



Development Kit User's Guide

Java Card™ 3 Platform, Version 3.0.2
Connected Edition

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 exclusivement 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 législation américaine 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 nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou reexportations vers des pays sous embargo des États-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière 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 regis par la législation américaine sur le contrôle des exportations et la liste de ressortissants spécifiquement désignés, 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 AUTORISÉE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE À LA QUALITÉ MARCHANDE, À L'APTITUDE À UNE UTILISATION PARTICULIÈRE OU À L'ABSENCE DE CONTREFAÇON.



Contents

Preface xv

Part I **Setup, Samples and Tools**

1. Introduction 1

Platform Architecture 2

Development Kit Description 3

 Connected Edition Features 4

 Connected Edition Security Model 5

 Application Models 6

 Development Kit Contents 6

 Reference Implementation 6

 Command Line Tools 7

 Samples 8

System Requirements 8

Additional Software 8

Java Card TCK 9

2. Installation 11

Prerequisites to Installing the Development Kit 11

Install and Setup the Development Kit 12

▼	Installing the Development Kit	12
▼	Setting Up the System Variables	14
	Installed Directories and Files	16
	Directories and Files Installed From All Bundles	16
	Subdirectories and Files Installed in the <code>src</code> Directory	18
	Uninstall the Development Kit	19
	Install and Setup the NetBeans IDE	20
▼	Installing the NetBeans IDE	20
▼	Setting Up the Java Card Platform	20
3.	Developing Java Card 3 Platform Applications	23
	Development Steps	23
4.	Using the Samples	27
	Running the Samples	28
▼	Running the Samples from the NetBeans IDE	28
▼	Accepting an Untrusted Certificate	29
	Using the Web Application Sample	29
	Using the HelloWorld Sample	30
▼	Run HelloWorld	30
	Using the Classic Applet Sample	31
	Using the Extended Applet Sample	31
5.	Starting the Java Card Runtime Environment	33
	Starting <code>cjcre.exe</code> from the Command Line	33
	<code>cjcre.exe</code> Command Line Options	34
	Java Card Runtime Environment Configuration Files	36
	Adding Proprietary Packages	36
6.	Compiling Source Code	37

Running the Compiler Tool from the Command Line	37
Compiler Tool Options	37
Format	38
Examples	39
7. Creating and Validating Application Modules	41
Packager Operation	41
Options	41
Basic Packaging Sequence	42
Use Cases	42
Signing	43
Use Cases	43
Running the Packager From the Command Line	43
create Subcommand	43
create Subcommand Format	44
create Subcommand Options	44
create Subcommand Examples	46
validate Subcommand	46
validate Subcommand Format	47
validate Subcommand Options	47
validate Subcommand Example	47
copyright Subcommand	48
copyright Subcommand Format	48
copyright Subcommand Options	48
copyright Subcommand Example	48
help Subcommand	48
help Subcommand Format	48
help Subcommand Options	48
help Subcommand Example	49

8. Loading and Managing Applications 51

Description of the On-Card Installer 51

On-card Installer Operation 52

On-card Installer Functionality 52

Description of the Installer Tool 53

Running the Installer Tool From the Command Line 53

`load` Subcommand 54

`create` Subcommand 56

`delete` Subcommand 58

`unload` Subcommand 60

`list` Subcommand 61

`help` Subcommand 63

Card Installer Use Case 63

Load an Application 64

Pre-Conditions 64

Post-Conditions 64

Sequence of Events 64

9. Backwards Compatibility for Classic Applets 65

Generating Application Modules From Classic Applets 65

Running the Normalizer From the Command Line 66

`normalize` Subcommand 67

`copyright` Subcommand 68

`help` Subcommand 68

Converting Class Files to CAP Files 69

Conversion Process Sequence 70

Specifying an Export Map 71

	Loading Export Files	71
	Creating a <code>debug.msk</code> Output File	72
	Verification of Input and Output Files	72
	File and Directory Naming Conventions	72
	Input File Naming Conventions	73
	Output File Naming Conventions	73
	Running the Converter From the Command Line	74
	<code>converter</code> Command Options	74
	Using a Command Configuration File	76
	Using Delimiters with Command Line Options	76
10.	Using the APDU Tool	77
	Running the APDU Tool From the Command Line	77
	Examples of Using the APDU Tool	78
	Directing Output to the Console	79
	Directing Output to a File	79
	Using APDU Script Files	79
	APDU Script File Commands	80
	APDU Script Preprocessor Commands	80
11.	Debugging Applications	83
	Debugger Architecture	83
	Running the Debugger From the Command Line	84
	<code>debug</code> Subcommand	85
	<code>copyright</code> Subcommand	85
	<code>help</code> Subcommand	85
	Debugging a Java Card 3 Platform Application	85
	Compile the Source Code	86
	Start the Debugger	86

Attach the Debugger to the IDE 86
Run `cjcre.exe` With `-debug` Option 86
Set Break Points 86

Part II Programming With the Development Kit

12. Configuring the RI 91

Configuring Authenticators 91
Creating Custom Protection Domains 92
 Creating a Custom Keystore 92
Configuring SSL Support 93
 Adding SSL Support 93
 Custom Certificates and Keys 94
 ▼ Generating an SSL Certificate 94

13. Building the RI From Sources 95

Prerequisites to Building the RI 95
Contents of `JC_CONNECTED_HOME\src` Folder 96
Running the ROMizer Tool From the Command Line 96
 `romize` Subcommand 97
 `romize` Subcommand Options 97
 `romize` Subcommand Example 98
 `copyright` Subcommand 98
 `help` Subcommand 98
 Apps list File Contents 99
 Example Contents of Apps List File 99
 Romizer Tool Output 99
Building a Custom `cjcre.exe` 100
 Preprocessor Symbols to Customize the VM 101
 ▼ Build a Custom RI From the Command Line 102

▼	Test the Custom RI	103
14.	Working with APDU I/O	105
	The APDU I/O API	105
	APDU I/O Classes and Interfaces	105
	Exceptions	106
	Two-interface Card Simulation	107
	Examples of Use	107
	To Connect To a Simulator	107
	To Establish a T=0 Connection To a Card	108
	To Power Up And Power Down the Card	108
	To Exchange APDUs	109
	To Print the APDU	110
A.	Application Module and Library Formats	111
	Web Application Module Format	112
	Extended Applet Application Module Distribution Format	113
	Classic Applet Application Module Format	113
	Extension Library Format	114
	Classic Library Format	115
	Glossary	117
	Index	127

Figures

FIGURE 1-1	Architecture of Connected Edition	3
FIGURE 2-1	Uninstalling the Development Kit	19
FIGURE 3-1	Java Card 3 Platform Application Development	24
FIGURE 9-1	Generating Application Modules From Classic Applets	66
FIGURE 11-1	Debugger Architecture	83
FIGURE 13-1	Building cjcre.exe From Sources	101
FIGURE A-1	Web Application Module Format	112
FIGURE A-2	Extended Applet Application Module	113
FIGURE A-3	Classic Applet Application Module	114
FIGURE A-4	Java Platform Standard Edition Library JAR File Format	115
FIGURE A-5	Classic Library Distribution Format	116

Tables

TABLE 2-1	Directories and Files Installed From All Bundles	16
TABLE 2-2	Contents of the <code>src</code> Directory	18
TABLE 6-1	Compiler Tool Options	37
TABLE 7-1	Packager Tool Input Files and Expected Output	42
TABLE 7-2	Packager Tool Signing Results	43
TABLE 7-3	<code>create</code> Subcommand Options	44
TABLE 7-4	<code>validate</code> Subcommand Options	47
TABLE 7-5	Use Cases for Command Line Arguments	49
TABLE 8-1	<code>load</code> Options	54
TABLE 8-2	<code>create</code> Options	56
TABLE 8-3	<code>delete</code> Options	59
TABLE 8-4	<code>unload</code> Options	60
TABLE 8-5	<code>list</code> Options	61
TABLE 9-1	<code>normalize</code> Subcommand Options	67
TABLE 9-2	<code>converter</code> Command Options	74
TABLE 10-1	<code>apdutool</code> Command Line Options	78
TABLE 10-2	Supported APDU Script File Commands	80
TABLE 11-1	<code>debug</code> Subcommand Options	85
TABLE 13-1	<code>romize</code> Subcommand Options	97

Preface

This document describes how to use the development kit for the Java Card 3 Platform, Connected Edition, Version 3.0.2, to develop applet applications, web applications, servlets, and extended applets. The Java Card 3 Platform currently includes Versions 3.0, 3.0.1, and 3.0.2 of Java Card technology.

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.

Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* and *Programming Notes, Java Card 3 Platform, Connected Edition* for additional information about creating extended applets. You must download the Java Card specifications bundle and the Programming Notes book 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 tools from the command line. The NetBeans IDE, Version 6.8 or higher, is required to run the samples and is suggested as your development environment.

Who Should Use This Document

The *Development Kit User's Guide, Java Card 3 Platform, Version 3.0.2, Connected Edition* is written for developers who are:

- Creating Java Card 3 web and servlet applications or extended applet applications for the Connected Edition.
- Creating classic applet applications for the Classic or Connected Editions.
- 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 required for this release.

[Chapter 3](#) provides a brief description of the steps involved in Java Card application development.

[Chapter 4](#) describes where to find the entire set of samples designed for the Java Card 3 platform, with further details for the set of samples included in this development kit. To run the samples, the NetBeans IDE, Version 6.8, is required.

[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 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, including how to generate and install SSL keys.

[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 customized Java Card 3 platform RI according to their specific requirements.

[Chapter 14](#) describes the APDU I/O API, which is a library used by development kit components, such as apdutool.

[Appendix A](#) describes the application module and library formats supported by the Java Card 3 platform card manager.

[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 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*
- *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)

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.

