  For example, to display detailed help about the `create` sub-command:

```
$>packager.bat help create
```

`create` Subcommand and Options

Options must follow the subcommand on the command line. Valid `create` subcommand options are as follows:

- `-A` (or `--alias`) *alias*

  Application signing attribute, where *alias* is the name used to retrieve the key from the keystore.

- `-c` (or `--compress`)

  (Optional) If specified, the Packager tool compresses the output Java Card 3 platform application with a deflate algorithm. If not specified, the Packager creates an uncompressed JAR file.

- `-e` (or `--exportpath`)

  (Optional) If specified, sets the export files path. System's `api_export` files are implicitely loaded.

- `-f` (or `--force`)

  (Optional) If specified, descriptors or application module assembly problems are automatically corrected when possible.

- `-K` (or `--keystore`) *keystore-file*

  Application signing attribute, where *keystore-file* is the path and filename where private keys are stored. A key utility such as the JDK keytool must be used to create and maintain this file.

- `-n` (or `--nowarn`)

  Suppress the warning messages.

- `-o` (or `--out`) *file-name*

  Where *file-name* specifies the name of the output file.

- `-p` (or `--packageaid`)

  (Optional) If specified, sets the package AID in //aid/<RID>/<PIX> format for `classic-lib`. Ignored if type is not `classic-lib`.

- `-P` (or `--passkey`) *key-password*

  Application signing attribute. Where *key-password* is the password for the private key.

- `-s` (or `--sign`)

  Specifies that the Packager sign the application. If `--sign` is specified, `--keystore` *keystore-file*, `--storepass` *keystore-password*, `--passkey` *key-password,* and `--alias` *alias* are required.

- `-S` (or `--storepass`) *keystore-password*

  Application signing attribute, where *keystore-password* is the password for the keystore.

- `-t` (or `--type`) *file-type*

  Where *file-type* is one of the following:

  - `web` (default)
  - `classic-lib`
  - `classic-applet`
  - `extension-lib`
  - `extended-applet`

`create` Subcommand Format

```
packager.bat create --out file-name [--type file-type] \
          [--exportpath path-of-export-files] \
          [--packageaid package-AID-for-classic-lib] \
          [--sign --storepass keystore-password --passkey key-password \
          --alias alias] [--compress] [--force] [--nowarn] \
          module-or-folder
```

`create` Subcommand Example

The following is an example of the packager `create` subcommand:

```
packager.bat create -o mymodule.jar -t web -c c:\mymodulefolder
```

`validate` Subcommand

The validate subcommand has a single option `-t` (or `--type)` that specifies the type of application module to be validated.

`validate` Subcommand Format

```
packager.bat validate [--type web | classic-lib | classic-applet | \
          extension-lib | extended-applet] \
          module-file-name (or module-directory-name)
```

`validate` Subcommand Example

The following is an example of the packager `validate` subcommand:

```
packager.bat validate -t web myapp.war
```

`copyright` Subcommand

There are no options for the `copyright` subcommand.

`copyright` Subcommand Format

The following is an example of the `copyright` subcommand format:

```
packager.bat copyright
```

`copyright` Subcommand Example

The following is an example of the `copyright` subcommand:

```
packager.bat copyright
```

`help` Subcommand

There are no options, but the `help` subcommand accepts a topic attribute of a specific subcommand name for which detailed information is displayed.

`help` Subcommand Format

The following is an example of the `help` subcommand format:

`packager.bat help` *subcommand topic*

`help` Subcommand Example

The following is an example of the `help` subcommand:

```
packager.bat help validate
```

`romizer.bat` Command

Use the ROMizer tool to create a ROM image used in building a custom Java Card runtime environment (`cjcre.exe`). See Chapter 13 for detailed information about the ROMizer tool.

Use the following command format to run the ROMizer tool:

`romizer.bat` [`options`] *api jar file path*

Options that can be used in the `romizer.bat` command include:

- `-o` (or) `--out` *out-file*

  Optional. Output file name. Default is `ROMJava.c`

- `-e` (or) `--e2pfile` *eeprom-file*

  Optional. File where the initial EEPROM image is stored. Default is `cjcre.eeprom`.