Typographic Conventions

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

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:

jc3-ri-feedback@sun.com

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

Development Kit User's Guide, Java Card 3 Platform, Version 3.0.2, Connected Edition

PART I Setup, Samples and Tools

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

Introduction

The Java Card 3 Platform, Version 3.0.2 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. However, the development kit for the Connected Edition, and the NetBeans IDE can be used to create classic applets that will also run on the Classic Edition RE.

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

This chapter contains the following sections:

- [Platform Architecture](#)
- [Development Kit Description](#)
- [System Requirements](#)
- [Additional Software](#)
- [Java Card TCK](#)

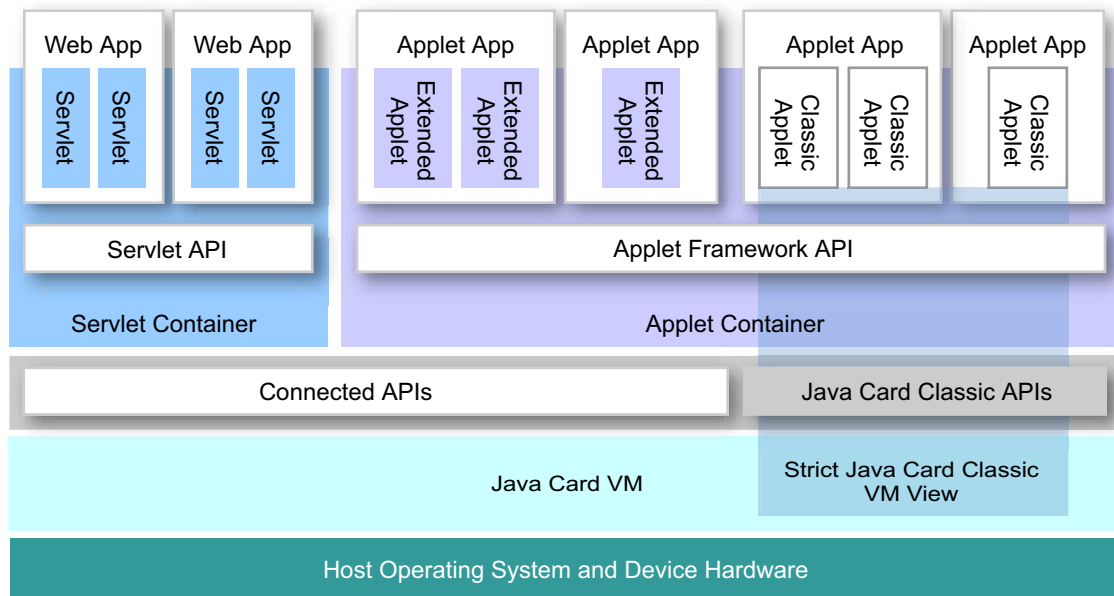
Platform Architecture

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, which are for web applications.

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.

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

This development kit describes how to use the command-line tools included in this bundle. It enables you to create 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. Any valid application written for, or any valid implementation of, the Connected Edition may also use features found in the Classic Edition.

Note – In this release, you will be able to use the development kit command-line tools or the NetBeans IDE to create applications for both Classic and Connected Editions. The NetBeans IDE is the suggested development environment. For details on using the NetBeans IDE for development, see the Java Card platform-specific online help provided in version 6.8 of the NetBeans IDE under Help > Help Contents. For details on programming for the Classic Edition, please see the user's guide in the Classic Edition development kit.

This development kit includes 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 can use the development kit tools to create applications that fully utilize the features of the Connected Edition.

Connected Edition Features

Developers creating implementations or 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:

- File system
- 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 contacted/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. Each JAR file 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 uninstaller tool for the development kit. The source bundles include the binaries, and 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 KitCommand Line Tools

[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`.
- **Uninstaller Tool** - Safely uninstalls this development kit. See [“Uninstall the Development Kit” on page 19](#).

Samples

The Java Card 3 platform samples provide a demonstration of the features in the Connected Edition and source code that gives an introduction to Java Card 3 platform programming. See [Chapter 4](#) for a description of the samples included with this development kit and where to find additional samples on <http://kenai.com>. The Connected Edition samples must be run from within the plugin provided with the NetBeans IDE, version 6.8, which is available for download from <http://netbeans.org>.

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. However, the NetBeans IDE is strongly recommended because without it, you cannot run the samples.

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 or 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 5.1.4 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** - The NetBeans IDE 6.8, including the Java Card platform plugin, can be used to develop applications and run the samples.

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 3.0.2 tests as described in *Java Card Technology Compatibility Kit, Version 3.0.2 User's Guide*.

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 and the NetBeans IDE, how to set system variables, and how to uninstall the development kit. This chapter also lists the files installed onto your system by the Connected Edition of 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 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 IDE, required to run samples)** - download the NetBeans IDE version 6.8 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.

▼ Installing 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.**

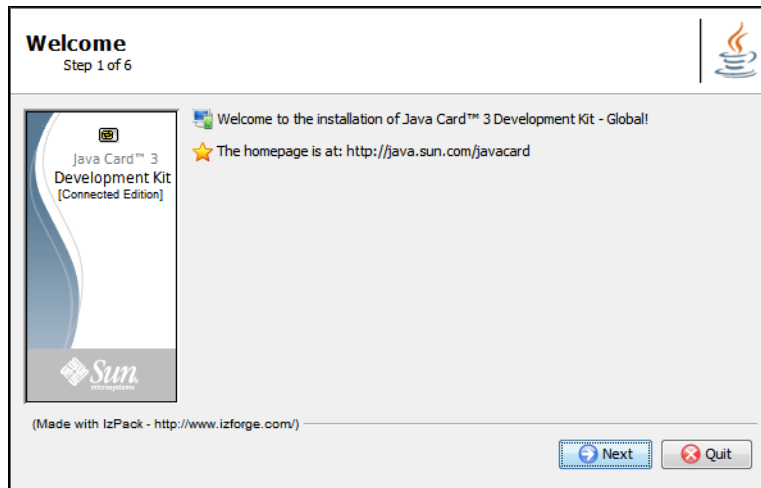
The development kit can be launched automatically when you download the JAR file or by using the Windows file manager tool to navigate to the directory containing the development kit JAR file and double clicking the file name or icon.

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:\JCDK3.0.2_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 “[Directories and Files Installed From All Bundles](#)” on [page 16](#). The files and directories are installed under the root installation directory, either `C:\JCDK3.0.2_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
<code>JAVA_HOME</code>	<code>C:\JAVA\jdk1.6.0_10</code>

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

```
set JAVA_HOME=C:\java_home_path;
```

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

```
set JAVA_HOME=C:\jdk6;
```

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
<code>ANT_HOME</code>	<code>C:\ant\apache-ant-1.6.5</code>

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

```
set ANT_HOME=C:\ANT_HOME_path;
```

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

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

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 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:\JCDK3.0.2_ConnectedEdition or the directory you specified during installation. For example, in the System variables box enter the following:

Variable	Value
JC_CONNECTED_HOME	C:\JCDK3.0.2_ConnectedEdition

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

```
set JC_CONNECTED_HOME=C:\JC_CONNECTED_HOME_path;
```

For example if you installed in C:\JCDK3.0.2_ConnectedEdition, enter:

```
set JC_CONNECTED_HOME=C:\JCDK3.0.2_ConnectedEdition;
```

4. Add %JAVA_HOME%, %JC_CONNECTED_HOME%, and %ANT_HOME% 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.

Installed Directories and Files

A development kit binary bundle installs the subdirectories and files described in [TABLE 2-1](#). A development kit source bundle installs all the subdirectories and files described in [TABLE 2-1](#), plus the source subdirectories and files described in [TABLE 2-2](#).

Directories and Files Installed From All Bundles

These files and directories are installed by the development kit under the root installation directory, `C:\JCDK3.0.2_ConnectedEdition`, or in the directory that you specified during installation.

TABLE 2-1 Directories and Files Installed From All Bundles

Directory or File	Description
<code>COPYRIGHT-software.html</code>	The copyright file for the Java Card 3 platform.
<code>COPYRIGHT-docs.html</code>	The copyright file for the documentation of the Java Card 3 platform.
<code>RELEASENOTES.html</code>	The release notes for this Java Card 3 platform development kit.
<code>document.css</code>	The style sheet for the HTML documentation.
<code>platform.properties</code>	Specifies properties of the Java Card 3 platform RI that are used by the tools.
<code>api_export_files\</code>	Contains <code>java</code> , <code>javacard</code> , and <code>javacardx</code> directories of API export files.
<code>bin\</code>	Contains all shell scripts and batch files (including the <code>cjcre.exe</code> binary executable) used in running the tools.

TABLE 2-1 Directories and Files Installed From All Bundles (*Continued*)

Directory or File	Description
docs\	Contains the following: <ul style="list-style-type: none">• api folder - The Javadoc tool files for the RI API in HTML format.• JCDevKitUG-Connected-3_0_2.pdf - This user's guide.• UserGuide_html folder - The HTML version of this user's guide.• apduio folder - The Javadoc tool files for the publicly available APDU I/O client classes.
legal\	Contains three files: <ul style="list-style-type: none">• 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.
lib\	Contains all Java programming language JAR files and config files required for the tools: <ul style="list-style-type: none">• ant-contrib-1.0b3.jar• api_classic.jar• api_connected.jar• asm-all-3.1.jar• bcel-5.2.jar• commons-cli-1.0.jar• 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• romizer.jar• system.properties• tools.jar
Uninstaller\	Contains the file <code>uninstaller.jar</code> to safely uninstall this development kit.

TABLE 2-1 Directories and Files Installed From All Bundles (*Continued*)

Directory or File	Description
<code>samples\classic_applets</code>	Contains the sample HelloWorld classic applet application adapted to run on the Connected Edition.
<code>samples\extended_applets</code>	Contains the sample extended applet application.
<code>samples\keystore</code>	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.
<code>samples\web</code>	Contains the sample web application.

Subdirectories and Files Installed in the `src` Directory

The `src` directory is installed only from a source bundle and contains the source code for the Java Card API, the romized applications, the development kit tools, and the Java Card virtual machine.

[TABLE 2-2](#) describes the contents of the subdirectories and files installed under the `src` directory.

TABLE 2-2 Contents of the `src` Directory

Directory or File	Description
<code>build.xml</code>	Resource file for rebuilding the development kit source bundle.
<code>apiImpl.jar</code>	
<code>bat.template</code>	
<code>crypto.jar</code>	
<code>api\</code>	Sources for the Java Card API version 3.0.2 in the following subdirectories: <ul style="list-style-type: none"> • <code>com\sun</code> • <code>java</code> • <code>javacard</code> • <code>javacardx</code> • <code>javax</code> • <code>org\mortbay</code>

TABLE 2-2 Contents of the src Directory (*Continued*)

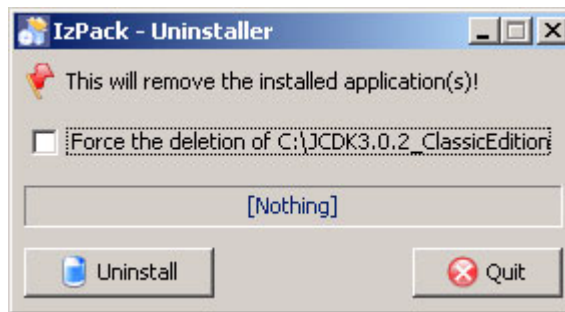
Directory or File	Description
romized_apps\	Sources for the CardManager servlet.
tools\	Sources for development kit tools.
vm\	Sources for the Java Card virtual machine in the following subdirectories and file: c - C programming language sources. h - Header files for the C programming language sources. lib - System and internal web configuration files ignore.list - List of classes ignored by the ROMizer

Uninstall the Development Kit

To uninstall the development kit, Version 3.0.2, run the Uninstaller tool found in your development kit at Uninstaller\uninstaller.jar. Do not change the location of this tool. Before running the Uninstaller, it is advisable to exit all development kit tools and the NetBeans IDE. Files under the control of your OS will not be uninstalled using the Uninstaller.

In the Uninstaller's dialog box, selecting the check box or not will have the same result, the development kit directory that the Uninstaller is in will be deleted, including the Uninstaller itself, see [FIGURE 2-1](#).

FIGURE 2-1 Uninstalling the Development Kit



You can also uninstall a development kit for any Java Card Platform release by simply deleting all its directories and files from your hard drive.

Install and Setup the NetBeans IDE

The NetBeans IDE, version 6.8, is required to run the samples. It is also recommended as your development environment, although alternatively, the development kit tools can be used from the command line.

To use the Java Card platform-specific plugin in the NetBeans IDE, you must add and configure the Java Card Platform.

▼ Installing the NetBeans IDE

1. Go to <http://www.netbeans.org>.

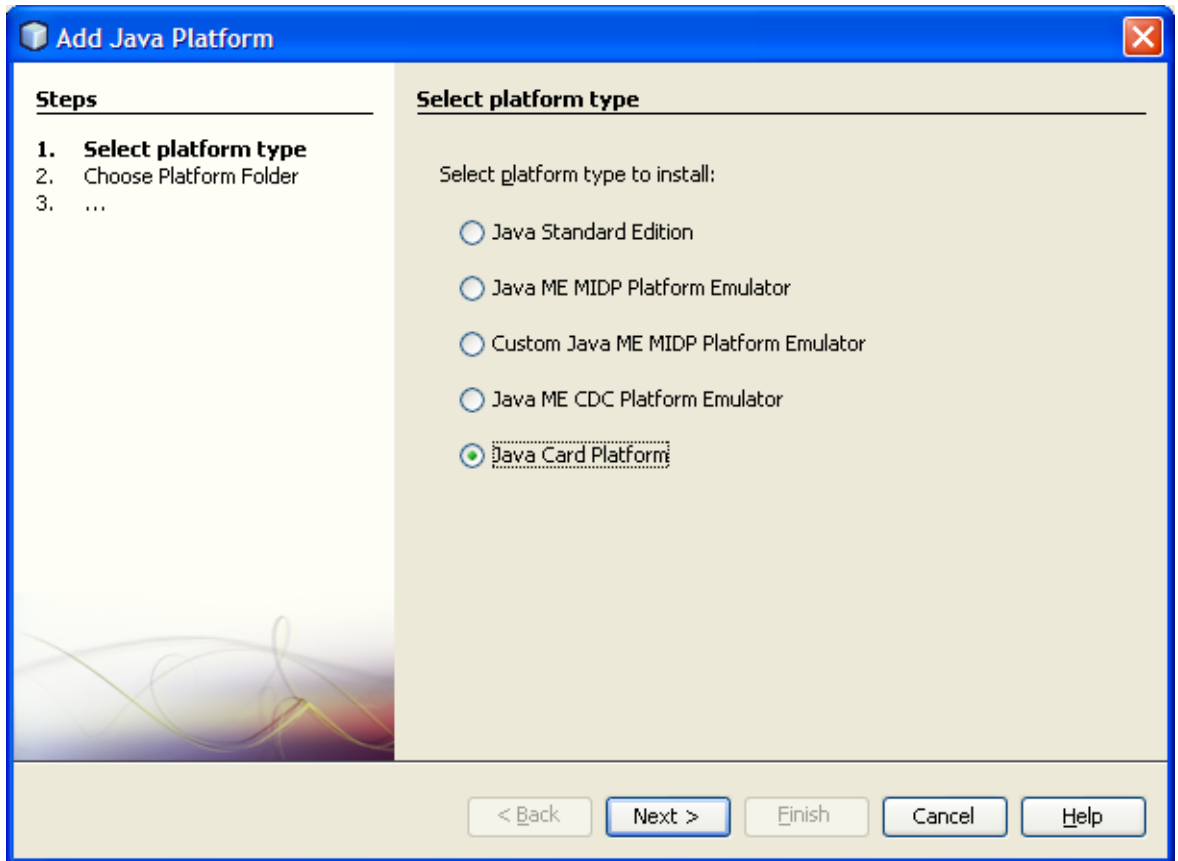
2. Download version 6.8.

Earlier versions of the NetBeans IDE and the plugin will not work with version 3.0.2 of the development kit. Within version 6.8 of the NetBeans IDE, the Java Card platform-specific plugin might already be installed. Check the installed plugins list and, if the Java card plugins are not installed, locate them on the NetBeans IDE update center and install them into the NetBeans IDE.

▼ Setting Up the Java Card Platform

1. In the NetBeans IDE, version 6.8, go to `Tools > Java Platforms` and click `Add Platform`.

You setup the Java Card Platform as you would any other Java Platform. If the Java Card Platform does not appear on the list of platform types, you might need to exit the NetBeans IDE and restart it.



2. **Select** Java Card Platform **and click** Next.

3. **Navigate to and choose the directory where you installed the Java Card Platform development kit and click** Next.

In the documentation for the development kit this directory is referred to as *JC_CONNECTED_HOME*.

4. **Click** Finish.

5. **Click** Close.

Once the installation is complete, there will be a new node, Java Card Runtimes, in the Services window. If the Services window is not already displayed, choose Window, then choose Services to activate it.

6. **Confirm** Java Card Platform **node is listed in** Services **window below the** Java Card Runtimes **node.**

7. **Confirm the Default Device instance is listed in the Services window below the Java Card Platform node.**
8. **In the Tools > Plugins dialog box, confirm the Available Plugins Tab lists the Java Card platform-specific plugin.**

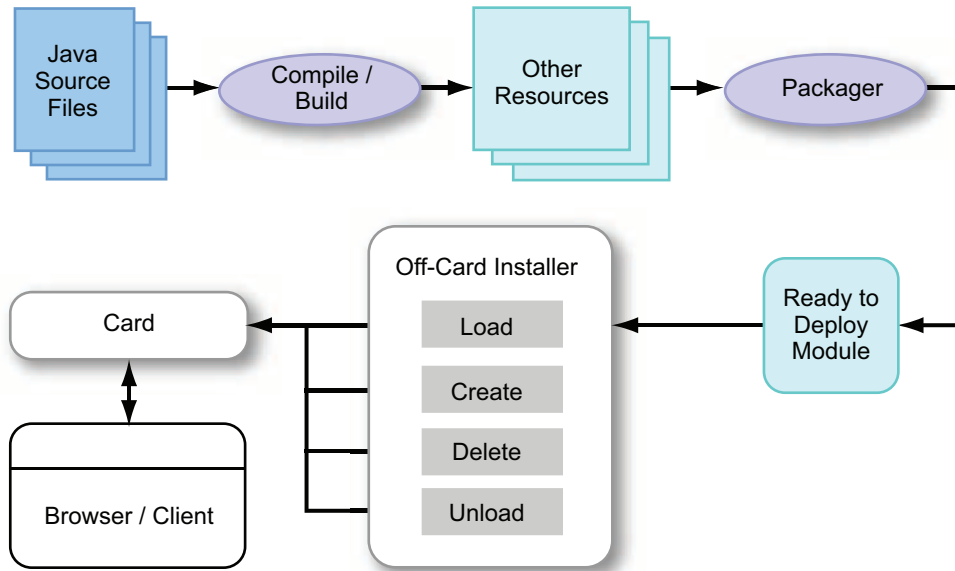
Developing Java Card 3 Platform Applications

This chapter provides a brief description of the activities and development kit tools involved in developing applications for the Java Card 3 platform. Whether you use the development kit tools or the NetBeans IDE as your development environment, these same, basic activities apply. 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.

Using the Samples

The samples included with the development kit demonstrate the basic features of the Java Card API, Connected Edition. The samples in this development kit include a simple web application, extended applet application, and classic applet application. Instructions on how to run the samples in this development kit are available in the online help for the Java Card platform plugin in the NetBeans IDE.