- `-a` (or) `--apps` *apps-list-file*

  Optional. The file contains a list of applications in separate lines.

  The format of each line in the *apps-list-file* is:

  application-file `-t` *type-of-application* `-s` *signature-file* `-n` *name*

  In the previous format example, the following parameters are used:

  - *application-file* - Absolute (or) relative path of the application file
  - *type* - One of `web`, `extended-applet`, `classic-applet`, `extension-lib` or `classic-lib`
  - *signature-file* - Signature file that is the same as the file used by the Installer tool.

- *name* - Name for this application bundle

Examples

```
romizer.bat --out MyROMJava.c --e2pfile my.eeprom yourapi.jar

romizer.bat ..ıib\api_connected.jar

romizer.bat yourapi.jar

romizer.bat -a myapps.list yourapi.jar
```

In the example, the contents of `myapps.list` might be in the form of:

```
helloworld1.war -t web -s helloworld1.signature -n hello1
helloworld2.eap -t extended-applet -s helloworld2.signature -n hello2
```

Glossary

3GPP Third Generation Partnership Project (3GPP) formed by telecommunications associations to develop 3rd Generation Mobile System specifications for systems deployed across the GSM market. These specifications are available on the 3GPP web site.

AID (application identifier) defined by ISO 7816, a string used to uniquely identify card applet applications and certain types of files in card file systems. An AID consists of two distinct pieces: a 5-byte RID (resource identifier) and a 0 to 11-byte PIX (proprietary identifier extension). The RID is a resource identifier assigned to companies by ISO. The PIX identifiers are assigned by companies.

A unique AID is associated with each applet class in an applet application module. In addition, a unique AID is assigned to each applet instance during installation. This applet instance AID is used by an off-card client to select the applet instance for APDU communication sessions.

Applet instance URIs are constructed from their applet instance AID using the "aid" registry-based namespace authority as follows:

`//aid/<RID>/<PIX>`

where `<RID>` (resource identifier) and `<PIX>` (proprietary identifier extension) are components of the AID.

Ant a platform-independent software tool written in the Java programming language that is used for automating build processes.

APDU an acronym for Application Protocol Data Unit as defined by ISO 7816-4 specifications. ISO 7816-4 defines the application protocol data unit (APDU) protocol as an application-level protocol between a smart card and an application on the device. There are two types of APDU messages, command APDUs and response APDUs. For detailed information on the APDU protocol see the ISO 7816-4 specifications.

APDU-based application environment consists of all the functionalities and system services available to applet applications, such as the services provided by the applet container.

API an acronym for Application Programming Interface. The API defines calling conventions by which an application program accesses the operating system and other services.

applet a stateless software component that can only execute in a container on the client platform. Within the context of this document, a Java Card applet, which is the basic component of applet-based applications and which runs in the APDU application environment.

applet application an application that consists of one or more applets.

applet container contains applet-based applications and manages their lifecycles through the applet framework API. Also provides the communication services over which APDU commands and responses are sent.

applet framework an API that enables applet applications to be built.

application descriptor see *descriptor*.

application developer The producer of an application. The output of an application developer is a set of application classes and resources, and supporting libraries and files for the application. The application developer is typically an application domain expert. The developer is required to be aware of the application environment and its consequences when programming, including concurrency considerations, and create the application accordingly.

application group a set of one or more applications executing in a common group context.

application URI a URI uniquely identifying an application instance on the platform.

atomicity a property of transactions that requires all operations of a transaction be performed successfully for the transaction to be considered complete. If all of a transaction's operations cannot be performed, none of them can be performed.

classic applet applets with the same capabilities as those in previous versions of the Java Card platform and in the Classic Edition.

Classic Edition one of the two editions in the Java Card 3 Platform. The Classic Edition is based on an evolution of the Java Card Platform, Version 2.2.2 and is backward compatible with it, targeting resource-constrained devices that solely support applet-based applications.

Connected Edition one of the two editions in the Java Card 3 Platform. The Connected Edition has a significantly enhanced runtime environment and a new virtual machine. It includes new network-oriented features, such as support for web applications, including the Java™ Servlet APIs, and also support for applets with extended and advanced capabilities. An application written for or an implementation of the Connected Edition may use features found in the Classic Edition.