In addition to the three samples in this development kit, many additional samples are available through the NetBeans IDE but are located on the kenai.com developer collaboration website. We recommend you access the instructions on how to run the samples located on kenai.com from within the NetBeans IDE as well, although they are also available through your browser on the kenai.com website at <http://kenai.com/projects/javacard/pages/Home>. The set of samples on kenai.com includes a suite of reference applications that demonstrate an entire application and can be used as a template to illustrate the use of advanced features, such as SIO, event handling, and communication between applications on the card.

Refer to the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* and *Programming Notes, Java Card 3 Platform, Connected Edition* for additional information about designing and writing Java Card 3 applications.

This chapter describes the three samples included in this development kit as they are running in the NetBeans IDE:

- [Running the Samples](#)
- [Using the Web Application Sample](#)
- [Using the Classic Applet Sample](#)
- [Using the Extended Applet Sample](#)

Running the Samples

All samples must be run from within the NetBeans IDE. They cannot be run from the command line in this release of the development kit. Therefore, this section does not describe how to run the samples, but rather how you interact with them as they are running. For detailed instructions on how to run the samples, see the online help provided with the NetBeans IDE under Help > Help Contents.

Note that two sets of samples are available for the Java Card 3 platform, the three basic samples included in this development kit and the full set of samples that can be found at <http://kenai.com/projects/javacard/pages/Home>, where how to run them and their use is described.

▼ Running the Samples from the NetBeans IDE

All Java Card 3 platform samples run only from within the NetBeans IDE, Version 6.8 or higher. However, there are two sets of samples available: the three basic samples included in this development kit and the full set of samples on kenai.com, each of which can easily be made into a project from within the NetBeans IDE.

1. If you have not done so, install and set up the NetBeans IDE.

For the details, see “[Install and Setup the NetBeans IDE](#)” on page 20.

2. If you want to use the samples in this development kit, run the NetBeans IDE, navigate to the Java Card platform project and then to the appropriate sample node.

The three basic samples are listed in the NetBeans IDE but reside in the development kit in subdirectories of `JC_CONNECTED_HOME\samples`. The online help provided in the NetBeans IDE under Help > Help Contents contains information on how to run these three samples.

3. If you want to run the samples that reside on kenai.com, from within the NetBeans IDE go to Team > Kenai > Get Sources from Kenai.

In the Get Sources from Kenai dialog box, click the Browse button beside the Kenai Repository field and use search to locate the project Java Card - Everything You Need to Know. Information on how to run these samples is automatically included with each kenai project you create.

4. In the NetBeans IDE, open the sample project.

The NetBeans IDE loads the built application, and, if used, opens the browser to access the application.

5. When they are run, some sample applications open your default browser.

Your browser displays a web page that serves as the primary user interface to some of the samples. Interactions with the three basic samples in the development kit are described in the following sections.

▼ 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's Certificate Manager to add an exception for the server certificate by performing the following procedure.

- 1. In the Firefox browser menu bar select the `Tools > Options` menu item.**
- 2. In the Options dialog box, select the Advanced icon in the tool bar.**
- 3. Under the Encryption tab, click View Certificates to open the Certificate Manager.**
- 4. Select the Servers tab and click Add Exception.**
- 5. 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`
- 6. Click Get Certificate 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.

Using the Web Application Sample

The simplest web application sample, `HelloWorld`, is included in this development kit. Another version of it is located on the `kenai.com` website. Both versions are accessible through the NetBeans IDE.

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.

Using the HelloWorld Sample

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

Using this sample consists of starting the HelloWorld application, entering a name in the web page, clicking the Say Hello button on the page, and then receiving the the greeting.

▼ Run HelloWorld

1. **Start the HelloWorld application by going to the HelloWorld project node in the NetBeans IDE.**

If you need help with running this sample, go to online help within the NetBeans IDE under Help > Help Contents. When running, a browser displays the following page:



The screenshot shows a web browser window displaying the HelloWorld application. The page has an orange header bar with the text "Hello World" and the Sun logo. Below the header, there is a form with the label "Enter Name:" followed by a text input field. Below the input field is a button labeled "Say Hello". The browser's address bar shows "Java Card™ 3 Platform Connected Edition".

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.



The screenshot shows the same web browser window as before, but now the text "Hello Sun!" is displayed in the main content area of the page. The header bar and the "Say Hello" button are still visible.

Using the Classic Applet Sample

This development kit includes the basic classic applet sample, HelloWorld, in the folder `JC_CONNECTED_HOME\samples\classic_applets`. Another version of it is located on the kenai.com website. Both versions are accessible through the NetBeans IDE.

This sample illustrates basic use of the Java Card API to run a classic applet application and demonstrates the basic structure of a classic applet that developers can use to develop, deploy, create, execute, delete, and unload classic applets. This sample is a minimal classic applet utilizing the simplest source code and meta-files.

This sample contains one project that demonstrates the function of a classic applet. The project is located in the folder `JC_CONNECTED_HOME\samples\classic_applets\HelloWorld`. Information on how to run this sample is located in the online help for the Java Card 3 platform plugin in the NetBeans IDE under Help > Help Contents.

Using this sample consists of starting the HelloWorld classic applet. When running, the project installs the classic applet, processes an incoming APDU, and responds with a greeting.

Using the Extended Applet Sample

This development kit includes the basic extended applet sample, HelloWorld, in `JC_CONNECTED_HOME\samples\extended_applets`. Another version of it is located on the kenai.com website. Both versions are accessible through the NetBeans IDE.

This sample illustrates basic use of the Java Card API to create an extended applet application and demonstrates the basic structure of an extended applet that developers can use to develop, deploy, create, execute, delete, and unload extended applets. This sample is a minimal extended applet utilizing the simplest source code and meta-files.

This sample contains one project that demonstrates the function of an extended applet. The project is located in the folder `JC_CONNECTED_HOME\samples\extended_applets\HelloWorld`. Information on how to run this sample is located in the online help for the Java Card 3 platform plugin in the NetBeans IDE under Help > Help Contents.

Using this sample consists of starting the HelloWorld extended applet. When running, the project installs the extended applet, processes an incoming APDU, and responds with a greeting.

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](#)
- [Java Card Runtime Environment Configuration Files](#)
- [Adding Proprietary Packages](#)

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):

- `-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).
- `-Dname=value`

Supplies a system property (such as `-Dmyproperty=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.
- `-debug <yes|true|no|false>`

Runs `cjcre` in debug mode if you specify `yes` or `true`. Otherwise, the default is `no` and `cjcre` will not be run in debug mode. The values `yes` and `true` are equivalent. The values `no` and `false` are equivalent.
- `-debugport portnumber`

Sets the debug port where the Debug proxy communicates. Valid only when `-debug` is set to `yes`. 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` keyword in Java code).

- `-exactlogger`

Displays only the log messages that match the level set by the `-loggerlevel` option.

- `-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.

- `-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. However, the debug related options (`-debug`, `-debugport`, and `-suspend`) must be specified along with `-resume` to resume the VM in debug mode. The range is 256 bytes to 8K.

- `-suspend <yes|true|no|false>`

Suspends the threads at `cjcre` startup if set to `yes` or `true`. The default is `yes`. The values `yes` and `true` are equivalent. However, `yes` and `true` are valid only when `-debug` is also set to `yes` or `true`. The values `no` and `false` are equivalent.

- `-version`

Displays version information.

- `-Xname=value`

Sets a single configuration property such as:
`-Xmyproperty=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. These are visible using `JCRuntime.getConfigProperty()`.

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.

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
<code>-g</code>	Generate all debugging info
<code>-g:none</code>	Generate no debugging info
<code>-g:{lines,vars,source}</code>	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 <i>path</i>	Specify where to find user class files and annotation processors
-cp <i>path</i>	Specify where to find user class files and annotation processors.
-sourcepath <i>path</i>	Specify where to find input source files.
-bootclasspath <i>path</i>	Override location of bootstrap class files.
-extdirs <i>dirs</i>	Override location of installed extensions.
-endorseddirs <i>dirs</i>	Override location of endorsed standards path.
-proc: {none, only}	Control whether annotation processing and/or compilation is done.
-processor <i>class1[,class2,class3...]</i>	Names of the annotation processors to run; bypasses default discovery process.
-processorpath <i>path</i>	Specify where to find annotation processors.
-d <i>directory</i>	Specify where to place generated class files.
-s <i>directory</i>	Specify where to place generated source files.
-implicit: {none, class}	Specify whether or not to generate class files for implicitly referenced files.
-encoding <i>encoding</i>	Specify character encoding used by source files.
-source <i>release</i>	Provide source compatibility with specified release.
-target <i>release</i>	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.
-Jflag	Pass <i>flag</i> 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:\JCDK3.0.2_ConnectedEdition\bin>javacardc.bat UsesFloat.java  
Java Card 3.0.2 Compiler  
UsesFloat.java:2: float keyword used  
    float f = 0;  
      ^  
1 error
```

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

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](#)
- [Running the Packager From the Command Line](#)

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 malformed 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 <code>--type</code> 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 43. 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.

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 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] [--keepproxysource directory]
--useproxyclass classpath [--nowarn] module-file-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 <i>alias</i> or --alias <i>alias</i>	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.
-C <i>command-options-file</i> or --commandoptionsfile <i>command-options-file</i>	Optional. Specifies a file containing command line options.
-e <i>path-of-export-files</i> or --exportpath <i>path-of-export-files</i>	Specifies the export files path. System's api_export files are implicitly loaded.
-f or --force	Optional. If specified, overwrite the output file. See “--force Option Behavior” on page 45.
-k <i>directory</i> or --keepproxysource <i>directory</i>	Optional. Cannot be used with --useproxyclass. Creates the proxy source files and other stub files in the specified directory.

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

Options	Description
-K <i>keystore-file</i> or --keystore <i>keystore-file</i>	Required only when the --sign option is specified. Application signing attribute, where <i>keystore-file</i> 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 92.
-n or --nowarn	Suppresses the warning messages.
-o <i>file-name</i> or --out <i>file-name</i>	Specifies the output application module file where <i>file-name</i> is the name of the output file.
-P <i>key-password</i> or --passkey <i>key-password</i>	Application signing attribute, where <i>key-password</i> is the password for the private key.
-p <i>package-AID-for-classic-lib</i> or --packageaid <i>package-AID-for-classic-lib</i>	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 <i>keystore-file</i> , --storepass <i>keystore-password</i> , --passkey <i>key-password</i> , and --alias <i>alias</i> are required.
-S <i>keystore-password</i> or --storepass <i>keystore-password</i>	Application signing attribute, where <i>keystore-password</i> is the password for the keystore.
-t <i>file-type</i> or --type <i>file-type</i>	Specifies the application module file type, where <i>file-type</i> can be web, extended-applet, classic-applet, classic-lib, or extension-lib. The default value is web.
-u <i>classpath</i> or --useproxyclass <i>classpath</i>	Specifies the user-supplied proxy classes (these proxy classes are not generated by Converter).

--force Option Behavior

The --force option causes the output file to be overwritten.

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 Format

The following is the validate subcommand format:

```
packager.bat validate [-e | --exportpath path-of-export-files]
                    [--type type] [-x | --exportfile files-to-export]
                    module-file-name | module-directory-name
```

validate Subcommand Options

The validate subcommand has the options described in [TABLE 7-4](#).

TABLE 7-4 validate Subcommand Options

Options	Description
-e or --exportpath <i>path-of-export-files</i> <i>module-file-name</i> <i>module-directory-name</i>	Specifies the path to use for the files to be exported.
-t or --type	Specifies the module’s file or directory.
-x or --exportfile <i>files-to-export</i>	Specifies the type of application module or group to be validated. The type can be web, extended-applet, classic-applet, classic-lib, or extension-lib.
	Specifies the files to use for the export.

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 Format

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

```
packager.bat copyright
```

copyright Subcommand Options

There are no options for the `copyright` subcommand.

copyright Subcommand Example

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

```
packager.bat copyright
```

help Subcommand

Prints information about using subcommands.

help Subcommand Format

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

```
packager.bat help subcommand
```

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 Example

The following is an example of the help subcommand:

```
packager.bat help validate
```

Use Cases

TABLE 7-5 provides use cases for the command line arguments and describes the expected output for each.

TABLE 7-5 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 <i>filename</i> 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.

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. The two installers 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](#)
- [Description of the Installer Tool](#)
- [Card Installer Use Case](#)

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 (off-card Installer) 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 functions are performed 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 From the Command Line

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 subcommands are available 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
<code>-c oncardinstaller-url</code> or <code>--cardmanager oncardinstaller-url</code>	Specifies the location of the on-card installer where <i>oncardinstaller-url</i> represents the complete URL of the on-card installer.
<code>-C command-options-file</code> or <code>--commandoptionsfile command-options-file</code>	Optional. Specifies a file containing command line options.
<code>-n module-or-library-name</code> or <code>--name module-or-library-name</code>	Specifies the name of the module or library on the card, where <i>module-or-library-name</i> represents the module or library name.
<code>-p password</code> or <code>--password password</code>	Optional. Used when authentication is required. Specifies the password for the user set by the <code>--user</code> or <code>-u</code> subcommand, where <i>password</i> represents the required user password.

TABLE 8-1 load Options

Option	Description
-s <i>signature-file</i> or --signature <i>signature-file</i>	<p>Specifies the name of the properties file that contains the BASE64 encoded certificate and signature, where <i>signature-file</i> represents the file name.</p> <p>This file is a simple properties file with properties: signature=<i>base64-encoded-signature</i> certificate=<i>certificate-to-validate-the-module-and-digest</i></p>
-t <i>file-type</i> or --type <i>file-type</i>	<p>Specifies the type of file being loaded, where <i>file-type</i> represents one of the following values:</p> <ul style="list-style-type: none"> • web • classic-applet • extended-applet • classic-lib • extension-lib
-u <i>user-id</i> or --user <i>user-id</i>	<p>Optional. Used when authentication is required to access the card manager. Specifies the user name, where <i>user-id</i> represents the user name.</p>

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

TABLE 8-2 create Options (*Continued*)

Option	Description
-n <i>module-or-library-name</i> or --name <i>module-or-library-name</i>	Specifies the name of the module or library loaded by <code>load</code> command, where <i>module-or-library-name</i> represents the module or library name.
-p <i>password</i> or --password <i>password</i>	Optional. Used when authentication is required. Sets the password for the user specified by the <code>--user</code> or <code>-u</code> subcommand.
-u <i>user-id</i> or --user <i>user-id</i>	Optional. If authentication is required to access the card manager, specifies the authorized user, where <i>user-id</i> 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 55](#). 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
```

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
<code>-c oncardinstaller-url</code> or <code>--cardmanager oncardinstaller-url</code>	Specifies the location of the on-card installer, where <i>oncardinstaller-url</i> represents the complete URL.
<code>-C command-options-file</code> or <code>--commandoptionsfile command-options-file</code>	Optional. Specifies a file containing command line options.
<code>-i name</code> or <code>--instance name</code> or <code>-i name;name1;name2; ...</code> or <code>--instance name;name1;name2; ...</code>	Specifies the instance of the application or multiple instances of applications to be deleted, where <i>name</i> represents the instance name of the application.
<code>-p password</code> or <code>--password password</code>	Optional. Used when authentication is required. Sets the password for the user specified by the <code>--user</code> or <code>-u</code> subcommand.
<code>-u user-id</code> or <code>--user user-id</code>	Optional. If authentication is required to access the card manager, specifies the authorized user, where <i>user-id</i> 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
<code>-c oncardinstaller-url</code> or <code>--cardmanager oncardinstaller-url</code>	Specifies the location of the on-card installer, where <i>oncardinstaller-url</i> represents the complete URL.
<code>-C command-options-file</code> or <code>--commandoptionsfile command-options-file</code>	Optional. Specifies a file containing command line options.
<code>-n module-or-library-name</code> or <code>--name module-or-library-name</code>	Specifies the name of the module or library loaded by <code>load</code> command, where <i>module-name</i> represents the module or library name.
<code>-f</code> or <code>--force</code>	Optional. Forces an attempt to delete any instances before unloading.
<code>-p password</code> or <code>--password password</code>	Optional. Used when authentication is required. Sets the password for the user specified by the <code>--user</code> or <code>-u</code> subcommand.
<code>-u user-id</code> or <code>--user user-id</code>	Optional. If authentication is required to access the card manager, specifies the authorized user, where <i>user-id</i> 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
<code>-c oncardinstaller-url</code> or <code>--cardmanager oncardinstaller-url</code>	Specifies the location of the on-card installer, where <i>oncardinstaller-url</i> represents the complete URL.
<code>-C command-options-file</code> or <code>--commandoptionsfile command-options-file</code>	Optional. Specifies a file containing command line options.

TABLE 8-5 `list` Options (*Continued*)

Option	Description
<code>-d</code> or <code>--detailed</code>	Optional. Displays complete details of the application-modules, instances, and libraries.
<code>-p password</code> or <code>--password password</code>	Optional. Used when authentication is required. Sets the password for the user specified by the <code>--user</code> or <code>-u</code> subcommand.
<code>-u user-id</code> or <code>--user user-id</code>	Optional. If authentication is required to access the card manager, specifies the authorized user, where <i>user-id</i> 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 appl -t web 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.

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:

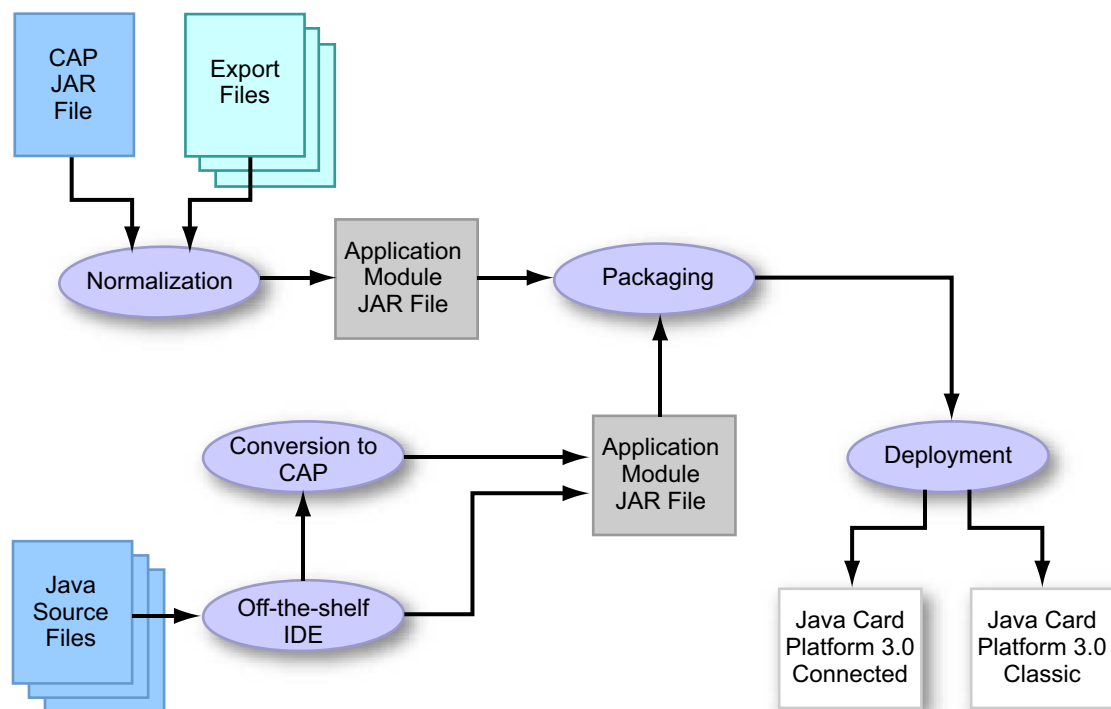
- [Generating Application Modules From Classic Applets](#)
- [Converting Class Files to CAP Files](#)

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 Generating Application Modules From Classic Applets



Running the Normalizer From the Command Line

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
<code>-C command-options-file</code> or <code>--commandoptionsfile command-options-file</code>	Optional. Specifies a file containing command line options.
<code>-i file</code> or <code>--in file</code>	Specifies the input CAP file name.
<code>-k</code> or <code>--keepall</code>	Specifies the directory to keep class files, proxy classes, and CAP components. The output format is as follows: <pre><directory> / <application classes> / proxy/ [proxy classes] / javacard/ *.cap</pre>
<code>-p path</code> or <code>--exportpath path</code>	Specifies the path of the export files used by the tool.
<code>-f file</code> or <code>--exportfile file</code>	Specifies the name of the export file.
<code>-o directory</code> or <code>--out directory</code>	(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 you can 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.