Converter a peice of software that preprocesses all of the Java programming language class files of a classic applet application that make up a package, and converts the package into a standalone classic applet application module distribution format (CAP file). The Converter also produces an export file.

create indicates that a web application of a *module* or an application group, that was loaded by *load*, needs to be created. As a result, the required application is accessible through some Web-Context root.

delete indicates that a web application instance created by *create* needs to be deleted.

ETSI the European Telecommunications Standards Institute (ETSI) is an official European Standards Organization that develops and publishes standards for information and communications technologies. Additional information is available on the ETSI web site.

descriptor a document that describes the configuration and deployment information of an application. A deployment descriptor conveys the elements and configuration information of an application between application developers, application assemblers, and deployers. A runtime descriptor describes the configuration and deployment information of an application that are specific to an operating environment to which the application is to be deployed.

distribution format structure and encoding of a distribution or deployment unit intended for public distribution.

extended applet an applet with extended and advanced capabilities (compared to a classic applet) such as the capabilities to manipulate `String` objects and open network connections.

garbage collection the process by which dynamically allocated storage is automatically reclaimed during the execution of a program.

global array an applet environment array objects accessible from any context.

global authentication the scope of a user authentication that can be tracked globally (card-wide). Global authentication is restricted to card-holder-users. Authorization to access resources protected by a globally authenticated card-holder-user identity is granted to all users.

GlobalPlatform (GP) an international association of companies and organizations that establish and maintain interoperable specifications for single and multi-application smart cards, acceptance devices, and infrastructure systems. Additional information is available on the GlobalPlatform web site.

group context protected object space associated with each application group and Java Card RE. All objects owned by an application belong to the context of the application group.

ISO the International Standards Organization (ISO) is a non-governmental organization of national standards institutes that develops and publishes international standards for both public and private sectors. Additional information is available on the ISO web site.

JAR file an acronym for Java Archive file, which is a file format used for aggregating and compressing many files into one.

Java Card Runtime Environment consists of the Java Card virtual machine and the associated native methods.

Java Card Virtual Machine (Java Card VM) a subset of the Java virtual machine, which is designed to be run on smart cards and other resource-constrained devices. The Java Card VM acts an engine that loads Java class files and executes them with a particular set of semantics.

JDK software an acronym for Java Development Kit. The JDK software is a Sun Microsystems, Inc. product that provides the environment required for software development in the Java programming language. The JDK software is available for a variety of operating systems, for example Sun Microsystems Solaris OS and Microsoft Windows.

KVM a virtual machine for small devices, the KVM is derived from the Java virtual machine (JVM) but is written in the C programming language and has a smaller footprint than the JVM. The KVM supports a subset of the JVM features.

list indicates that the client is requesting information about all loaded application groups and instances.

load indicates that a *module* or an application group needs to be deployed onto the card but not yet made accessible.

mask production (masking) refers to embedding the Java Card virtual machine, runtime environment, and applications in the read-only memory of a smart card during manufacture.

mode (communication) designates the type or protocol of communication (HTTPS, SSL/TLS, SIO...) and the mode of operation (client or server) that characterizes a communication endpoint.

module a unit of distribution and deployment of component applications. Modules or component applications are individual applications (standalone) and can be assembled into application groups. Applications that rely on a single component application can be deployed directly as standalone application modules in addition to deployment as application groups.

MMC MultiMediaCard (MMC) is a flash memory card standard developed and published by the MultiMediaCard Association.

namespace a set of names in which all names are unique.

non-volatile memory — memory that is expected to retain its contents between card tear and power up events or across a reset event on the smart card device.

normalization (classic applet) — the process of transforming and repackaging a Java application packaged for the Java Card Platform, Version 2.2.2, for deployment on both the Java Card 3 Platform, Connected Edition and the Java Card 3 Platform, Classic Edition.

normalization (URI) — the process of removing unnecessary "." and ".." segments from the path component of a hierarchical URI.