Conversion Process Sequence

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

1. 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 71](#)). Loads the class files of the Java package and creates a data structure to represent the package.

2. Subset checking.

Checks for unsupported Java features in class files.

3. Conversion.

Checks for consistency between the applet AIDs and the imported package AIDs.

4. 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 71](#)).

5. 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*.

6. Optimization.

Optimizes the bytecode.

7. 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.

8. 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 `JCDK3.0.2_ConnectedEdition\api_export_files\java\lang\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 74.](#))

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 From the Command Line

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 76](#).

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
<code>-classdir</code> <i>root- directory-of-class-hierarchy</i>	Specifies the root directory where the Converter looks for classes.
<code>-keepproxysource</code> <i>directory</i>	Cannot be used with <code>-useproxyclass</code> . Creates the proxy source files and other stub files in the specified directory. The Converter also builds CAP files as usual in the specified output directory. Supports customizing the proxy files generated by the Converter. Requests the Converter retain the intermediate proxy class source code in the specified directory and the source code of the associated stub classes representing the dependent external classes using the hierarchical directory structure of the Java package name(s).
<code>-usecapcomponents</code> <i>component</i>	User-supplied cap components
<code>-useproxyclass</code> <i>classpath</i>	User-supplied proxy classes (proxy classes not generated by converter)
<code>-i</code>	Specifies support the 32-bit integer type.
<code>-exportpath</code> <i>list-of-directories</i>	Specifies the root directories where the Converter looks for export files.
<code>-exportmap</code>	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 <code>exportpath</code> .

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

Option	Description
<code>-applet AID class-name</code>	Sets the applet AID and the class that defines the install method for the applet.
<code>-d root-directory-for-output</code>	Specifies the root directories where the Converter outputs the files.
<code>-out [CAP] [EXP] [JCA]</code>	Specifies that the Converter output the CAP file, and/or the JCA file, and/or the EXP export file.
<code>-V</code> or <code>-version</code>	Displays the Converter version number.
<code>-v</code> or <code>-verbose</code>	Enables verbose output.
<code>-help</code>	Displays the contents of this table.
<code>-nowarn</code>	Instructs the Converter to not report warning messages.
<code>-mask</code>	Identifies this package is used for a mask. Restrictions on native methods are relaxed.
<code>-debug</code>	Enables generation of debugging information.
<code>-nobanner</code>	Suppresses standard output messages.
<code>-noverify</code>	Turns off verification. Verification is default.
<code>-sign</code>	Signs the output CAP file.
<code>-keystore keystore</code>	Specifies the keystore to use in signing.
<code>-storepass storepass</code>	Specifies the keystore password.
<code>-alias alias</code>	Specifies the keystore alias to use in signing.
<code>-passkey passkey</code>	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 76 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
```

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.

The APDU I/O packages provide a convenient API for writing client-side applications that communicate with Java Card technology-enabled smart cards, see [Chapter 14](#).

This chapter includes the following sections:

- [Running the APDU Tool From the Command Line](#)
- [Using APDU Script Files](#)

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] [-k] [-mi]
[-d | --descriptiveoutput] [-o outputFile] [-p portNumber]
[-s serialPort] [-t0] [-version] [-verbose] [inputFile ...]
```

The option values and their actions are shown in [TABLE 10-1](#).

TABLE 10-1 apdutool Command Line Options

Option	Description
-d or --descriptiveoutput	Formats the output in a more readable format.
-h <i>hostname</i>	Specifies the host name on which the TCP/IP socket port is found. (See the -p option.)
-help	Displays online documentation for this command.
-k	When using preprocessor directives in an APDU script, this option generates a related preprocessed APDU script file in the same directory as the APDU script.
-mi	Optional, however, if the APDU script is switching between contacted and contactless interfaces multiple times in the same script file, this option is required.
-noatr	Suppresses outputting an ATR (answer to reset).
-nobanner	Suppresses all banner messages.
-o <i>outputFile</i>	Specifies an output file. If an output file is not specified with the -o option, output defaults to standard output.
-p <i>portNumber</i>	Specifies a TCP/IP socket port other than the default port of 9025.
-s <i>serialPort</i>	<p>Specifies the serial port to use for communication, rather than a TCP/IP socket port. For example, <i>serialPort</i> can be COM1.</p> <p>To use this option, the <code>javax.comm</code> package must be installed on your system.</p> <p>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.</p>
-t0	Runs T=0 single interface.
-version	Outputs the version information.
<i>inputFile</i>	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.

APDU Script File Commands

APDU script file commands are not case sensitive. 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 <i>Integer</i> ;	Pauses execution of the script for the number of milliseconds specified by <i>Integer</i> .
echo " <i>string</i> ";	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.

APDU Script Preprocessor Commands

APDU script supports preprocessor directives as depicted in the following script file example, `test.scr`.

```
#define walletApplet //aid/A000000062/03010C0101
#define purseApplet //aid/A000000062/03010C0102
#define walletCommand 0x80 0xCA 0x00 0x00 0x02 0xAB 0x08 0x7F
powerup;
SELECT purseApplet;
Send walletCommand to walletApplet on 19;
powerdown;
```

To check what the preprocessor has done, run the APDUTool with the -k flag to create a file named `test.scr.preprocessed` in the same directory as `test.scr`. The `test.scr.preprocessed` content then looks like this:

```
powerup;  
SELECT //aid/A000000062/03010C0102;  
Send 0x80 0xCA 0x00 0x00 0x02 0xAB 0x08 0x7F to  
//aid/A000000062/03010C0101 on 19;  
powerdown;
```

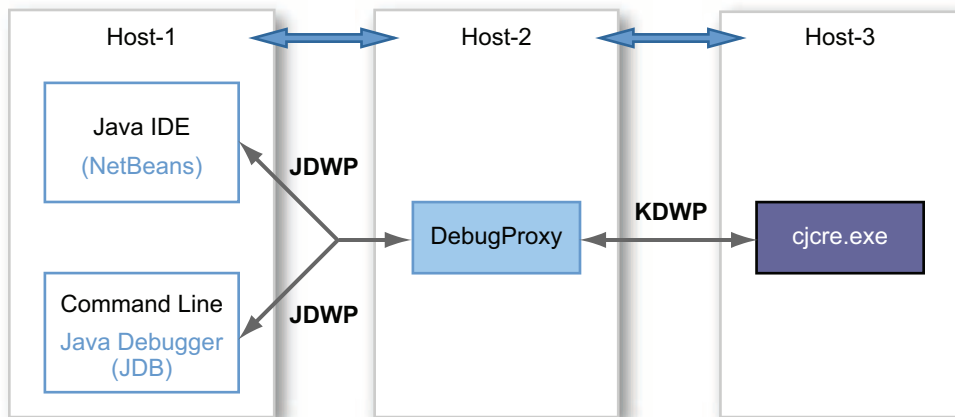

Debugging Applications

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

`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.

Running the Debugger From the Command Line

Use the Debugger tool’s functionality by starting `debugproxy` (`debugproxy.bat`), then attaching it to a Java technology-enabled IDE, and then starting `cjcre` with the `-debug` option. The file `debugproxy.bat` is located in the directory `JC_CONNECTED_HOME\bin`. Various command line options are available to configure the Debugger and `cjcre`. See “[cjcre.exe Command Line Options](#)” on [page 34](#) for additional details on the `cjcre` commands.

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

```
debugger.bat subcommand [option]
```

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

- `debug` - Debugs the specified file
- `copyright` - Displays detailed copyright notice
- `help` - Displays information about the Debugger command

debug Subcommand

Use the debug subcommand to debug a file at a specified location.

TABLE 11-1 debug Subcommand Options

Option	Description
<i>-c filelocation</i> or <i>--classpath filelocation</i>	Specifies the path of the class files to be debugged.
<i>-C command-options-file</i> or <i>--commandoptionsfile command-options-file</i>	Optional. Specifies a file containing command line options.

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 Debugger command. Options are used with the `help` subcommand to specify the information that is displayed about each subcommand. For example, to display detailed help about the debug subcommand, type:

```
debugproxy.bat help debug
```

Debugging a Java Card 3 Platform Application

This section describes how to debug an application using the development kit command line tools.

Compile the Source Code

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

Start the Debugger

The Debugger needs to know the location of class files being debugged. The Debugger can be run from the command line using the following syntax, for example:

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

When starting `debugproxy`, you can include the `-c` (or `--classpath`) option in the command line to specify the path of the class files to be debugged. In this example, `myapp.war` is the location of the class files to be debugged.

Attach the Debugger to the IDE

This procedure is performed from within the IDE and so the details depend on the IDE used. If your IDE requires the Debugger to be attached, refer to the documentation provided with the IDE.

Run `cjcre.exe` With `-debug` Option

This `-debug` option of `cjcre` enables debugging functionality in `cjcre`. Without this option, debugging functionality is disabled in `cjcre`. See [“`cjcre.exe` Command Line Options” on page 34](#) for additional details on the `cjcre` commands.

Set Break Points

Break points must be set in the application source code. The exact procedure depends on the IDE used. The following steps are typical for most IDEs. Refer to the documentation provided with your IDE for specific instructions.

1. **Display the source code of the application in the IDE.**
2. **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.
3. **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.

PART II Programming With the Development Kit

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

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 development 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](#)
- [Creating Custom Protection Domains](#)
- [Configuring SSL Support](#)

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 preceding 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 pin for this authenticator.
- The `digest` property is set to `true` or `false` depending on if the provided authenticator is of type `digest` or not.

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 preceding 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 keystore can be created by using the `keytool` command to generate the certificates and private keys. The `keytool` command runs in batch mode without prompting for input values.

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"
```

Configuring SSL Support

An SSL implementation requires four algorithms:

- digital signature
- key establishment
- bulk encryption
- message digest

Note – It is beyond the scope of this document to fully describe SSL configuration and setup. There are many excellent books on this subject, and we direct advanced users to this literature.

Adding SSL Support

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 `lib\config.properties`. In the file `lib\config.properties`, the following properties must be added to add SSL support:

- `ssl.trusted.ca.#` - Index-based property to specify BASE-64 encoded certificates of the CA root that Java Card 3 RI trusts. Used in the normal SSL handshake. This property is index based. User can configure multiple CA roots by appending the index at the end of the property, such as `ssl.trusted.ca.0`, `ssl.trusted.ca.1`, and `ssl.trusted.ca.2`. The indexes are assumed to be in sequence starting with zero. When the sequence is broken, it is assumed the properties have ended.
- `ssl.accepted.issuer.#` - Index based property to specify BASE-64 encoded accepted issuer's certificates. Used only in client authentication handshake. User can configure multiple issuer certificates by appending the index at the end of the property, such as `ssl.accepted.issuer.0`, `ssl.accepted.issuer.1`, and `ssl.accepted.issuer.2`. The indexes are assumed to be in sequence starting with zero. When the sequence is broken, it is assumed the properties have ended.
- `ssl.selfIdentityAsServer` - BASE-64 encoded server certificate. This is the certificate that the Java Card 3 platform uses to identify itself when operating in SSL server mode.

- `ssl.selfIdentitySSLPrivateKeyExp` - BASE-64 encoded private key (exponent) of the server certificate.
- `ssl.selfIdentitySSLPrivateKeyMod` - BASE-64 encoded private key (modulus) of the server certificate.
- `PSKIdentityHint` - String value used in the PSK protocol as a server side identity hint.

Custom Certificates and Keys

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 a WAR file and are used in SSL and HTTPS transactions.

▼ 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
```

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 development 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 customized 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](#)
- [Contents of JC_CONNECTED_HOME\src Folder](#)
- [Running the ROMizer Tool From the Command Line](#)
- [Building a Custom `cjcre.exe`](#)

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 these requirements.

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.
- **aplImpl.jar**
- **bat.template**
- **crypto.jar**

Running the ROMizer Tool From the Command Line

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 file system, so that these files are available during the execution of the application. See [“Building a Custom `cjcre.exe`” on page 100](#) for detailed description of using the ROMizer tool.

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

```
romizer.bat subcommand [options]
```

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

- `romize` - Romizes the system class files and application modules
- `copyright` - Displays detailed copyright notice
- `help` - Displays information about the ROMizer command

romize Subcommand

Use the `romize` subcommand and its options to romize the system class files and application modules. Options are used with the `romize` subcommand to specify files and directories.

romize Subcommand Options

TABLE 13-1 identifies the `romize` subcommand options and provides their descriptions.

TABLE 13-1 `romize` Subcommand Options

Option	Description
<code>-a apps-filename</code> or <code>--apps apps-filename</code>	Specifies the file that contains the list of applications to be masked.
<code>-C command-options-file</code> or <code>--commandoptionsfile command-options-file</code>	Optional. Specifies a file containing command line options.
<code>-e EEPROM- filename</code> or <code>--e2pfile EEPROM- filename</code>	Specifies the file where the initial eeprom file is written.
<code>-o ROM-output-filename</code> or <code>--out ROM-output-filename</code>	Specifies the file where the mask is written.

romize Subcommand Example

Either of the following commands will run the ROMizer tool:

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

or

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

In the previous examples, the following options are used:

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

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

See [“Romizer Tool Output” on page 99](#) 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 99](#) for a description of the configuration file.

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

For example, to see detailed help about the ROMizer tool, type:

```
romizer.bat help romize
```

Apps list File Contents

The apps list file contains information about applications that need 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
GCFCClient.war -t web -s key2.txt -n gcfcapp
```

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.

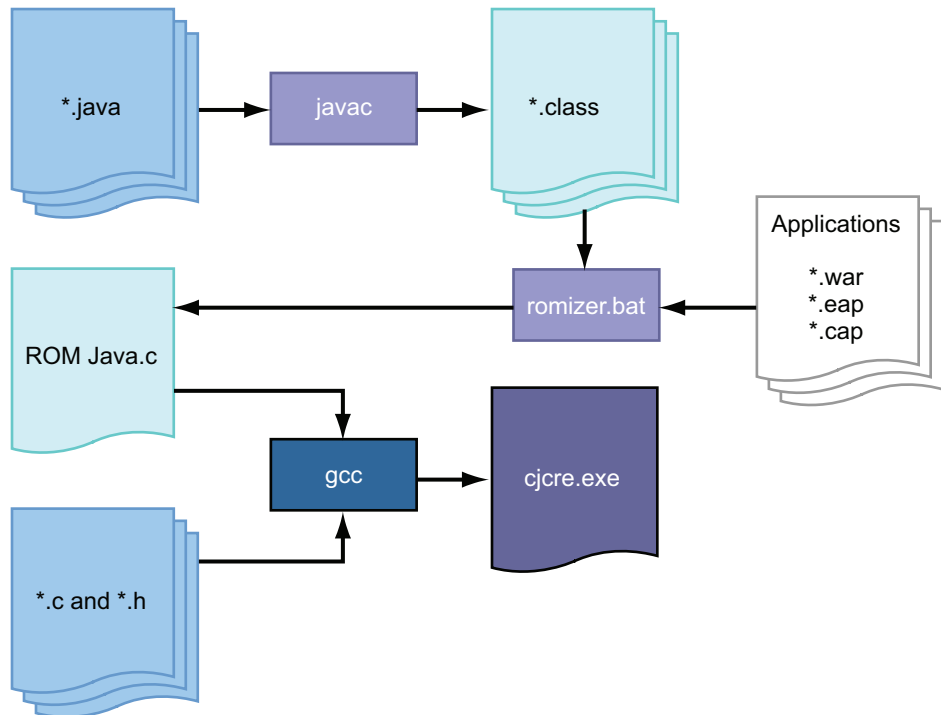
Developers can modify the RI by adding or modifying the reference implementation code and using the ROMizer tool. RI consists of `.java` and C source files. The core VM is written in C programming language and rest of the API and supported implementation is written in the Java programming language. The 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 file system, so that these files are available during the execution of the application.

FIGURE 13-1 illustrates the procedure of building the `cjcre.exe` from sources.

FIGURE 13-1 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 101](#)). 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 From the Command Line” on page 102](#).

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 `kδwp` code. Default is 1. If set to 0, then `cjcre` can not be used to debug the applications.

- `ENABLE_LOGGING=1`

Enables the logging messages that are printed by using `-loggerlevel=<value>`. Default is 1. If set to 0, then even if `-loggerlevel` is set to `all`, no logging messages are printed from native code.

- `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 From the Command Line

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` command as shown:

```
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.

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`.

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.

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 located in this bundle at `JC_Connected_HOME\docs\apduio\`.

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 107](#) 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 and T=1 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`

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`) 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`) 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`, 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 107](#).

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 [“Prerequisites to Installing the Development Kit” on page 11](#) for details on how to obtain this library.