Normalizer — in the Connected Edition, a backwards compatibility tool that allows Java applications programmed for the Java Card Platform, Version 2.2.2, to be deployed on both the Java Card 3 Platform, Connected Edition and on the Java Card 3 Platform, Classic Edition. It also allows Java applications packaged for Version 2.2.2 to be transformed through the normalization process and then repackaged for deployment on both the Connected and Classic Editions.

In the Classic Edition, a compatibility tool that enables developers to generate application modules for Java Card 3 platform classic applets they are creating or from classic applets created for previous versions of the Java Card platform. These application modules contain CAP files and are downloadable on both the Java Card 3 platform Classic Edition and Connected Edition smart cards.

off-card client — see *off-card client application*.

off-card client application — an application that is not resident on the card, but runs at the request of a user's actions.

off-card installer — the off-card application that transmits the application and library executables to the card manager application running on the card.

package — a namespace within the Java programming language that can have classes and interfaces.

platform protection domain — a set of permissions granted to an application or group of applications by the platform security policy. A platform protection domain is defined by two sets of permissions: a set of included permissions that are granted and a set of excluded permissions that are denied and can never be granted.

platform security policy — the permission-based security policy that maps application models to sets of permissions granted to applications implementing these application models. For each of the application models, the platform security policy guarantees the consistency and integrity of the applications implementing the application model.

protected content — see *protected resource*.

protected resource — an application or system resource that is protected by an access control mechanism.

protection domain — a set of permissions granted to an application or group of applications.

RAM (random access memory) — temporary working space for storing and modifying data. RAM is non-persistent memory; that is, the information content is not preserved when power is removed from the memory cell. RAM can be accessed an unlimited number of times and none of the restrictions of EEPROM apply.

reference implementation — a fully functional and compatible implementation of a given technology. It enables developers to build prototypes of applications based on the technology.

reference applications — blue print-like applications that demonstrate the interactions between various applications on the card using advanced features such as SIO and events.

remote user — an user whose identity may be assumed by a remote entity, such as a remote card administrator.

remotely accessible web application — an application that is not expected to interact with the card holder but with other-users, potentially remote.

restartable task — an object implementing the `Runnable` interface that has been registered for recurrent execution over card sessions. A task executes in its own thread.

restartable task registry — a Java Card RE facility that is used for registering tasks for recurrent execution over card sessions.

security requirements — the required security characteristics for a particular secure communication being established by either an application or by the web container on behalf of a web application.

server application — an on-card application that provides a service to its clients.

service — a shareable interface object that a server application uses to provide a set of well-defined functionalities to its clients.

service facility — a Java Card RE facility (or subsystem) that is used for inter-application communications.

service factory — an object that the Java Card RE invokes to create a service - on behalf of the server application that registered that service - for a client application that looked up the service.

service registry — the core component of the service facility. The service facility is used for registering and looking up services.

service URI — a URI that uniquely identifies a service provided by a server application.

servlet — a web application component, managed by a container, that generates dynamic web content and that runs in the web application environment.

servlet container — see *web application container*.

servlet context — a container-managed object that defines a servlet's view of the web application within which the servlet is running. A servlet context is rooted at a known path within a web server: a context path.

servlet mapping — a servlet definition that is associated by a servlet container with a URL path pattern. All requests to that path pattern are handled by the servlet associated with the servlet definition. See *Java Servlet Specification, Connected Edition*.

shareable interface — an interface that defines a set of shared methods. These interface methods can be invoked from an application in one group context when the object implementing them is owned by an application in another group context.

shareable interface object (SIO) — an object that implements the shareable interface.

shareable interface object-based service — see *service*.

smart card — a card that stores and processes information through the electronic circuits embedded in silicon in the substrate of its body. Unlike magnetic stripe cards, smart cards carry both processing power and information. They do not require access to remote databases at the time of a transaction.