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)
```

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:

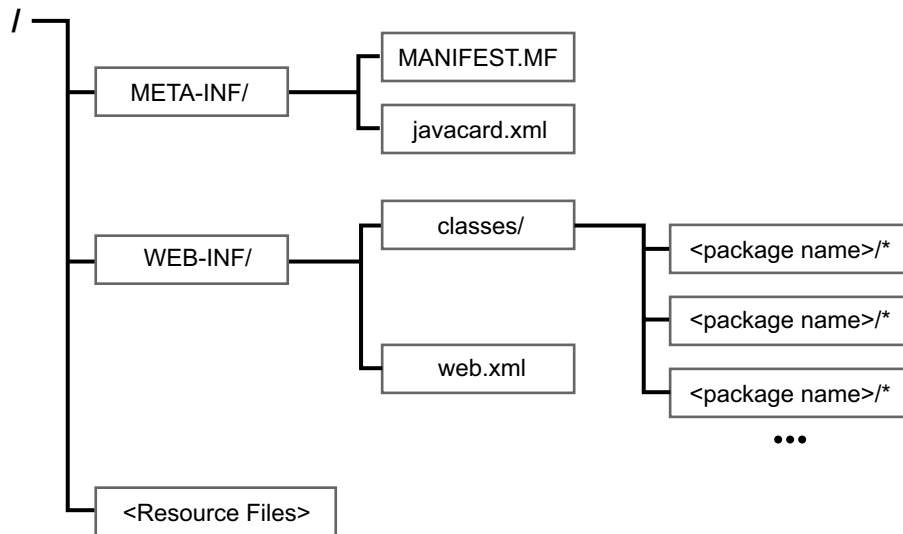
- [Web Application Module Format](#)
- [Extended Applet Application Module Distribution Format](#)
- [Classic Applet Application Module Format](#)
- [Extension Library Format](#)
- [Classic Library Format](#)

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

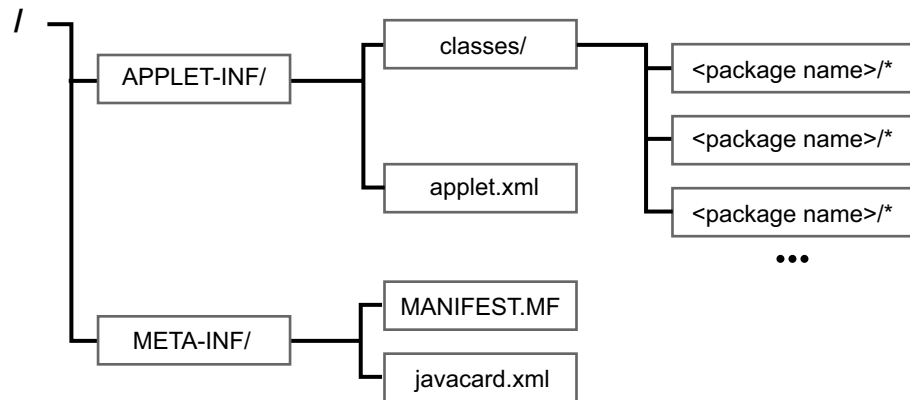


See the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for specific details about the web application module format.

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



See the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for specific details about the extended applet application module format.

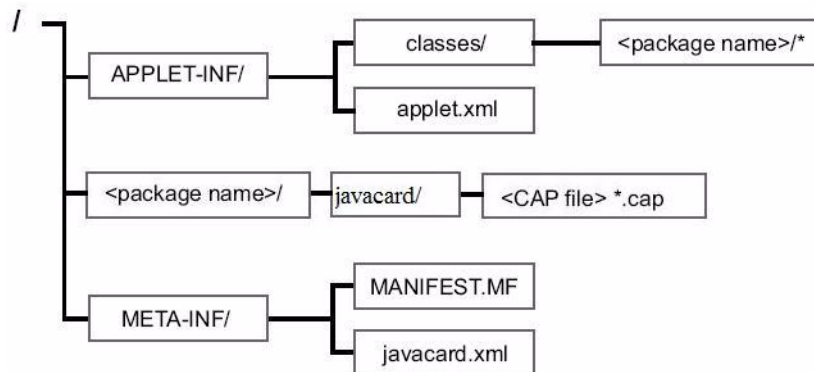
Classic Applet Application Module Format

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.

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.

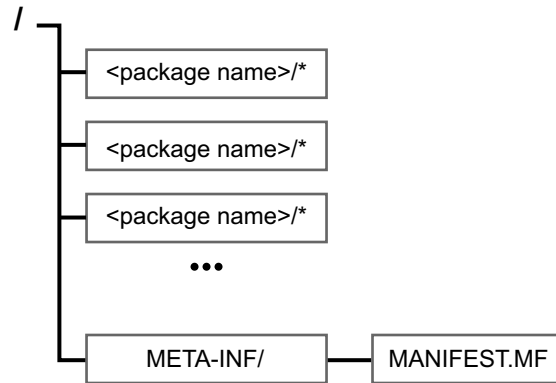
FIGURE A-3 Classic Applet Application Module



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 File Format



See the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for specific details about the extension library format.

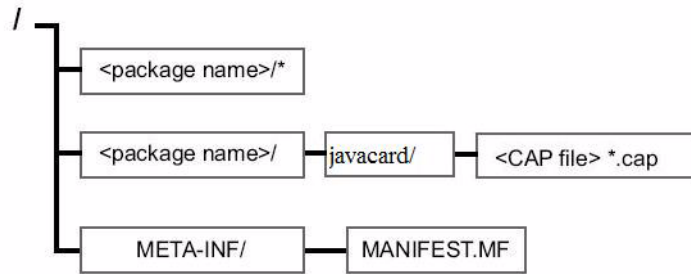
Classic Library Format

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*.

See the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Connected Edition* for specific details about the classic library format.

FIGURE A-5 Classic Library Distribution Format



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)	<p>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.</p> <p>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.</p> <p>Applet instance URIs are constructed from their applet instance AID using the "aid" registry-based namespace authority as follows:</p> <pre>//aid/<RID>/<PIX></pre> <p>where <RID> (resource identifier) and <PIX> (proprietary identifier extension) are components of the AID.</p>
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 <i>descriptor</i> .
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 piece 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 <i>module</i> or an application group, that was loaded by <i>load</i> , 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 <i>create</i> 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 <i>String</i> 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 as 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 <i>module</i> 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	<p>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.</p> <p>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.</p>
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 <code>Runnable</code> 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 <i>web application container</i> .

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 <i>Java Servlet Specification, Connected Edition</i> .
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 <i>currently active context</i> .
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	<p>classes whose instances can have their ownership transferred to a context different from their currently owning context. Transferable classes are of two types:</p> <p>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 <code>Boolean</code> and literal <code>String</code> objects.</p> <p>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 <code>String</code> objects.</p>
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 <i>load</i> needs to be removed completely from the card. By default, if there are some instance(s) created, then unload will fail. Optional <code>-f</code> (or <code>-force</code>) 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	<p>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.</p> <p>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 <i>Java Servlet Specification, Connected Edition</i>.</p>
web application archive	<p>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.</p> <p>A web application archive file is identified by the <code>.war</code> extension and is often referred to as a WAR file. A new extension is used instead of <code>.jar</code> 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 <i>Java Servlet Specification, Connected Edition</i>.</p>
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

A

- AID (application identifier), 117
- APDU, 77
 - script file commands, 80
 - script files, 79
- APDU I/O, xvii, 105
- APDU tool
 - `apdutool.bat`, 77
 - command line options, 78
 - command line syntax, 77
 - description, 77
 - running, 77
- APDU-based application environment, 117
- `apdutool.bat`, 77
- API, 118
- applet, 118
- applet application, 118
- applet container, 118
- applet framework, 118
- application descriptor, 118
- application developer, 118
- application group, 118
- application module formats, 111
- Application Protocol Data Unit, 77
- application URI, 118
- architecture
 - Debugger tool, 83

C

- C Java Card Runtime Environment, 33
- CAP file, 113

- suppressing output, 73
- card installer
 - off-card Installer tool, 51
 - on-card installer, 51
 - use case, 63
- `cjcre.exe`, 6
 - starting, 33
- classic applet, 118
- classic applet application module
 - distribution format, 113
- Classic Edition, 118
- classic library
 - distribution format, 115
- classic_applets sample, 31
- command configuration file, 76
- command line examples
 - Compiler tool, 39
- command line options
 - APDU tool, 78
- command line syntax
 - Compiler tool, 38
 - Packager tool, 43
- Compiler tool
 - command line examples, 39
 - command line options, 37
 - command line syntax, 38
 - description, 37
 - running, 37
 - unsupported features, 37
- configuring
 - Debugger tool, 84
- Connected Edition, 118

- Converter, 119
- Converter tool
 - command configuration file, 76
 - creating a debug.msk file, 72
 - described, 69
 - input file naming conventions, 73
 - invoking the off-card verifier, 72
 - output, 69
 - output file naming conventions, 73
- converting
 - Java class files, 69

D

- debug.msk file
 - creating, 72
- Debugger tool
 - architecture, 83
 - configuring, 84
 - description, 83
 - running, 86
- description
 - Compiler tool, 37
 - Debugger tool, 83
 - Installer tool, 53
 - javacardc.bat, 37
 - on-card installer, 51
 - samples, 27
 - web samples, 29
- developing applications, 23
- Development Kit
 - additional software (required), 8
 - bundle, 6
 - Connected Edition features, 4
 - installation, 11
 - Normalizer tool, 65
 - samples, 8
 - system requirements, 8
 - tools, 7
 - uninstalling, 19
- distribution format, 119
 - classic applet application module, 113
 - classic library, 115
 - extended applet application module, 113
 - extension library, 114

E

- EEPROM, 33

- export file
 - loading, 71
- export map
 - specifying, 71
- extended applet, 119
- extended applet application module
 - distribution format, 113
- extended_applets sample, 31
- extension library
 - distribution format, 114

F

- functionality
 - Installer tool, 53
 - on-card installer, 52

I

- input file
 - naming conventions for the Converter tool, 73
- input files
 - suppressing verification, 72
 - verifying, 72
- installation of Development Kit, 11
- Installer tool, 53
 - description, 53
 - functionality, 53
 - installer.bat, 53
 - running, 53
 - subcommands, 53
- installer.bat, 53

J

- Java Card 3 platform
 - bundle, 6
 - developing applications, 23
 - Reference Implementation, 6
- Java Card Runtime Environment, 33
- Java Card runtime environment
 - starting, 33
- Java Card TCK, 9
- Java Debug Wire Protocol, 84
- javac, 37
- javacardc.bat, 24, 37
 - description, 37
- JDK compiler, 37
- JDWP, 84

K

KDWP, 84
KVM Debug Wire Protocol, 84

L

library formats, 111
loading applications, 51

M

managing applications, 51

N

Normalizer tool, 65
`normalizer.bat`, 65

O

off-card verifier
 invoking, 72
 suppressing verification, 72
on-card installer
 description, 51
 functionality, 52
 operation, 52
operation
 on-card installer, 52
 Packager tool, 41
options
 Packager tool, 41
output file
 naming conventions for the Converter tool, 73
output files
 suppressing verification, 72
 verifying, 72

P

Packager tool
 command line syntax, 43
 operation, 41
 options, 41
 output conditions, 42
 `packager.bat`, 43
 signing a module, 43
 subcommands, 43
`packager.bat`, 43
protected content, 121

R

Reference Implementation, 6, 33
 starting, 33
reimplementing a package or method, 71
RI, 6
running
 Compiler tool, 37
 Debugger tool, 86
 Installer tool, 53

S

samples, 8
 classic_applets, 31
 description, 27
 extended_applets, 31
 web, 29
script file commands
 APDU, 80
script files
 APDU, 79
signing a module, 43
starting
 `cjcre.exe`, 33
subcommands
 Installer tool, 53

T

TCK *see* Java Card TCK
Technology Compatibility Kit *see* Java Card TCK
thread's active context, 123
tools, 7

U

uninstalling the Development Kit, 19
use-case
 card installer, 63

W

web samples
 description, 29