SSL — Secure Socket Layer (SSL), like the later *TLS* protocol, is a cryptographic protocol for securely transmitting documents by using a two key cryptographic system (a public key and a private key) to encrypt and decrypt data.

terminal — is typically a computer in its own right with an interface which connects with a smart card to exchange and process data.

thread — the basic unit of program execution. A process can have several threads running concurrently each performing a different job, such as waiting for events or performing a time consuming job that the program doesn't need to complete before going on. When a thread has finished its job, it is suspended or destroyed.

thread's active context — when an object instance method is invoked, the owning context of the object becomes the currently active context for that particular thread of execution. Synonymous with *currently active context*.

transaction — an atomic operation in which the developer defines the extent of the operation by indicating in the program code the beginning and end of the transaction.

transaction facility — a Java Card RE facility that enables an application to complete a single logical operation on application data atomically, consistently and durably within a transaction.

transient object the state of transient objects do not persist from one card session to the next, and are reset to a default state at specified intervals. Updates to the values of transient objects are not atomic and are not affected by transactions.

transferable classes classes whose instances can have their ownership transferred to a context different from their currently owning context. Transferable classes are of two types:

Implicitly transferable classes - Classes whose instances are not bound to any context (group contexts or Java Card RE context) and can, therefore, be passed and shared between contexts without any firewall restrictions. Examples are `Boolean` and literal `String` objects.

Explicitly transferable classes - Classes whose instances must have their ownership explicitly transferred to another application's group context in order to be accessible to that other application. Examples are arrays and newly created `String` objects.

transfer of ownership a Java Card RE facility that allows for an application to transfer the ownership of objects it owns to an other application. Only instances of transferable classes can have their ownership transferred.

trusted client an on-card or off-card application client that an on-card application trusts on the basis of credentials presented by the client.

trusted client credentials credentials that an on-card application uses to ascertain the identity of clients it trusts.

TLS Transport Layer Security (TLS), like the earlier *SSL* protocol, is a cryptographic protocol for securely transmitting documents either by endpoint authentication of the server or by mutual authentication of the server and the client.

unload indicates that the module or application group that was loaded by *load* needs to be removed completely from the card. By default, if there are some instance(s) created, then unload will fail. Optional -f (or –force) will attempt to delete all instances before unloading.

uniform resource identifier (URI) a compact string of characters used to identify or name an abstract or physical resource. A URI can be further classified as a uniform resource locator (URL), a uniform resource name (URN), or both. See RFC 3986 for more information.

uniform resource locator (URL) a compact string representation used to locate resources available via network protocols or other protocols. Once the resource represented by a URL has been accessed, various operations may be performed on that resource. See RFC 1738 for more information. A URL is a type of uniform resource identifier (URI).

USB Universal Serial Bus (USB) is a serial bus specification developed and published by the USB Implementers Forum that when implemented enables external devices such as flash drives, PDAs, and printers to connect to a host controller.

verification a process performed on an application or library executable that ensures that the binary representation of the application or library is structurally correct.

volatile memory memory that is not expected to retain its contents between card tear and power up events or across a reset event on the smart card device.

volatile object an object that is ideally suited to be stored in volatile memory. This type of object is intended for a short-lived object or an object which requires frequent updates. A volatile object is garbage collected on card tear (or reset).

web application a collection of servlets, HTML documents, and other web resources that might include image files, compressed archives, and other data. A web application is packaged into a web application archive.

All compatible servlet containers must accept a web application and perform a deployment of its contents into their runtime. This may mean that a container can run the application directly from a web application archive file or it may mean that it will move the contents of a web application into the appropriate locations for that particular container. See *Java Servlet Specification, Connected Edition*.

web application archive the physical representation of a web application module. A single file that contains all of the components of a web application. This archive file is created by using standard JAR file tools, which allow any or all of the web components to be signed.

A web application archive file is identified by the `.war` extension and is often referred to as a WAR file. A new extension is used instead of `.jar` because that extension is reserved for files which contain a set of class files and that can be placed in the classpath. As the contents of a web application archive are not suitable for such use, a new extension was required. See *Java Servlet Specification, Connected Edition*.

web application container contains and manages web applications and their components (for example, servlets) through their lifecycle. Also provides the network services over which HTTP requests and responses are sent and manages security of web applications.

web application environment in addition to the Java Card RE, consists of all the functionalities and system services available to web applications, such as the services provided by the web application container.

web client an off-card entity that requests services from an on-card web application. A typical example is a web browser.

Index

D

E

F

I

J

K

L

M

N

O

P

R

S

T

U

W